Automated Inductive Verification of Parameterized Protocols*

Abhik Roychoudhury[1] and I.V. Ramakrishnan[2]

[1] School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543. abhik@comp.nus.edu.sg
[2] Dept. of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA. ram@cs.sunysb.edu

Abstract. 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. We employ a program transformation based proof methodology to automate such induction proofs. Our proof technique is geared to automate nested induction proofs which do not involve strengthening of induction hypothesis. Based on this technique, we have designed and implemented a prover for parameterized protocols. The prover has been used to automatically verify safety properties of parameterized cache coherence protocols, including broadcast protocols and protocols with global conditions. Furthermore we also describe its successful use in verifying mutual exclusion in the Java Meta-Locking Algorithm, developed recently by Sun Microsystems for ensuring secure access of Java objects by an arbitrary number of Java threads.

1 Introduction

There is a growing interest in verification of *parameterized* concurrent systems since they occur widely in computing *e.g.* in distributed algorithms. 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 [6, 21, 24] 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 [2]. Thus, verification of parameterized networks is often accomplished via theorem proving [14, 17, 22], or by synthesizing network invariants [7, 19, 28]. Alternatively, one can identify subclasses of parameterized systems for

* This work was partially supported by NSF grants CCR-9711386, CCR-9876242 and EIA-9705998. The first author was a Ph.D. student at SUNY Stony Brook during part of this work.

which verification is decidable [9, 10, 13, 15]. Another approach [8, 11, 12, 16, 18] finitely represents the state space of a parameterized system and applies (symbolic) model checking over this finite representation.

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 [25] 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 the equivalence of predicates in this program. The predicate equivalences are then established by *transforming* the predicates. The proof of semantic equivalence of two predicates proceeds automatically by a routine induction on the structure of their transformed definitions. One of our transformations (unfolding) represents resolution and performs on-the-fly model checking. The others (*e.g.* folding) represent deductive reasoning. The application of these transformations are arbitrarily interleaved in the verification proof of a parameterized system. This allows our framework to tightly integrate algorithmic and deductive verification.

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. We construct an automatic and programmable first order logic based prover with limited deductive capability. The prover can also exploit knowledge of network topology (chain, tree etc) to facilitate convergence of proofs.
2. Our program transformation based technique produces induction proofs. We clarify the connection between our transformations and inductive reasoning.
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. 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 [4]. This is the first machine generated proof of the algorithm which is parameterized by the number of threads.

The rest of the paper is organized as follows. Section 2 presents an overview of our program transformation based proof technique for parameterized systems presented in [25]. Section 3 clarifies the connection between program transformations and inductive reasoning. Section 4 discusses the functioning of our automated prover for parameterized protocols. Section 5 presents the successful

```
nat(0).
nat(s(X)) :- nat(X).
trans(s(X), X).
p(0).
```
System Description

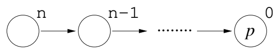
```
efp(S) :- p(S).
efp(S) :- trans(S, T), efp(T).
thm(X) :- nat(X), efp(X).
```
Property Description

Fig. 1. Proving liveness in infinite chain

use of our prover in verifying parameterized cache coherence protocols as well as the Java meta-locking algorithm. Finally, Section 6 concludes the paper with related work and possible directions for future research.

2 Overview

In this section, we recapitulate our core technique for inductive verification [25] through a very simple example. Let us consider an unbounded length chain whose states are numbered n, $n-1$, ..., 0. Further suppose that the start state is n, the end state is 0 and a proposition `p` is true in state 0. Suppose we want to prove the CTL property **EF** `p` for every state in the chain. Alternatively, we can view this chain as an infinite family of finite chains of length 0,1,2,... and the proof obligation as proving **EF** `p` for every start state of the infinite family. Either way, our proof obligation amounts to $\forall n \in \mathbb{N}\ n \models$ **EF** `p`. Our proof technique dispenses this obligation by an induction on n.



Encoding the problem In the above example, the states are captured by natural numbers which we represent by a logic program predicate `nat` (refer Figure 1; the term `s(K)` denotes the number `K+1`). The transition relation is captured by a binary predicate `trans` s.t. `trans(S, T)` is true iff there exists a transition from state `S` to state `T`. [1] The temporal property **EF** `p` is encoded as a unary predicate `efp` s.t. for any state `S`, `efp(S)` $\Leftrightarrow$ `S` $\models$ **EF** `p`. The first clause of `efp` succeeds for states in which proposition `p` holds. The second clause of `efp` checks if a state satisfying `p` is reachable after a finite sequence of transitions. Thus $\forall n \in \mathbb{N}\ n \models$ **EF**`p` iff $\forall$`X` `nat(X)` $\Rightarrow$ `efp(X)`. Moreover this holds if $\forall$`X` `thm(X)` $\Leftrightarrow$ `nat(X)` in P_0.

Proof by Program Transformations We perform inductive verification via logic program transformations using the following steps. A detailed technical presentation of this proof technique appears in [25].

1. Encode the system and property description as a logic program P_0.

[1] For realistic parameterized systems, the global transition relation is encoded recursively in terms of local transition relations of constituent processes; see section 4.

2. Convert the verification proof obligation to predicate equivalence proof obligation(s) of the form $P_0 \vdash \mathtt{p} \equiv \mathtt{q}$ ($\mathtt{p}, \mathtt{q}$ are predicates)
3. Construct a transformation sequence $P_0, P_1, \ldots, P_k$ s.t.
   (a) Semantics of P_0 = Semantics of P_k
   (b) from the syntax of P_k we infer $P_k \vdash \mathtt{p} \equiv \mathtt{q}$

For our running example, the logic program encoding P_0 appears in Figure 1. We have reduced the verification proof obligation to showing the predicate equivalence $P_0 \vdash \mathtt{thm} \equiv \mathtt{nat}$. We then transform program P_0 to obtain a program P_k where `thm` and `nat` are defined as follows.

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

Thus, since the transformed definitions of `thm` and `nat` are "isomorphic", their semantic equivalence can be inferred from syntax. In general, we have a sufficient condition called *syntactic equivalence* which is checkable in polynomial time w.r.t. program size (refer [25, 27] for a formal definition).

Note that inferring the semantic equivalence of `thm` and `nat` based on the syntax of their transformed definitions in program P_k proceeds by induction on the "structure" of their definitions in P_k (which in this example amounts to an induction on the length of the chain). The program transformations employed in constructing the sequence $P_0, P_1, \ldots, P_k$ correspond to different parts of this induction proof. In the next section, we will clarify this connection between program transformations and inductive reasoning.

3 Program Transformations for Inductive Verification

Unfold/fold program transformations 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 2. 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. A formal definition of the unfold/fold transformation rules, along with a proof of semantics preservation of any interleaved application of the rules, appears in [26].

An example of inductive verification We now apply these transformations to the definition of `thm` in the program P_0 shown in Figure 1. First we unfold `nat(X)` in the definition clause of `thm` to obtain the following clauses. This unfolding step corresponds to uncovering the schema on which we induct, *i.e.* the schema of natural numbers.

```
thm(0) :- efp(0).
thm(s(X)) :- nat(X), efp(s(X)).
```

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

Program P_0'

$\xrightarrow{Unf.}$

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

Program P_1'

$\xrightarrow{Fold}$

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

Program P_2'

Fig. 2. Illustration of unfold/fold transformations

We now repeatedly unfold `efp(0)`. These steps correspond to showing the base case of our induction proof. Note that showing the truth of `efp(0)` is a finite state verification problem, and the unfolding steps employed to establish this exactly correspond to on-the-fly model checking. We obtain:

```
thm(0).
thm(s(X)) :- nat(X), efp(s(X)).
```

We repeatedly unfold `efp(s(X))` in the second clause of `thm`. These steps correspond to finite part of the induction step, *i.e.* the reasoning that allows us to infer $n+1 \models$ **EF** `p` provided the induction hypothesis $n \models$ **EF** `p` holds. We get

```
thm(0).
thm(s(X)) :- nat(X), efp(X).
```

Finally, we fold the body of the second clause of `thm` above using the original definition of `thm` in P_0. Application of this folding step enables us to recognize the induction hypothesis (`thm(X)` in this case) in the induction proof.

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

The semantic equivalence of `thm` and `nat` can now be shown from their syntax (by a routine induction on the structure of their definitions). This completes the verification (by induction on `nat`).

What kind of induction? Since unfolding represents a resolution step, it can be used to prove the base case and the finite part of the induction step. However, folding recognizes the occurrence of clauses of a predicate `p` *in an earlier program* $P_j (j \leq i)$, within the current program P_i. Thus, folding is *not* the reverse of unfolding. It can be used to remember the induction hypothesis and recognize its occurrence. Application of unfold/fold transformations constructs induction proofs which proceed without strengthening of hypothesis. This is because the folding rule only recognizes instances of an earlier definition of a predicate, and does not apply any generalization. In the next section, we will discuss how our transformation based proof technique can support nested induction proofs.

4 An Automated Prover for Parameterized Protocols

The inductive reasoning accomplished by our transformation based proof technique has been exploited to build an automated prover for parameterized protocols. Note that our program transformation based technique for proving predicate equivalences can be readily extended to prove predicate implication proof obligations of the form $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$.[2]

Since our transformations operate on *definite logic programs* (logic programs without negation), we only verify temporal properties with either the least or the greatest fixed point operator. For the rest of the paper, we restrict our attention to only proof of *invariants*.

4.1 System and Property Specification

To use our prover, first the initial states and the transition relation of the parameterized system are specified as two logic program predicates `gen` and `trans`. The global states of the parameterized system are represented by unbounded terms, and `gen`, `trans` are predicates over these terms. The recursive structure of `gen` and `trans` depends on the topology of the parameterized network being verified. For example, consider a network of similar processes where any process may perform an autonomous action or communicate with any other process. We can model the global state of this parameterized network as an unbounded list of the local states of the individual processes. 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)`/`out(A)`) 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 clause recursively searches the global state representation until one of the first three rules can be applied. The third clause allows any process to make an autonomous action. The first and second clauses correspond to the scenario where *any* two processes communicate with each other. In particular, the first (second) clause of `trans` allows a process to make an `in(A)` (`out(A)`) action and

[2] The proof obligation $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$ formally means: for all ground substitutions θ we have $\mathtt{p}(\overline{\mathtt{X}})\theta \in M(P_0) \Rightarrow \mathtt{q}(\overline{\mathtt{X}})\theta \in M(P_0)$ 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.

invokes `trans_rest` to recursively search for another process which makes the corresponding `out(A)` (`in(A)`) action.

A safety property, denoted in CTL as **AG** ¬`bad` can be verified 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 is shown by establishing (1) `bad_dest` ⇒ `bad_src`, and (2) `bad_start` ⇒ `false` where the predicates `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(X), bad(X).
```

4.2 Controlling the Proof Search

A skeleton of the proof search conducted by our prover is given below. Given a predicate implication $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$ the prover proceeds as follows.

1. Repeatedly *unfold* the clauses of `p` and `q` according to an unfolding strategy which is is designed to guarantee termination.
2. Apply *folding* steps to the unfolded clauses of `p`, `q`.
3. (a) *Compare* the transformed definitions of `p` and `q` to compute a finite set $\{(\mathtt{p}_1, \mathtt{q}_1), \ldots, (\mathtt{p}_k, \mathtt{q}_k)\}$ s.t. proving $\bigwedge_{1 \leq l \leq k} P_0 \vdash \mathtt{p}_l \Rightarrow \mathtt{q}_l$ 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).
   (b) Prove $P_0 \vdash \mathtt{p}_1 \Rightarrow \mathtt{q}_1, \ldots, P_0 \vdash \mathtt{p}_k \Rightarrow \mathtt{q}_k$ via program transformations.

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. Note that for the example in Figure 1, steps (1) and (2) were sufficient to complete the proof and therefore step (3) did not result in any nested proof obligations.

The above proof search skeleton forms the core of our automated prover which has been implemented on top of the XSB logic programming system [29]. Note that the proof search skeleton is nondeterministic *i.e.* several unfolds or several folds may be applicable at some step. For space considerations we omit a full discussion on how a transformation step is selected among all applicable transformations The interested reader is referred [27] (Chapter 6) for a detailed discussion, including a description of how the unfolding strategy guarantees termination. However, note that the prover allows the user to provide some problem-specific information at the *beginning* of the proof, namely *(i) Network topology* (linear, tree etc.) of the parameterized system, *(ii)* which *predicates* in the program encode the *safety property* being verified. This user guidance enables the prover to select the transformation steps in the proof attempt (which then proceeds without *any* user interaction). Below we illustrate illustrate how the user-provided information guides the prover's proof search.

Network Topology The communication pattern between the different constituent processes of a parameterized network is called its network topology. To illustrate the role of network topology in our proof search let us suppose that we are proving `bad_dest`$\Rightarrow$ `bad_src` (refer Section 4.1). 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 binary tree network whose left and right subtrees do not communicate directly. Let the clauses of `bad_dest` and `bad_src` after unfolding and folding be:

```
bad_dest(f(root1,L1,R1), f(root2,L2,R2)) :- p(L1,L2), q(R1,R2).
bad_src(f(root1,L1,R1), f(root2,L2,R2)) :- p'(L1,L2), q'(R1,R2).
```

then by default $\mathtt{p} \wedge \mathtt{q} \Rightarrow \mathtt{p}' \wedge \mathtt{q}'$ needs to be proved to establish `bad_dest` $\Rightarrow$ `bad_src`. Instead, the prover recognizes that `p`, $\mathtt{p}'$ (`q`, $\mathtt{q}'$) are predicates defined over left (right) subtrees. 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 (whose transitions are independent of each other): $\mathtt{p} \Rightarrow \mathtt{p}'$ and $\mathtt{q} \Rightarrow \mathtt{q}'$. In other words, knowledge of transition system is exploited by the prover to choose nested proof obligations (as a heuristic for faster convergence of the proof attempt).

Predicates encoding temporal property By knowing which program predicates encode the safety property, the prover avoids unfolding steps which may disable deductive steps leading to a proof. To see how, note that the logic program encoding of a verification problem for parameterized systems inherently has a "*producer-consumer*" nature. For example to prove transition invariance, we need to show `bad_dest` $\Rightarrow$ `bad_src` (refer Section 4.1) where `bad_dest(S, T) :- trans(S, T), bad(T).` The system description predicate (`trans`) is the producer, since by unfolding it produces instantiations for variable `T`. Suppose by unfolding `trans(S, T)` we instantiate variable `T` to a term $\overline{\mathtt{t}}$ representing global states of the parameterized family. Now, by unfolding $\mathtt{bad}(\overline{\mathtt{t}})$ we intend to *test* whether `bad` holds in states represented by $\overline{\mathtt{t}}$. In other words, the property description predicate is a consumer. Unfolding of $\mathtt{bad}(\overline{\mathtt{t}})$ should consume the instantiation $\overline{\mathtt{t}}$, rather than producing further instantiation via unification. Hence our prover incorporates heuristics to prevent unfoldings of property description predicates which result in instantiation of variables. Such unfolding steps can disable deductive steps converging to a proof e.g. folding of conjunction of `trans` and `bad` to `bad_dest`. The user-provided information tells us which predicates encode the safety property and enables us to identify these unfolding steps.

In general, to prove $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$, we first repeatedly unfold the clauses of `p` and `q`. Deductive steps like folding are applied subsequently. Therefore, it is possible to apply finite sequence of unfolding steps $P_0 \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_n$ s.t. a folding step applicable in program P_i which leads to a proof of $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$ is disabled in P_n. One way to prevent such disabling of deductive steps is to check for applicable deductive steps ahead of unfolding steps. However, this would add theorem proving overheads to model checking (model checking is accomplished by unfolding). Our goal is to perform zero-overhead theorem proving, where deductive steps are never applied if model checking alone can complete the verification task. The other solution is to incorporate heuristics for identifying unfolding steps which disable deductive steps. This approach is taken in our prover. The prover prevents any unfolding of a predicate encoding temporal property which generates variable instantiations.

5 Case Studies and Experimental Results

In this section, we first illustrate the use of our prover in proving mutual exclusion of the Java meta-locking algorithm [1]. Then, in section 5.2 we present the experimental results obtained 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.

5.1 Mutual Exclusion of Java Meta-Lock

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. 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. Model checking has previously been used to verify *instances* of the Java Meta-locking algorithm, obtained by fixing the number of threads [4].

The formal model of the algorithm consists of asynchronous parallel composition (in the sense of Milner's CCS) of an object process, a *hand-off* process and an arbitrary number of thread processes. To completely eliminate busy waiting by any thread, the algorithm performs a *race* between a thread acquiring the meta-lock and the thread releasing the meta-lock. The winner of this race is determined by the hand-off process, which serves as an arbiter.

We model the object process without the synchronization data since we are only interested in verifying mutually exclusive access of this data. Apart from the synchronization data, the meta-locking algorithm *implicitly* maintains another queue : the queue of threads currently contending for the meta-lock to access the synchronization data. However, for verifying mutual exclusion we only model the length of this queue. The local state of the object process therefore contains a natural number, the number of threads waiting for the meta-lock. This makes the object an infinite state system.

The thread and the hand-off processes are finite state systems. A thread synchronizes with the object to express its intention of acquiring/releasing the meta-lock. A thread that faces no contention from other threads while acquiring/releasing the meta-lock is said to execute the *fast path*. Otherwise, it executes the *slow path* where it gets access to the meta-lock in a FIFO discipline. When its turn comes, it is woken up by the hand-off process which receives acquisition/release requests from the acquiring/releasing threads.

We straightforwardly encoded the state representations and the transitions in the formal model of the protocol as a logic program. The modeling of the protocol took less than a week, with the help of a colleague who had previously modeled it in a CCS-like language for model checking. A global state in the logic program encoding is a 3-tuple `(Th, Obj, H)` where `Th` is an unbounded list of thread states, `Obj` is a state of the object process (containing an unbounded term representing a natural number) and `H` is a state of the hand-off process.

Our prover automatically proves transition invariance for a strengthening of the mutual exclusion invariant (the mutual exclusion invariant states that < 2 threads own the meta-lock). This strengthening was done manually, by reasoning about the local states of the hand-off and object processes. This is because the mutual exclusion invariant is not preserved by every transition (even though a state violating mutual exclusion is never reachable from the initial state of the algorithm). Thus, to prove mutual exclusion by transition invariance the invariant to be proved must be strengthened. Since our inductive prover cannot strengthen induction hypothesis in a proof, the strengthening was done manually. However, once the strengthened invariant is fed, the proof proceeds completely automatically. The timings and the number of proof steps are reported in Table 1 and further discussed in next section.

Recall from section 4.1 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.

5.2 Experimental Results

Table 1 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.

Protocol	Invariant	Time(secs)	# Unfolding	#Deductive
Meta-Lock	#owner + #handout < 2	129.8	1981	311
Mesi	#m + #e < 2	3.2	325	69
	#m + #e = 0 ∨ #s = 0	2.9	308	63
Illinois	#dirty < 2	35.7	2501	137
Berkeley RISC	#dirty < 2	6.8	503	146
Tree-cache	#bus_with_data < 2	9.9	178	18

Table 1. Summary of protocol verification results

In the table, we have used the following notational shorthand: $\#s$ denotes the number of processes in local state s. *Mesi* and *Berkeley RISC* are single bus broadcast protocols [3, 11, 12]. *Illinois* is a single bus cache coherence protocol with global conditions which cannot be modeled as a broadcast protocol [8, 23]. *Tree-cache* is a binary tree network which simulates the interactions between the cache agents in a hierarchical cache coherence protocol [27].

The running times of our prover are slower than the times for verifying single bus cache coherence protocols reported in [8]. Unlike [8], our prover implements the proof search via meta-programming. It is feasible to implement our proof search at the level of the underlying abstract machine thereby improving efficiency. Moreover, note that the abstraction based technique of [8] is not suitable for verifying parameterized tree networks.

The number of deductive steps in our proofs is consistently small compared to the number of unfolding steps, since our proof search strategy applies deductive steps lazily. Due to its tree topology, the state representation of *Tree-cache* has a different term structure. This results in a larger running time with fewer transformation steps as compared to other cache coherence protocols. Finally, the proof of Java meta-locking algorithm involves nested induction over both control and data of the protocol. This increases the number of nested proof obligations, and hence the running time.

6 Related Work and Conclusions

Formal verification of parameterized systems has been researched widely in the last decade. Some of the well studied techniques include network invariants [7, 19, 20, 28] (where a finite state process invariant is synthesized), and use of general purpose theorem provers *e.g.* PVS [22], ACL2 [17], Coq [14]. In the recent past, a lot of activity has been directed towards developing automated techniques for verifying (classes of) parameterized systems. These include identification of classes for which parameterized system verification is decidable [9, 10, 13, 15], and application of model checking over rich languages [8, 12, 16, 18].

The rich language model checking approach finitely represents the state space and transition relation of a parameterized family via rich languages *e.g.* regular, tree-regular languages for linear, tree networks. Note that our approach achieves a *different* finite representation; we finitely represent infinite sets of states as recursively defined logic program predicates. In comparison to the rich language approach, our technique is not tied to specific classes of networks based on the choice of the rich language. Thus we have verified parameterized networks of various topologies *e.g.* chain, ring, tree, complete graph, star networks. Moreover, the rich language approach constructs proofs by state space traversal (uniform proofs) whereas our proofs are inductive.

Our prover is a lightweight automated inductive theorem prover for constructing nested induction proofs. Note that in our approach, the induction schema as well as the lemmas to be used in the inductive proof must be implicit in the logic program itself. This is a *limitation* of our method. Besides, our proof technique does not support strengthening of induction hypothesis in an inductive proof. However, if the schema and the lemmas are implicit in the logic program, our syntax based transformations uncover the induction schema and reason about its different cases by uncovering the requisite lemmas.

As future work, we plan to integrate automated invariant strengthening techniques [5] into our proof technique. This would involve developing a proof methodology containing both program analysis (to strengthen invariants) and program transformation (to inductively prove the invariants).

Acknowledgments The authors would like to thank Samik Basu for his help in modeling the Java meta-locking algorithm.

References

1. O. Agesen et al. An efficient meta-lock for implementing ubiquitous synchronization. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999. Technical report available from `http://www.sun.com/research/techrep/1999/abstract-76.html`.
2. K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *Information Processing Letters*, 15:307–309, 1986.
3. J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4, 1986.
4. S. Basu, S.A. Smolka, and O.R. Ward. Model checking the Java meta-locking algorithm. In *IEEE International Conference on the Engineering of Computer Based Systems*. IEEE Press, April 2000.
5. N. Bjorner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
7. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5), 1997.
8. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification (CAV), LNCS 1855*, 2000.
9. E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, 1995.

10. E.A. Emerson and K.S. Namjoshi. Automated verification of parameterized synchronous systems. In *Computer Aided Verification (CAV), LNCS 1102*, 1996.
11. E.A. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite state systems. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1998.
12. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1999.
13. S. German and A. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.
14. INRIA Rocquencourt, URL `http://pauillac.inria.fr/coq/doc/main.html`, Paris, France. *The Coq Proof Assistant : Reference Manual*, 1999.
15. C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. *Formal Methods in System Design*, 14(3), May 1999.
16. B. Jonsson and M. Nilsson. 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*, 2000.
17. M. Kaufmann, P. Manolis, and J.S. Moore. *Computer-Aided Reasoning: An approach*. Kluwer Academic, 2000.
18. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Computer Aided Verification (CAV), LNCS 1254*, 1997.
19. R.P. Kurshan and K. Mcmillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.
20. D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, pages 346–357, 1997.
21. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, 1985.
22. S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. In *International Conference on Automated Deduction (CADE)*, 1992.
23. F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transacations on Parallel and Distributed Systems*, 6(8), 1995.
24. J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *International Symposium on Programming, LNCS 137*, 1982.
25. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I.V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1785*, pages 172–187, 2000.
26. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. In *International Conference on Principles and Practice of Declarative Programming (PPDP), LNCS 1702*, pages 396–413, 1999.
27. Abhik Roychoudhury. *Program Transformations for Verifying Parameterized Systems*. PhD thesis, State University of New York at Stony Brook, Available from `http://www.cs.sunysb.edu/~abhik/papers`, 2000.
28. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *LNCS 407*, 1989.
29. XSB. The XSB logic programming system v2.2, 2000. Available for downloading from `http://xsb.sourceforge.net/`.