Program Transformations for Verifying Parameterized Concurrent Systems

Abhik Roychoudhury

Department of Computer Science

State University of New York at Stony Brook

Parameterized Systems

- Infinite state concurrent system parameterized by the number of component processes *e.g.* n process token ring
- Occurs widely in distributed algorithms *e.g.* an algorithm to ensure mutual exclusion of a shared resource among n users.
- Verification of temporal properties of parameterized systems is undecidable. [AK86]

Model Checking

- Given a finite state concurrent system s and a temporal property f about the execution runs of s, determine automatically:

$$\mathtt{s} \overset{?}{\models} \mathtt{f}$$

- Traverses a finite graph (the state space of s) to determine truth of f in the execution runs of s.

- Model checking can only verify properties of finite instances of parameterized systems.

Can we enhance the underlying evaluation of a model checker to verify parameterized systems ?

Overall Perspective

- Verification of finite & inf. state systems can be directly encoded as a (constraint) logic program.
- Finite state model checking via memoized query evaluation of LP encoding. [RRR$^+$97]
- Verification of real-time/hybrid systems via query evaluation of CLP encoding. [DRS00, U96]
- Verification of parameterized systems via program transformation of LP encoding.

Contributions

- Extend resolution based query evaluation to a program transformation framework for constructing inductive proofs of parameterized systems.
- Develop new, more general transformation rules for constructing induction proofs of temporal properties.
- An automated lightweight prover for parameterized protocols. Tightly integrates model checking and deductive verification.

Logic Programming

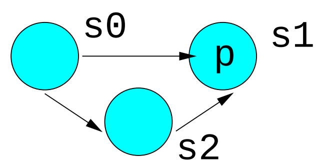
- Directly executable specifications. Natural to specify problems with:
  - *Backtracking Search*
  - *Fixed point computations*
- Applications : *NLP, Program Analysis, Model Checking*
- A (definite) logic program is a finite set of Horn clauses.

```
app([], L, L).
app([H|X], Y, [H|Z]) :- app(X, Y, Z).
```

- Computation proceeds by query evaluation via resolution.

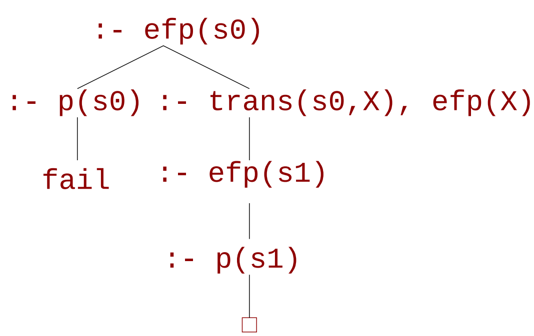
```
app([1,3], [2], X)   app(X, [1], [0,1])
X = [1,3,2]          X = [0]
```

Model Checking by Query Evaluation



$$\mathtt{s0} \stackrel{?}{\models} \mathtt{EFp}$$

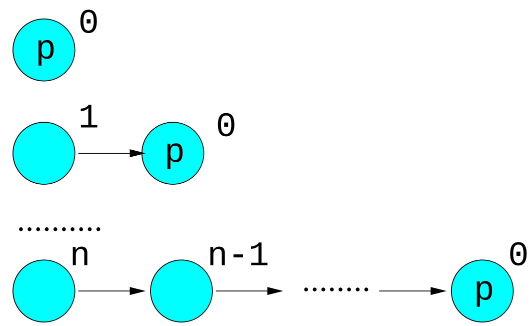
- Predicate $\mathtt{efp}$ encodes property $\mathtt{EFp}$ *i.e.* $\mathtt{efp(S)} \equiv \mathtt{S} \models \mathtt{EFp}$
- Evaluate the query $\mathtt{efp(s0)}$ to verify $\mathtt{s0} \models \mathtt{EFp}$




```
efp(S) :- p(S).
efp(S) :- trans(S,T), efp(T).
trans(s0, s1).
trans(s0, s2).
trans(s2, s1).
```

On-the-fly explicit state model checking via query evaluation.

Query Evaluation is not enough



To check whether **EFp** is true in the start state of each member of the family we need to deduce $\forall n\ \mathtt{nat(n)} \Rightarrow \mathtt{efp(n)}$ where:

```
efp(S) :- p(S).
efp(S) :- trans(S, T), efp(T).
```

Simple tricks like querying `:- efp(N)` do not work as the set of answers is infinite and query evaluation enumerates all of them.

A Human Proof

Theorem: $\forall$ `n` `nat(n)` $\Rightarrow$ `efp(n)`

Proof: By induction on `nat`.

- Base case: `n = 0` : Then `efp(0)` is true since `p(0)` is true *i.e.* proposition `p` holds in state 0.
- Induction hypothesis: $\forall$ `n`$\leq$ `m` `nat(n)` $\Rightarrow$ `efp(n)`
- Induction step: `n = m + 1` : `efp(m + 1)` is true because `efp(m)` is true (by induction hypothesis) and there exists transition `m + 1` $\longrightarrow$ `m` *i.e.* `trans(m + 1, m)` is true. $\square$

Pgm. txf. combine query eval. and ind. reasoning to automate this proof.

Proof by Program Transformations

1. Define in program P_0 `thm(N) :- nat(N), efp(N).`

2. Convert the proof obligation to a predicate equivalence in P_0. It is sufficient to prove $\forall$`n` `nat(n)` $\wedge$ `efp(n)` $\Leftrightarrow$ `nat(n)` *i.e.* `thm` $\equiv$ `nat`.

3. Transform def. `thm` in pgm. P_0 to the following def. in pgm. P_k

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

   Recall that the set of natural numbers is defined by `nat`:

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

   The def. of `thm` in pgm. P_k is *isomorphic* to the def. of `nat`.

Step (3) uncovers the induction schema and constructs the proof.

Verification by Transformations - Method

1. Encode the temporal property to be verified as well as the parameterized system as a logic program P_0.

2. Convert the verification proof obligation to a predicate equivalence proof obligation $P_0 \vdash \mathtt{p} \equiv \mathtt{q}$ ($\mathtt{p}, \mathtt{q}$ are predicates)

   e.g. we need to prove $\mathtt{thm} \equiv \mathtt{nat}$

3. Construct an unfold/fold 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 $P_k \vdash \mathtt{p} \equiv \mathtt{q}$

   e.g. since the transformed definitions of `nat` and `thm` are isomorphic, we can infer their equivalence from syntax.

[LOPSTR '99]

Program Transformation : Unfolding

$$
\begin{array}{l}
\mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'. \\[1em]
\mathtt{p} :- Bd_1. \\
\mathtt{p} :- Bd_2. \\
\vdots \\
\mathtt{p} :- Bd_n.
\end{array}
\quad \xrightarrow{Unfold} \quad
\begin{array}{l}
\mathtt{q} :- \mathtt{g}, \boxed{\begin{array}{c} Bd_1, \\ Bd_2, \\ \\ \\ Bd_n, \end{array}} \mathtt{g}'. \\
\vdots \\
\mathtt{p} :- Bd_1. \\
\vdots \\
\mathtt{p} :- Bd_n.
\end{array}
$$

- Each step in query evaluation is an application of unfolding.
- Corresponds to the base case as well as finitely evaluated portions of the induction step in an induction proof.

Program Transformation : Folding

$$\dfrac{P_j\ (j \leq i):\quad \begin{array}{l} \mathtt{p} :- Bd_1. \\ \vdots \\ \mathtt{p} :- Bd_n. \end{array}}{P_i:\quad \begin{array}{lll} \mathtt{q} :- \mathtt{g}, & Bd_1, & \mathtt{g}'. \\ \mathtt{q} :- \mathtt{g}, & Bd_2, & \mathtt{g}'. \\ \vdots & & \\ \mathtt{q} :- \mathtt{g}, & Bd_n, & \mathtt{g}'. \end{array} \quad \xrightarrow{\textit{Fold}} \quad \mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'.}$$

- Replaces occurrence of clause body (from a previous program in the transformation sequence) by its head.

Verification by Transformations - Example

P_0 :

```
thm(N) :- nat(N), efp(N).

nat(0).

nat(s(N)) :- nat(N).

efp(S) :- p(S).

efp(S) :- trans(S,T), efp(T).

p(0).      trans(s(N), N).
```

$\Big\downarrow$ $Unfold$ (Discovering induction schema)

P_1 :

```
thm(0) :- efp(0).

thm(s(N)) :- nat(N), efp(s(N)).
```

Verification by Transformations - Example

P_1 :

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

$\downarrow$ *Unfolds* (Base case completed)

P_4 :

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

$\downarrow$ *Unfolds* (Finite part of induction step)

P_7 :

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

$\downarrow$ *Fold* (Applying induction hypothesis)

P_8 :

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

Folding and Inductive Reasoning

Folding corresponds to recognizing induction hypothesis in an induction proof of a predicate equivalence.

P_0:

$$\mathtt{thm(N)} :- \mathtt{nat(N)}, \mathtt{efp(N)}.$$
$$\mathtt{nat(0)}.$$
$$\mathtt{nat(s(N))} :- \mathtt{nat(N)}.$$

P_i:

$$\begin{array}{lcl}
\mathtt{thm(0).} & & \mathtt{thm(0).} \\
\mathtt{thm(s(N))} :- \boxed{\mathtt{nat(N),\ efp(N).}} & \xrightarrow{\textit{Fold}} & \mathtt{thm(s(N))} :- \mathtt{thm(N).} \\
\mathtt{nat(0).} & & \mathtt{nat(0).} \\
\mathtt{nat(s(N))} :- \mathtt{nat(N).} & & \mathtt{nat(s(N))} :- \mathtt{nat(N).}
\end{array}$$

Need to develop new transformation rules

- Inductive proofs of a pred. equiv proof obligation $P_0 \vdash \mathtt{p} \equiv \mathtt{q}$ are constructed via a unf./fold txf. seq $P_0, P_1, \ldots, P_k$ s.t.
  - Semantics of P_0 = Semantics of P_k
  - from the syntax of P_k automatically infer $P_k \vdash \mathtt{p} \equiv \mathtt{q}$
- For inductive proofs of temporal properties, the corresp. txf. seq. might involve folds using the definition of the temporal property.
- Temporal properties involve fixed point operators and are defined using *multiple recursive clauses* e.g. `efp`.
- Current transformation systems do not allow such folding steps.

Transformations must preserve semantics!

- To prove temporal property EFp about an infinite family st
  - encode property EFp as a pred. $\mathtt{efp(S)} \equiv \mathtt{S} \models \mathtt{EFp}$
  - induction step is of the form $\mathtt{efp(st(N))} \Rightarrow \mathtt{efp(st(N+1))}$
- Recognizing the induction hypothesis $\mathtt{efp(st(N))}$ may involve multiple folds *e.g.* folding using the definition of property efp.

```
efp(S) :- p(S).
efp(S) :- trans(S, T), efp(T).
```

Such a folding is semantics preserving ?

Semantics of Logic Programs

$$\begin{aligned} M(P) &= \text{Least Herbrand Model of } P \\ &= \text{Gnd. atoms which are logical consequences of } P \\ &= \text{Gnd. atoms which are proved by resolution with clauses of } P \end{aligned}$$

$P =$

```
p(a).
p(f(X)) :- p(X).
```

$P' =$

```
p(f(X)) :- p(X).
```

$M(P) = \{\texttt{p(a)}, \texttt{p(f(a))}, \texttt{p(f(f(a)))}, \ldots\}$

$M(P') = \{\}$

Proofs in P:

```
p(a)    p(f(a))    p(f(f(a))) ...
 |         |            |
 □       p(a)        p(f(a))
           |            |
           □          p(a)
                        |
                        □
```

Proofs in P': None

Theorem $A \in M(P) \Leftrightarrow$ A appears in the root of a proof.

A Problem with Folding

- All unfold/fold transformation seq. are **not** semantics preserving.

```
p :- q.    --Fold-->    p :- p.
q.                      q.
```

- Folding may *introduce* circularity in predicate definitions, thereby removing finite proof paths.

- *State of the Art*: [TS84,TS86,KF87,GK94,PP99]
  To preserve semantics, existing transformation systems conservatively impose syntactic restrictions on the clauses participating in folding, *e.g.* only a single clause and/or only non-recursive clauses

Proving Correctness of Transformations

- *Need to prove:* $M(P_i) = M(P_{i+1})$
- $\forall A \in M(P_i)$ $\exists$ proof of A in P_{i+1} (& vice-versa).
- By induction on some measure α of ground atoms.

$P_j (j \leq i)$: $\quad \mathtt{p} :- Bd_1.$

P_i: $\quad \mathtt{q} :- \mathtt{g}, \boxed{Bd_1,}\ \mathtt{g}'. \quad \xrightarrow{\textit{Fold}} \quad P_{i+1}$: $\quad \mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'.$

- This can only be shown if $\alpha(\mathtt{p}) \prec \alpha(\mathtt{q})$ (by induction hypothesis).
- Syntactic restrictions on clauses of $\mathtt{p}$ ensure this inequality.

Our transformation framework

- Attach abstract measures to program clauses and atoms:
  - Update *clause measures* in each transformation step.
  - Impose (automatically checkable) conditions on *clause measures* to disallow incorrect folding steps.
  - Induct on an ordering of *atom measures* to prove preservation of program semantics.
- The only restrictions in a folding step are on the clause measures and not on the syntax of clauses.
- Previously existing unfold/fold transformation systems can be derived by simply instantiating the measures.

[PPDP '99]

Clause Measures

For any transformation sequence $P_0, P_1, \ldots, P_n$ annotate each clause $C \in P_i$ with a pair $(\gamma^i_{lo}(C), \gamma^i_{hi}(C))$ where:

- $\gamma^i_{lo}(C), \gamma^i_{hi}(C)$ denote the minimum and maximum savings (in proof size) incurred by applying C in a ground proof in P_i.
- a folding step is allowed when the previously accumulated savings are not offset by this folding step.
- In general, the clause measures can come from a commutative group $\langle \mathcal{M}, \oplus \rangle$ where $\oplus$ is monotonic w.r.t. $\succ$.
- Appropriate instantiation of this algebraic structure yields previously existing works, *e.g.* [TS84, TS86, KF87, GK94, PP99].

Extending to normal logic programs

- This transformation framework is extended to produce more general unfold/fold transformation system for normal logic programs.
- Uniform proof of correctness of transformation system for different semantics of negation *e.g* well-founded and stable model semantics.
- This uniform correctness proof involves
  - identifying model-theoretic notion of semantic kernel [DK89] which captures the commonality of the different semantics.
  - developing a proof-theoretic equivalent of semantic kernel, and showing that it is preserved by the transformations.

[ASIAN '99]

Implications of our framework

- Application of transformations are restricted only by mechanically checkable conditions on clause measures, and not by pgm. syntax.
- Allows us to identify an abstract program property (Measure Consistency) which ensures correctness of unfold/fold transformations. Can introduce new transformations without redoing correctness proofs.
- Existing Txf. Systems are specific instances of our framework $\Rightarrow$ allows us to understand strengths and limitations of existing systems.
- Construction of new unfold/fold txf. systems which is more powerful (in terms of transformation sequences allowed).

Verification by Transformations - Recap

1. Encode the temporal property to be verified as well as the parameterized system as a logic program P_0.
2. Convert the verification proof obligation to a predicate equivalence proof obligation $P_0 \vdash \mathtt{p} \equiv \mathtt{q}$ ($\mathtt{p}, \mathtt{q}$ are predicates)
3. Construct an unfold/fold transformation sequence $P_0, P_1, \ldots, P_k$
   (a) Semantics of P_0 = Semantics of P_k
   (b) from the syntax of P_k we infer $P_k \vdash \mathtt{p} \equiv \mathtt{q}$

Control Strategies for Automation

- Control strategies to guide application of transformations are needed to automate the construction of a verification proof.
- We have developed (a family of) terminating strategies which:
  - Apply model checking (unfolding) steps "as much as possible".
  - Allow interleaving of model checking and deductive steps.
- Model checking via unfolding is an instance of our strategies.
- Process structure irrelevant to the infinite part does not affect control strategy.

[TACAS '00]

Nesting of proofs

- Our semantics preserving unfold/fold transformations can simulate induction proofs.
- Human proofs are *hierarchical* ; an induction proof may use a lemma which is again proved by induction.
- Nested induction proofs simulated by nesting pred. equiv. obligations.
- An unfold/fold proof of an equivalence $\mathtt{p} \equiv \mathtt{q}$ uses pred. equivalences $\mathtt{p}' \equiv \mathtt{q}'$ and $\mathtt{p}'' \equiv \mathtt{q}''$ as lemma. These equivalences are again proved by unfold/fold.

A Liveness Example

System:

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

Property:

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

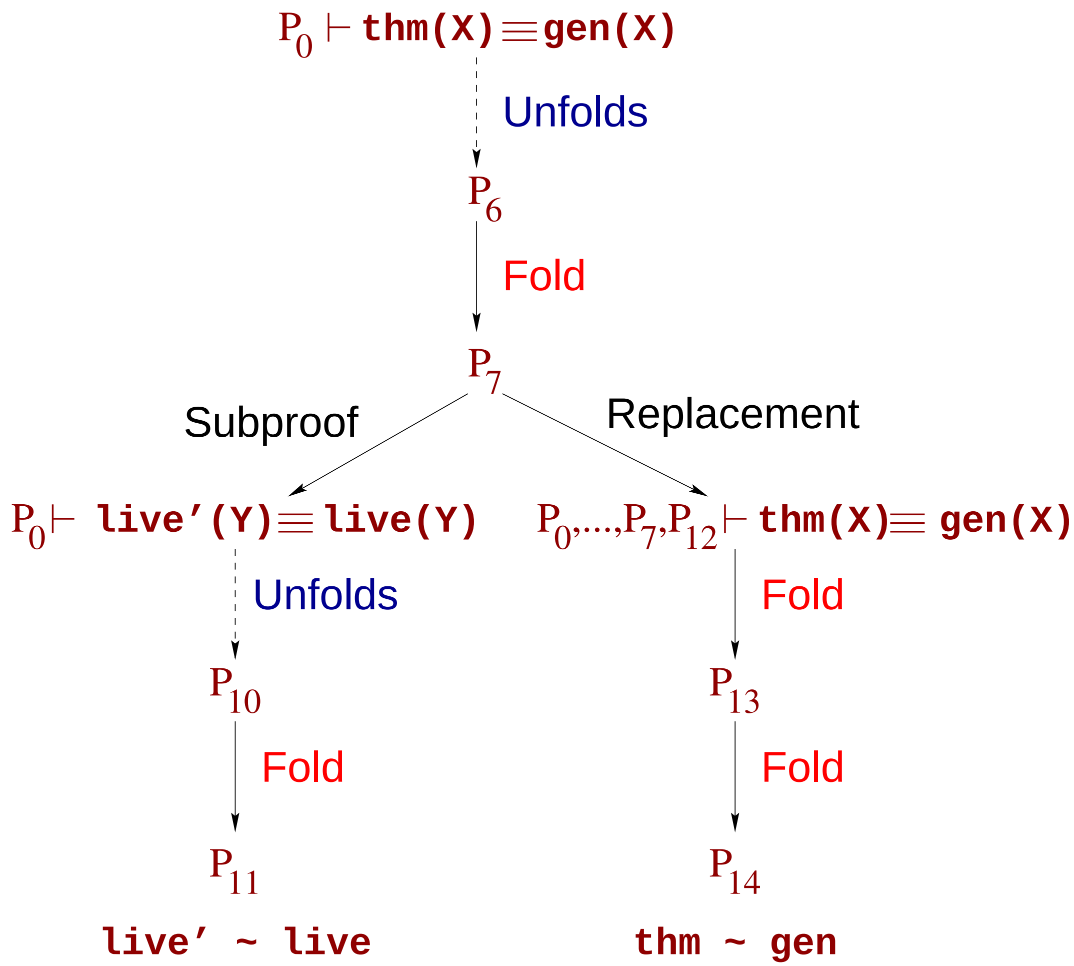
Proof obligation : $\forall$ `X gen(X)` $\Rightarrow$ `live(X)`

Sufficient to prove: $\forall$ `X gen(X)` $\wedge$ `live(X)` $\Leftrightarrow$ `gen(X)`

`thm` $\equiv$ `gen` where:

```
thm(X) :- gen(X), live(X)
```

A Liveness Proof



Design of a Lightweight Prover

- Geared to automate *nested* induction proofs where each induction proceeds without *strengthening* of hypothesis.
- A proof attempt is guaranteed to *terminate*, but may not be *successful*.
- The prover is both automated and programmable. User can set problem-specific information before the proof attempt begins.
- Typical user settings to control proof search: Network topology of the parameterized system, Predicates encoding temporal property
- User settings are exploited by the prover to *resolve nondeterminism* when multiple transformation rules are applicable.

Implementation

- Built an automated prover based on program transformations on top of XSB tabled LP system.
- **Process Structure:** Used to produce liveness and safety proofs for parameterized networks of different topologies: uni- and bi-directional chains, rings, trees.
- **Predicate Equivalences:** Given the temporal property and the parameterized network, finding the predicates equivalences to be proved requires no user ingenuity.
- **Case Studies :**
  Cache Coherence Protocols
  Java Meta-Locking Algorithm (Sun MicroSystems)

Case Study: Java Meta-Locking Algorithm

- In Java program execution, any object can be synchronized upon by different threads.
- To maintain mutually exclusive access to an obj., each obj. maintains synchronization data (queue of threads waiting to access obj).
- The meta-locking algorithm is a distributed algorithm observed by each thread and any object in accessing the synch. data.
- Need to prove mutual exclusion in access of synch. data.
- Previous proof attempts include an informal proof sketch [ADG$^+$99] and model checking of instances of the algorithm. [BSW00]

Modeling the algorithm

- Parameterized System : $\mathit{Handoff} \parallel \mathit{Object} \parallel \mathit{Thread} \parallel \ldots \parallel \mathit{Thread}$
- Thread : *Finite* state sys. which acquires/releases meta-lock via fast/slow path.
- Handoff : *Finite* state sys. which resolves contention between acquiring and releasing threads. Eliminates busy waiting.
- Object : *Inf.* state sys. Records # of threads waiting for the meta-lock.
- Inf. state sys. with infiniteness in both data & control.

Verifying Mutual Exclusion

1. Represent a global state as an unbounded term `(Th, Obj, H)`. Define transition relation as predicate over these terms.
2. Must prove $\mathbf{AG}\ \neg$`bad` where `bad` allows ≥ 2 threads to own meta-lock.
3. Encode `bad` as a predicate over global states which considers the semantics of the local states of object and handoff as well.
4. Prove pred. equiv. corresp. to invariant automatically.

Running time:	5 mins, 23 secs
# Unfolding:	1981
# Deductive:	311
# Pred. equiv.:	39

Related Work

- Developing Decision Procedures for Subclasses: [EN95, ID99] : Typically, establishing a "*cutoff number*", and checking that the property holds for members up to the cutoff number.
- Network Invariants [KM95,CGJ95,LHR97] : For an infinite concurrent composition (of finite state processes) *synthesize* I s.t. it is sufficient to verify I.
- Rich language Model Checking [KMM97,EFM99] : Represent the state-space of a parameterized system finitely by an underlying rich language; perform (symbolic) model checking over the rich language.
- Theorem Provers [PVS,ACL2,Oyster-CLAM,RRL] : Construct verification proofs through substantial human assistance.

Features of our Proof Technique

- We enhance the underlying evaluation technique (unfolding) of a model checker with deductive capabilities (*e.g.* folding)
- Our program-transformation based technique allows arbitrary interleaving of unfolding (model checking) and folding (deduction) steps in a verification proof.

  $\Rightarrow$ *Tighter integration of model checking and theorem proving.*
- Since deductive steps are applied lazily:
  - deductive steps do not affect efficiency of model checking steps
  - for finite state systems our technique reduces to model checking.

  $\Rightarrow$ *Supports "zero overhead theorem proving".*

Future Work

- Transformations of Constraint Logic Programs
- Verifying Parameterized Real-time Systems
- Explaining Txf. Proofs/ Developing Txf. Proof Checker[a]
- Integrating automated assertion strengthening techniques to the proof technique.
- Identifying subclasses for which specific strategies of unfold/fold proof technique serves as a decision procedure.

[a] Initial work in this direction [PPDP '00]

EXTRA: Relating Atom and Clause Measures

- Roughly, for a clause $C = (A\text{:-}\ Bd)$ in program P_i appearing in a (canonical) proof, the clause measures of C bound the difference in atom measures of head and body.

$$\gamma_{lo}^{i}(C) \ \preceq \alpha(A) \ominus \sum_{B \in Bd} \alpha(B) \ \preceq \gamma_{hi}^{i}(C)$$

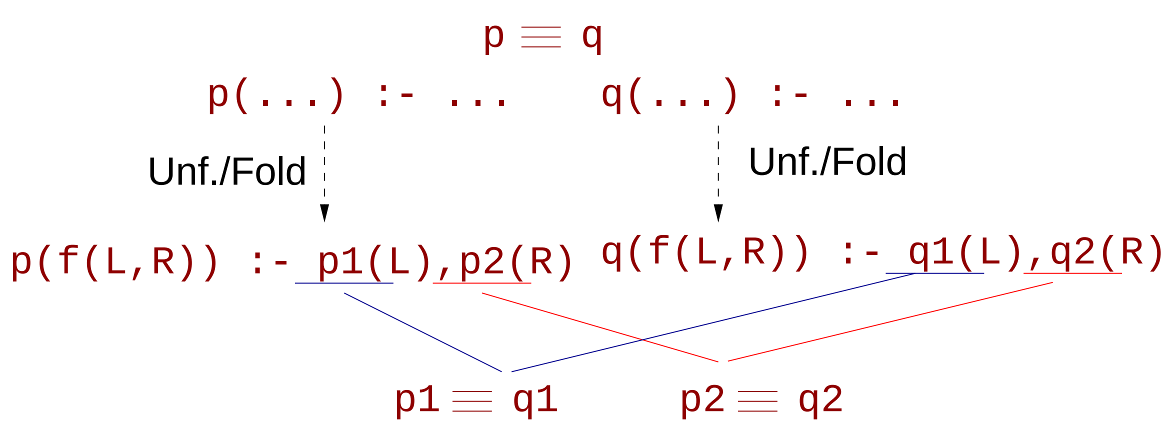
- By imposing checks on the clause measures of C and D we ensure $\alpha(\mathtt{p}) \prec \alpha(\mathtt{q})$.

$P_j\,(j \leq i)$: $\quad D = (\mathtt{p} : -Bd_1.)$

P_i: $\quad C = (\mathtt{q} : -\mathtt{g}, \boxed{Bd_1,}\ \mathtt{g}'.) \quad \xrightarrow{Fold} \quad P_{i+1}$: $\quad \cdots$

EXTRA: Exploiting User Settings to Control Proof Search

- Network topology is implicit in the recursive structure of predicates describing the parameterized network.
- Unf./Fold of these predicate occurrences exposes this structure.
- Knowledge of the network topology then exploited by the prover to choose nested proof obligations to be dispensed.



EXTRA: Comparison to Related Work

- Network Invariants: Uses a specific induction principle.

  $$P \models \varphi \qquad I \models \varphi \Rightarrow I \times P \models \varphi$$

  This induction proof can be constructed using our transformations.

- Rich language model checking:
  - Rich language assertions capture an exact finite description of the state space. In our work, this is captured by recursive predicates.
  - Constructs uniform proofs. We construct induction proofs.

- Theorem Proving: Integrate model-checking (and other automated techniques) as decision procedures to the theorem prover.

  We enhance the underlying evaluation technique (unfolding) of a model checker (XMC) with deductive capabilities (*e.g.* folding).