Verification Using Tabled Logic Programming

C. R. Ramakrishnan

SUNY, Stony Brook

Organization

- ***Verification by query evaluation***
- An overview of Tabled Logic Programming
- Representing transition systems
- Model checking modal mu-calculus
- Infinite-state systems and Constraint LP
- Induction proofs via program transformation
- Symbolic bisimulation for value-passing systems
- Justification of verification proofs

Verification by Query Evaluation

Model Checking: Given a system description with start state s_0 and a property φ

$$s_0 \overset{?}{\models} \varphi$$

Query Evaluation: Encode the "$\models$" relation as predicate `models` in a logic program.

$s_0 \overset{?}{\models} \varphi$ is determined by solution to the query $\texttt{models}(s_0, \varphi)$

Logic Programming-Based Model Checking

LMC Project: [SUNY, Stony Brook]

Explore the application of Tabled Logic Programming for Model Checking.

- Semantic equations of process calculi and temporal logics can be directly encoded as Horn Clauses and evaluated by tabled resolution.
- Constraint processing and Tabling can be combined to compute fixed points over infinite domains: for verifying properties of infinite-state systems.
- Certain deduction (theorem proving) strategies can be encoded as logic rules: can be used to verify systems by a *combination* of model checking and theorem proving.

The XMC System

- Semantics of temporal logics are encoded as a logic program.
- Transition systems are described by rules expressed in Horn logic *(derived from specifications in a process algebra)*.
- Model-checking queries are evaluated using *tabled resolution*.
- Proofs/counter-examples are derived from lemmas stored by the resolution strategy.

Sources can be obtained from

`http://www.cs.sunysb.edu/~lmc`

Model Checking using LP: Other Work

- Genova: G. Delzanno (originally with A. Podelski)
- MPI: A. Podelski & S. Mukhopadhyay
- Linköping: U. Nilsson & J. Lübke
- UT Dallas/ NMSU: G. Gupta & E. Pontelli
- Southampton: M. Leuschel
- . . .

A Simple Example

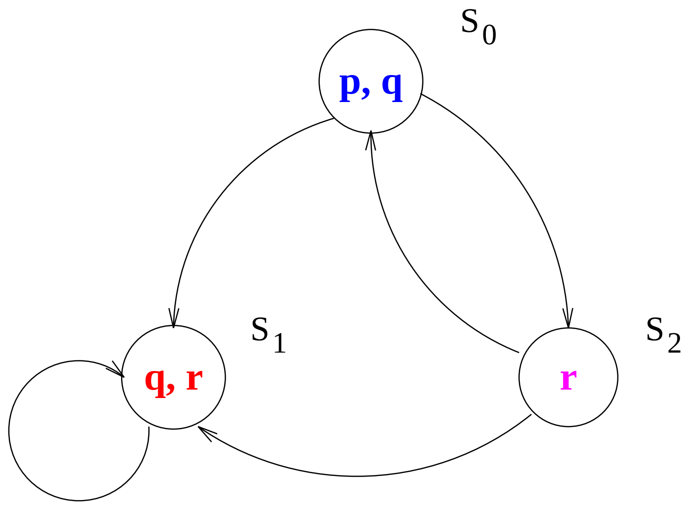
Verifying Reachability Properties

- Encode Kripke Structure using “EDB facts”
- Encode reachability relation using Horn Clauses
- Issue appropriate query

A Simple Example – II

Encoding Kripke Structures using EDB facts

Structure



Encoding

```
edge(s0, s1).
edge(s0, s2).
edge(s1, s1).
edge(s2, s0).
edge(s2, s1).

prop(s0, p).
prop(s0, q).
prop(s1, q).
prop(s1, r).
prop(s2, r).
```

A Simple Example — III

Reachability relation:

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Query: e.g., "Is a state where 'r' is true reachable from state s_0?

```
?- reach(s0, S), prop(S, r).
```

Answers: `S=s1`, `S=s2`

Query Evaluation Techniques

SLD Resolution: Goal directed, complete.

"Oracle" for selecting literal to be resolved.

OLD resolution: Goal directed, fixed literal selection order, incomplete.

Implemented by Prolog engines.

Bottom-up evaluation: Complete for Datalog; Set-at-a-time.

Global evaluation.

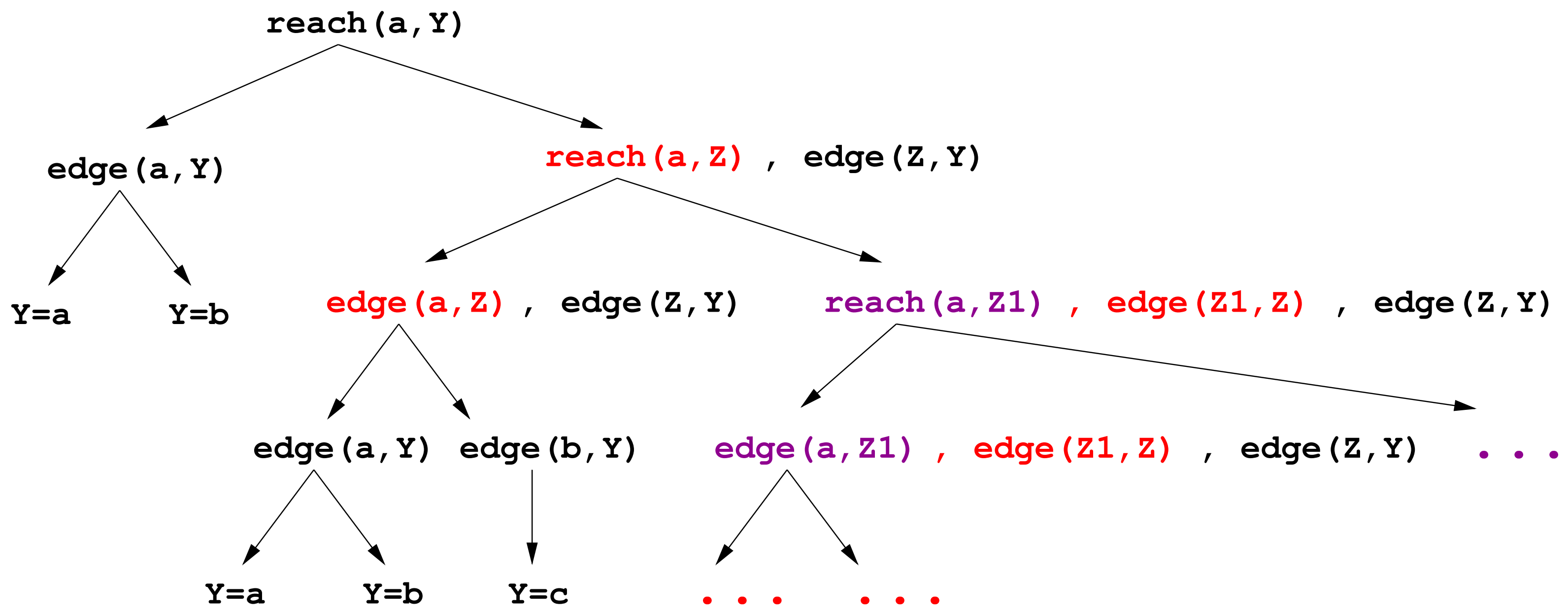
Magic-Sets: Add goal direction to "bottom-up" evaluation.

OLDT: OLD resolution with tabling.

Complete for Datalog; Goal-directed.

Prolog Evaluation: An Example

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



Organization

- Verification by query evaluation
- ***An overview of Tabled Logic Programming***
- Representing transition systems
- Model checking modal mu-calculus
- Infinite-state systems and Constraint LP
- Induction proofs via program transformation
- Symbolic bisimulation for value-passing systems
- Justification of verification proofs

What is Tabled Resolution?

Memoize results to avoid repeated subcomputations.

- *Termination:* Avoid performing subcomputations that repeat infinitely often.
  - Complete for datalog programs
- *Efficiency:* dynamically share common subexpressions.

Power: Effectively computes fixed points of Horn clauses viewed as set equations.

Tabled Resolution

Record goals in *call table*
and their provable instances in *answer table*.

On encountering a goal G,

- If G is present in call table:
  - Resolve G with the associated *answers*.
- If G is not present in call table:
  - Enter G in call table
  - Resolve G with *program* clauses to generate answers
  - Enter each answer in the associated answer table.

Evaluation using Tabled Resolution

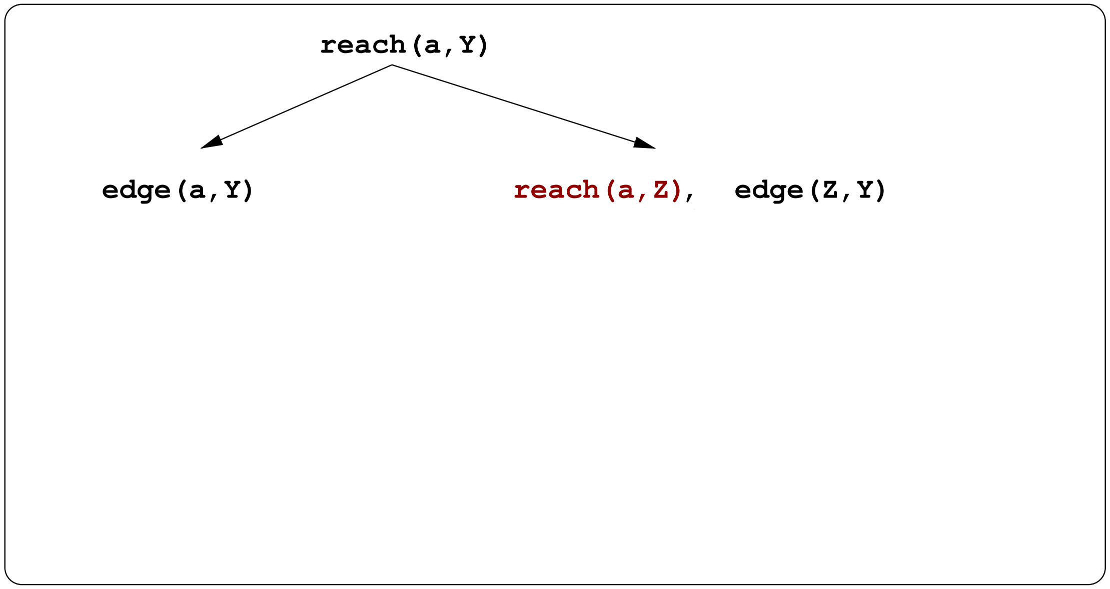
```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```

Calls

```
reach(a,V)
```

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```

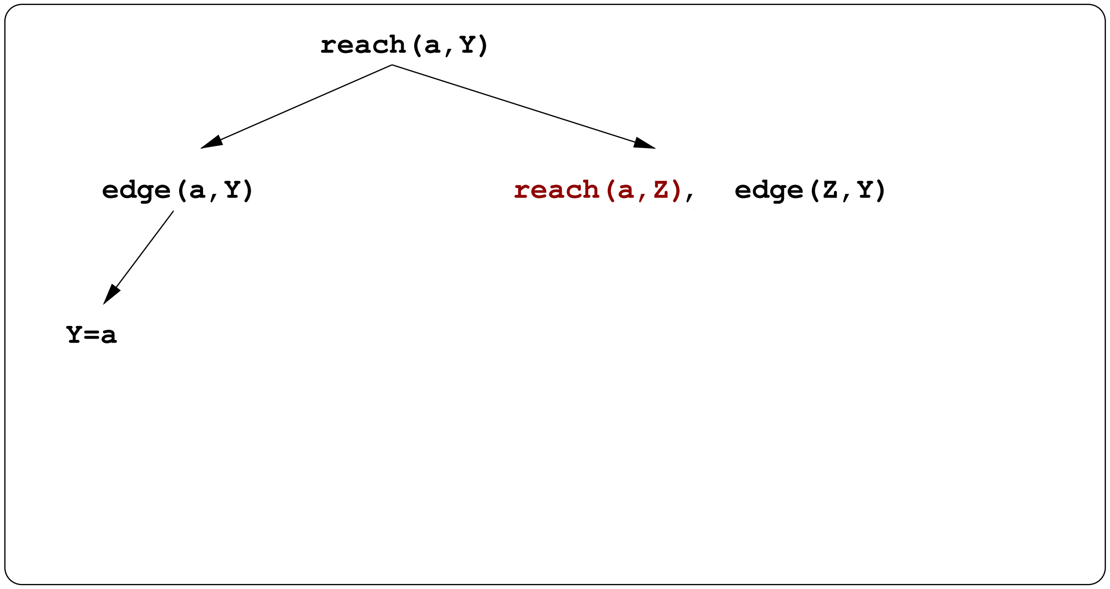


Calls

`reach(a,V)` **Answers**

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



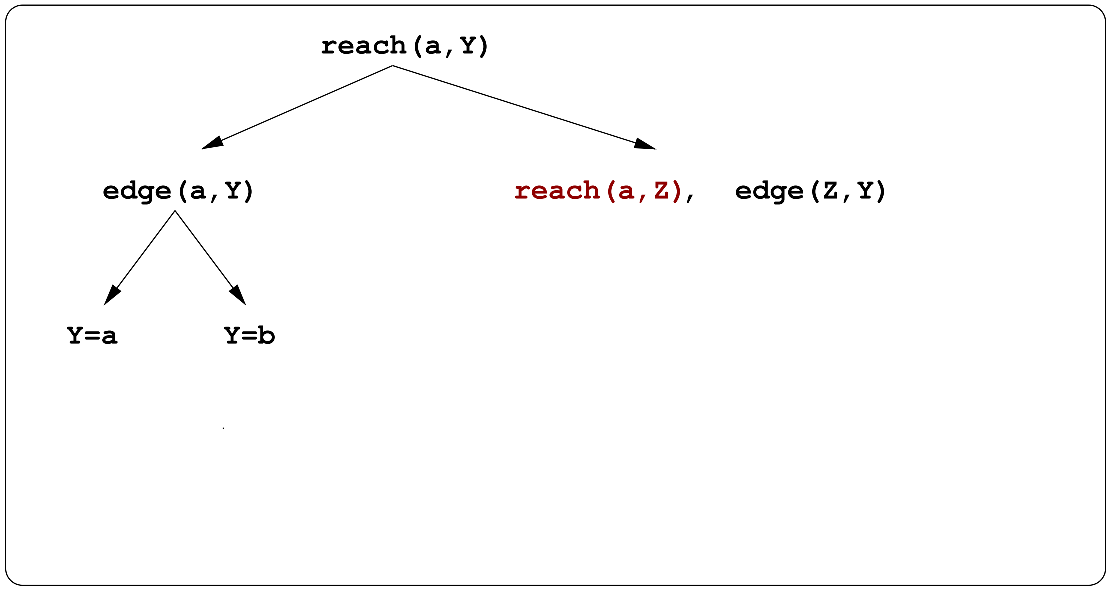
Calls

`reach(a,V)` **Answers**

`V=a`

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



Calls

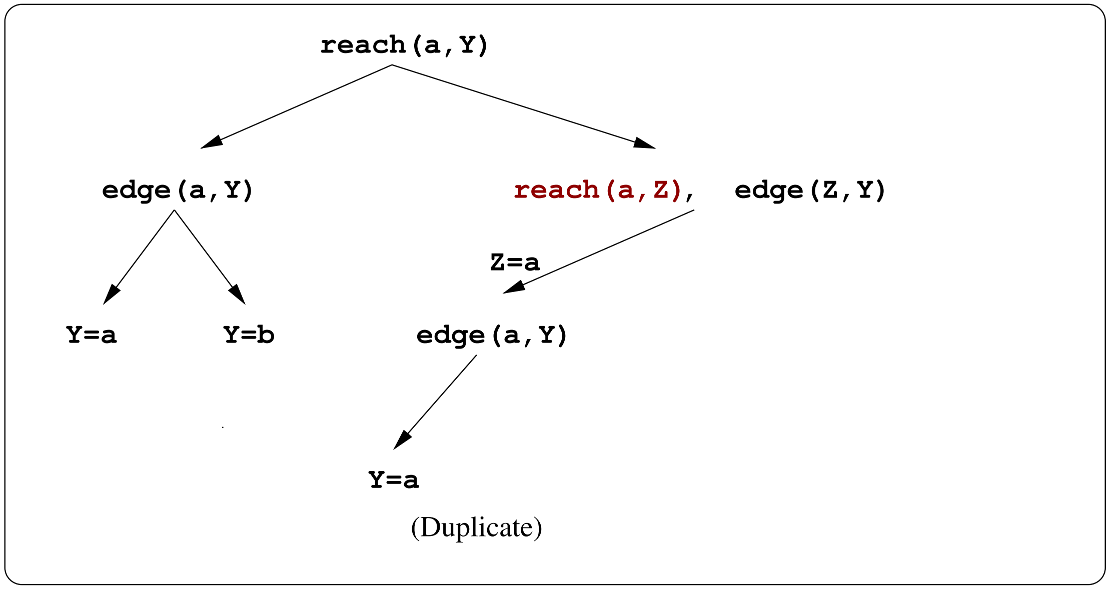
`reach(a,V)` **Answers**

`V=a`

`V=b`

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



Calls

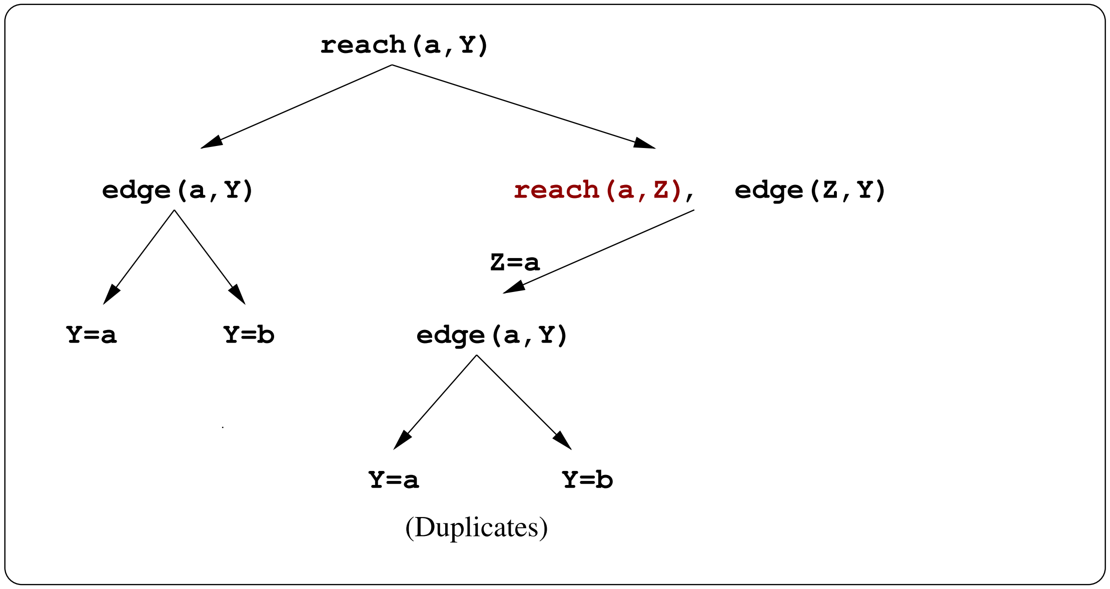
`reach(a,V)` **Answers**

`V=a`

`V=b`

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



Calls

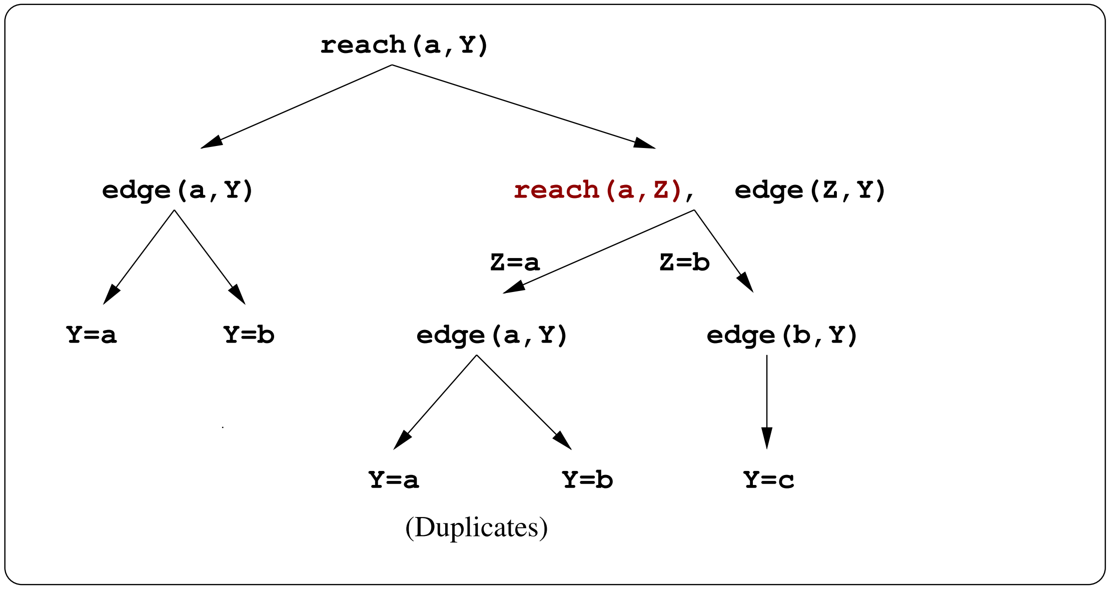
`reach(a,V)` **Answers**

`V=a`

`V=b`

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



Calls

`reach(a,V)` **Answers**

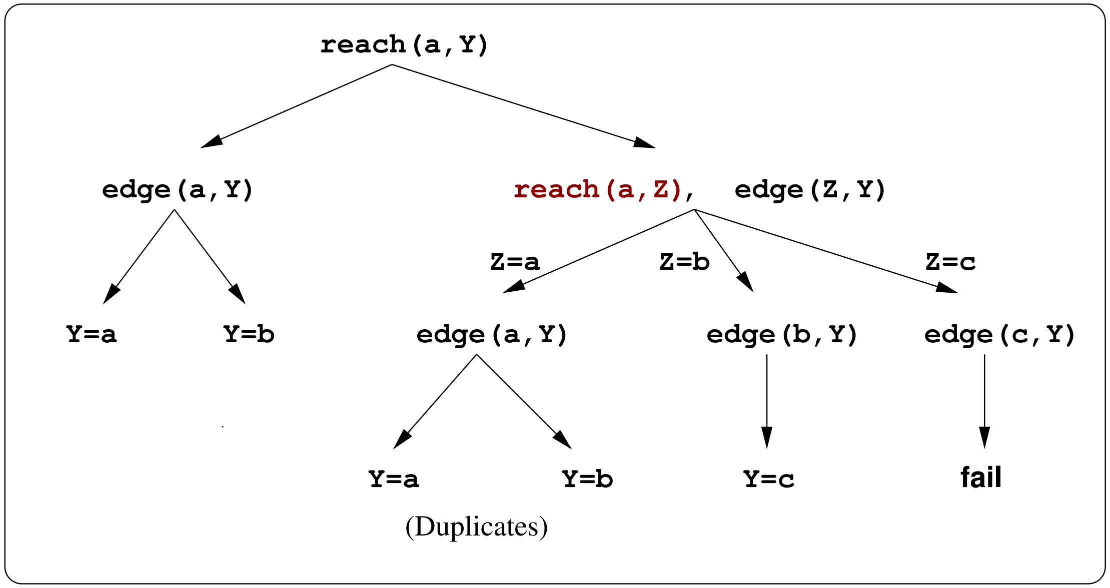
`V=a`

`V=b`

`V=c`

Evaluation using Tabled Resolution

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
edge(a,a).
edge(a,b).
edge(b,c).
```



Calls

reach(a,V) **Answers**

V=a

V=b

V=c

Tabling for Normal Logic Programs

SLG resolution [Chen & Warren '96]

- For positive programs $\equiv$ OLDT resolution [Tamaki & Sato '86]

  Complete for datalog programs: computes minimal models.

- For programs with negation, computes the (three-valued) *well-founded semantics* [van Gelder et al '91]

  – For predicates with *unknown* truth value, generates the set of dependencies that lead to this conclusion.

Well-founded models: An Example

```
p :- q, not r.
q :- not s.
q :- p, r.
r.
s :- not q, r.
```

Model:

True: r

False: p

Unknown: q, s

Residual Program:

```
q :- not s.
s :- not q.
```

XSB: An Implementation of Tabled Resolution

- Conservative extension of the WAM
- Can combine tabled and nontabled (Prolog-style) evaluation in one program.
  - Tabled predicates specially annotated with "`:- table ...`" directive.
  - "`tnot`" signifies tabled (well-founded) negation, distinct from Prolog's `not`.
- Tables represented using *Tries*

  Efficient support for terms in tables.
- Scheduling of tabling operations:

  Equivalent to *semi-naive evaluation*

...other implementations (e.g., YAP) are just beginning to appear...

Operational Behavior of Tabled Programs

- Program resolution for any goal is done at most once.
- Each table has one producer, possibly many consumers.
- Only distinct answers are supplied to consumers.
- When is a consumer C supplied answers from a table for goal G?

  Variance-based: C and G are identical modulo variable renaming.

  Subsumption-based: C is an instance of G.

- Well-founded models are computed in polynomial time.

Estimating Complexity of Tabled Programs

Right-recursive `reach`:

```
:- table reach/2
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- edge(X,Z), reach(Z,Y).
```

Time to evaluate `reach(+,?)`: $O(|V| \cdot |E|)$.

Left-recursive `reach`:

```
:- table reach/2.
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Time to evaluate `reach(+,?)`: $O(|E|)$.

Organization

- Verification by query evaluation
- An overview of Tabled Logic Programming
- ***Representing transition systems***
- Model checking modal mu-calculus
- Infinite-state systems and Constraint LP
- Induction proofs via program transformation
- Symbolic bisimulation for value-passing systems
- Justification of verification proofs

Representing transition systems

- Single-step transitions:
  - Direct representation of automata (one ground fact per transition)
  - Interpreters for process languages [CAV'97]
    - *On-the-fly generation of reachable state space*
  - Rules representing the transition relation [PSTV'99]
- Rules representing the reachability relation [Delzanno & Podelski '99].

  E.g.,

$$p(s,x) \leftarrow x' = f(x), p(s',x')$$

Interpreting Process Languages: CCS

`trans`: Single-step Transition Relation: *State* × *Action* × *State*

Prefix
```
trans(A o P, A, P).
```

Choice
```
trans(P1 # P2, A, Q)   :-trans(P1, A, Q).
trans(P1 # P2, A, Q)   :-trans(P2, A, Q).
```

Restriction
```
trans(P \ L, A, Q \ L) :-trans(P, A, Q),
                          not member(A, L).
```

Relabelling
```
trans(P @ F, A, Q @ F):-trans(P, B, Q),
                        map(F, B, A).
```

Semantics of CCS (contd.)

Parallel composition

```
trans(P | Q, A, P1 | Q):-trans(P, A, P1).
trans(P | Q, A, P | Q1):-trans(Q, A, Q1).
 trans(P | Q, tau, P1 | Q1) :-
                         trans(P, A, P1),
                         trans(Q, B, Q1),
                         complement(A, B).
 complement(in(A), out(A)).
 complement(out(A), in(A)).
```

Definition

```
trans(Pname, A, Q)     :-Pname ::= Pexp,
                         trans(Pexp, A, Q).
```

XL: XMC's Process Specification Language

Supports:

- Concurrency and synchronization *a la* CCS.
- Parameterized processes and channels as parameters.
- Algebraic (possibly recursive) datatypes with polymorphic type inference.
- Embedding computations written in Prolog.

XL: An example

```
medium(Get, Put) ::=  Get ? Data;
                        {   Put ! Data
                        #   action(drop) }; medium(Get, Put).

sender(AckIn, DataOut, Seq) ::=
        %% Seq is the sequence number of the next frame to be sent
        DataOut ! Seq;
        {   AckIn ? AckSeq;
            if (AckSeq == Seq) then {
                 %% successful ack, next message
                 NSeq is 1-Seq;
                 sendnew(AckIn, DataOut, NSeq)
            else %% resend message
                 sender(AncIn, DataOut, Seq)
        #   %% No ack, timeout and resend message
            sender(AncIn, DataOut, Seq)
        }.
.....
```

XMC's Compiler

- Representation: Process terms in XL are translated into *rules* representing global and local transition relations.
- Optimizations:
  - Merges communication-free and choice-free paths into atomic steps
  - Computes potential synchronizations at compile time (where possible)
  - Eliminates dead variables from state expressions.

[PSTV'99]

XMC's Compiler: Sample output

```
trans(medium_0(A,B,C),in(A,D),medium_1(B,A,D,C)).
trans(medium_1(A,B,C,D),out(A,C),medium_0(B,A,D)).
trans(medium_1(A,B,C,D),action(drop),medium_0(B,A,D)).
trans(sender_0(A,B,C,D),out(B,C),sender_1(A,B,C,D)).
...
trans(abp_7_0(sendnew_0(A,B,C,D),E,F,G,H),action(sendnew),
          abp_7_0(sender_0(A,B,C,D),E,F,G,H)).
trans(abp_7_0(A,medium_1(B,C,D,E),F,G,H),action(drop),
          abp_7_0(A,medium_0(C,B,E),F,G,H)).
trans(abp_7_0(sender_1(A,B,C,D),E,medium_1(A,F,G,H),I,J),tau,
          abp_7_0(sendnew_0(A,B,K,D),E,medium_0(F,A,H),I,J)) :-
              G == C,
              K  is  1 - C.
...
```

Organization

- Verification by query evaluation
- An overview of Tabled Logic Programming
- Representing transition systems
- ***Model checking modal mu-calculus***
- Infinite-state systems and Constraint LP
- Induction proofs via program transformation
- Symbolic bisimulation for value-passing systems
- Justification of verification proofs

Modal Mu-calculus: Syntax

```
Fexp -->  Fname
       |  tt
       |  Fexp /\ Fexp
       |  Fexp \/ Fexp
       |  diam(A, Fexp)
       |  diamMinus(A, Fexp)
       |  box(A, Fexp)
       |  boxMinus(A, Fexp)

Definition -->
          Fname -= Fexp
       |  Fname += Fexp
```

An Example: *deadlock freedom*

```
df -= boxMinus(nil,  df) /\ diamMinus(nil, tt)
```

Modal Mu-Calculus: Semantics

```
models(S, tt).

models(S, F1 \/ F2)) :- models(S, F1) ; models(S, F2).

models(S, F1 /\ F2)) :- models(S, F1),  models(S, F2).

models(S, diam(A, F)) :- trans(S, A, T),  models(T, F).

models(S, diamMinus(A, F)) :-
    trans(S, B, T),  A \= B, models(T, F).

models(S, box(A, F)) :-
    forall(T, trans(S, A, T), models(T, F)).

models(S, boxMinus(A, F)) :-
    forall((B,T), (trans(S, B, T), A \= B), models(T, F)).
```

Implementing `forall`

Actual encoding of box formulas makes free variables explicit:

```
models(S, box(A, F)) :-
   forall(T,
      (S,A,F)^trans(S, A, T),
      models(T, F)).

forall(Bv, Fv^Ant, Cons) :-
   findall((Fv,Cons), Ant, L),
   all_true(Fv, L).

all_true(_, []).
all_true(Fv, [(Fv, Cons)|Rest]) :-
   Cons,
   all_true(Fv, Rest).
```

Fixed Points

Minimal model of the logic program $\equiv$ least fixed point.

```
models(S, Fname) :-
    Fname += Fexp,
    models(S, Fexp).
```

Greatest fixed points can be computed using the identity

$$\nu X.f(X) \equiv \neg\mu X.\neg f(\neg X)$$

```
models(S, Fname) :-
    Fname -= Fexp,
    negate(Fexp, NFexp),
    not models(S, NFexp).
```

where `negate(F, NF)` is such that NF $\equiv \neg$F and NF itself doesn't contain '$\neg$'.

Nested Fixed Points

- The well-founded model coincides with (the) 2-valued stable model for (dynamically) stratified programs

  $\Longrightarrow$ implementation is complete for alternation-free fragment of modal mu-calculus

- Alternation in formula leads to non-stratified programs.
  - Results in *signed* programs with stable models. The structure of alternation dictates a preference order among the stable models.
  - Stable models can be computed from the residual program.

Value-Passing Modal Mu-calculus

```
order -= [a(X)] follow(X) /\ [-] order.
follow(X) += <b(X)>tt \/ [-b(X)]follow(X).
```

- Variables are quantified by modalities

  variables in <> are existential; variables in [ ] are universal.

- For finite-state systems, the encoding of the model checker (as it stands) verifies value-passing formulae.

- For handling infinite-state systems we need constraint processing...

Model Checkers for other logics

Proofs may not be trees

Example: LTL and Action LTL [TAPD'00]

- Look for "good paths" (i.e, paths leading to *true* leafs or "good cycles") in a proof graph

Phase 1: Represent proof graph explicitly and search for *true* leafs

Phase 2: Do cycle checking if the search in Phase 1 fails.

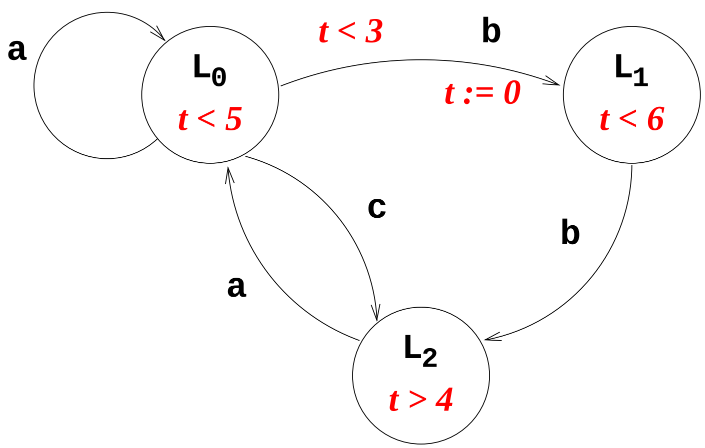
  - Known "delcarative" encodings of cycle detection are nonlinear (e.g., quadratic if **`reach(X,X)`** is used).
  - Linear-time SCC detection can be programmed using table primitives in XSB.

Organization

- Verification by query evaluation
- An overview of Tabled Logic Programming
- Representing transition systems
- Model checking modal mu-calculus
- ***Infinite-state systems and Constraint LP***
- Induction proofs via program transformation
- Symbolic bisimulation for value-passing systems
- Justification of verification proofs

Real-time Systems

Timed Automaton



Encoding

```
trans(l0(T), a, l0(T)).
trans(l0(T), b, l1(0)) :- T < 3.
trans(l0(T), c, l2(T)).
trans(l1(T), b, l2(T)).
trans(l2(T), a, l0(T)).

trans(l0(T0), eps(D), l0(T1)) :-
   T1 = T0+D, T1<5.
trans(l1(T0), eps(D), l1(T1)) :-
   T1 = T0+D, T1<6.
trans(l2(T0), eps(D), l2(T1)) :-
   T1 = T0+D, T1>4.

inv(l0(T)) :- T < 5.
inv(l1(T)) :- T < 6.
inv(l2(T)) :- T > 4.
```

Verifying Reachability Properties

```
reach(X,Y) :- trans(X,_,Y), inv(Y).
reach(X,Y) :- reach(X,Z), trans(Z,_,Y), inv(Y).
```

- X and Y correspond to location/zone pairs.
- Terminates when evaluated using a Constraint LP system with tabling.
  - Needs entailment check when searching through tables.
- Encoding is suited for forward reachability (note the use of location invariants).

  Formulation of backward reachability is similar & straightforward.

Verifying Timed Mu-Calculus Properties

Conservative extension to mu-calculus model checker

1. Universal modality for untimed calculus was encoded operationally, using `forall`.

2. Universal time modality $[\epsilon]$ cannot be encoded in the same way due to quantification over an infinite domain.

3. Elimination of universally quantified interval varible (`D` in above formulation) can be programmed as a basic operation.

4. Needs a ternary `models` relation `models(S, F, SubS)`, where `SubS` is a collection location/zone pairs such that

   - $[\![\texttt{SubS}]\!] \subseteq [\![\texttt{S}]\!]$
   - $\forall \pi \in [\![\texttt{SubS}]\!] \quad \pi \models \texttt{F}$
   - `SubS` is the largest such collection

Implementing Real-Time Model Checkers

No mature LP system that combines constraint processing with tabulation

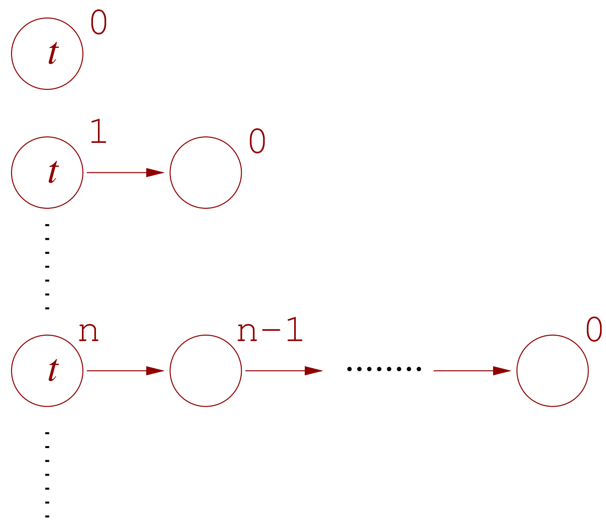
- Preliminary implementation of hooks for constraint libraries in XSB. [Cui & Warren ’00]
- Tabulation implemeted by meta-programming in SICStus Prolog [Delzanno & Podelski ’99; Mukhopadhyay & P. ’00]
- Interface for polyhedra packages with XSB [RTSS’00]

Organization

- Verification by query evaluation
- An overview of Tabled Logic Programming
- Representing transition systems
- Model checking modal mu-calculus
- Infinite-state systems and Constraint LP
- ***Induction proofs via program transformation***
- Symbolic bisimulation for value-passing systems
- Justification of verification proofs

Parameterized Systems

Infinite family of finite-state systems.




```
trans(c(s(M)), c(M)).

base(c(0), p).

ef(S, P) :- base(S, P).
ef(S, P) :-
    trans(S, T), ef(T, P).
```

Model Checking Parameterized Systems

Consider:

```
trans(c(s(M)), c(M)).

base(c(0), p).

ef(S, P) :- base(S, P).
ef(S, P) :- trans(S, T), ef(T, P).
```

- Query "`?- ef(c(`k`), p)`" terminates for any finite k.
- Query "`?- ef(c(N), p)`" enumerates all solutions:

```
N = 0;

N = s(0);

N = s(s(0));
```

A Human Proof

Define:

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

Theorem: $\forall$ `N` `nat(N)` $\Rightarrow$ `ef(c(N), p)`

Proof: By induction on `nat`.

Base case (`N = 0`): `ef(c(0), p)` is true since `base(c(0), p)` is true.

Induction hypothesis: $\forall$ `K` $\leq$ `M` `nat(K)` $\Rightarrow$ `ef(c(K), p)`

Induction step (`N = s(M)`): `ef(c(s(M)), p)` is true because `ef(c(M), p)` is true (by induction hypothesis) and there exists a transition from state `c(s(M))` to state `c(M)` □

Can we extend query evaluation to automate this proof ?

Proof by Program Transformations

Define in program P_0 `thm(N) :- nat(N), ef(c(N),p).`

Convert the proof obligation to a predicate equivalence `thm` $\stackrel{?}{\equiv}$ `nat` in P_0.

Transform definition of `thm` in P_0 to the following definition in a program P_k:

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

The definition of `thm` in program P_k is syntactically equivalent to that of `nat`; hence `thm` $\equiv$ `nat`

Program Transformation : Unfolding

$$
\begin{array}{l}
\mathtt{q} :- G, \boxed{\mathtt{p}}, G'. \\
\mathtt{p} :- Bd_1. \\
\mathtt{p} :- Bd_2. \\
\vdots \\
\mathtt{p} :- Bd_n.
\end{array}
\quad \overset{\textit{Unfold}}{\longrightarrow} \quad
\begin{array}{l}
\mathtt{q} :- G, \; Bd_1, \; G'. \\
\mathtt{q} :- G, \; Bd_2, \; G'. \\
\vdots \\
\mathtt{q} :- G, \; Bd_n, \; G'. \\
\mathtt{p} :- Bd_1. \\
\vdots \\
\mathtt{p} :- Bd_n.
\end{array}
$$

- 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

P_j $(j \leq i)$:

$$\begin{array}{l} \mathbf{p} :- Bd_1. \\ \vdots \\ \mathbf{p} :- Bd_n. \end{array}$$

P_i:

$$\begin{array}{llll} \mathbf{q} :- G, & Bd_1, & G'. \\ \mathbf{q} :- G, & Bd_2, & G'. & \stackrel{Fold}{\longrightarrow} \mathbf{q} :- G, \boxed{\mathsf{p}}, G'. \\ \vdots & & & \\ \mathbf{q} :- G, & Bd_n, & G'. \end{array}$$

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

Verification by Transformations: An Example

P_0 :

```
thm(N) :- nat(N), ef(c(N),p).
nat(0).
nat(s(N)) :- nat(N).
ef(S,P) :- base(S,P).
ef(S,P) :- trans(S,T), ef(T,P).
base(c(0),p).
trans(c(s(N)), c(N)).
```

↓ *Unfold* (Discovering induction schema)

P_1 :

```
thm(0) :- ef(c(0),p).
thm(s(N)) :- nat(N), ef(c(s(N)), p).
```

Transformation Example (contd.)

P_1 :

```
thm(0) :- ef(c(0), p).
thm(s(N)) :- nat(N), ef(c(s(N)), p).
```

↓ *Unfolds* (Base case completed)

P_4 :

```
thm(0).
thm(s(N)) :- nat(N), ef(c(s(N)), p).
```

↓ *Unfolds* (Finite part of induction step)

P_7 :

```
thm(0).
thm(s(N)) :- nat(N), ef(c(N), p).
```

↓ *Fold* (Applying induction hypothesis)

P_8 :

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

Verification by Program Transformation

- Can do nested inductions, using goal replacement transformations.
- Strategies to control the order of transformations:
  - Apply model checking (unfolding) steps as much as possible.
  - Allow interleaving of model checking and deductive (folding, replacement) steps.
- [TACAS'00] Implemented our control strategies to produce proofs for:
  - Liveness in chains (previous example)
  - Mutual exclusion in token rings,
  - Liveness in a family of binary trees, etc.

Organization

- Verification by query evaluation
- An overview of Tabled Logic Programming
- Representing transition systems
- Model checking modal mu-calculus
- Infinite-state systems and Constraint LP
- Induction proofs via program transformation
- ***Symbolic bisimulation for value-passing systems***
- Justification of verification proofs

Bisimulation for basic LTS

Given a labeled transition system $L = (S, T)$, $\mathcal{R}$ is a bisimulation relation if $\mathcal{R}$ is the largest relation such that

$$\begin{array}{lll} \forall s_1, s_2 \in S \quad s_1 \,\mathcal{R}\, s_2 & \Rightarrow & \forall (s_1, a, t_1) \in T \quad \exists (s_2, a, t_2) \in T \quad t_1 \,\mathcal{R}\, t_2 \\ & \wedge & \forall (s_2, a', t_2') \in T \quad \exists (s_1, a', t_1') \in T \quad t_1' \,\mathcal{R}\, t_2' \end{array}$$

Consider $\overline{\mathcal{R}}$, the complement of the bisimulation relation:

$$\begin{array}{lll} \forall s_1, s_2 \in S \quad s_1 \,\overline{\mathcal{R}}\, s_2 & \Leftarrow & \exists (s_1, a, t_1) \in T \quad \forall (s_2, a, t_2) \in T \quad t_1 \,\overline{\mathcal{R}}\, t_2 \\ & \vee & \exists (s_2, a', t_2') \in T \quad \forall (s_1, a', t_1') \in T \quad t_1' \,\overline{\mathcal{R}}\, t_2' \end{array}$$

$\overline{\mathcal{R}}$ is the smallest such relation (i.e., least model for the logical formula above).

Implementing Bisimulation Checking

```
:- table nbisim/2.
nbisim(S1, S2) :-
   trans(S1, A, T1),
   no_matching_trans(S2, A, T1).
nbisim(S1, S2) :- nbisim(S2, S1).
no_matching_trans(S2, A, T1) :-
   forall(T2, trans(S2,A,T2), nbisim(T1, T2)).
bisim(S1, S2) :- tnot(nbisim(S1,S2)).
```

Performs *LOCAL* bisimulation checking

Time Complexity: $O(|S| \times |T|)$ assuming unit-time table lookups.

(Tables as binary trees introduces $O(log|S|)$ factor.)

Implementing `forall`

In practice, encoding of `forall` makes free variables explicit.

E.g.,

```
no_matching_trans(S2, A, T1) :-
   forall(T2,
      (S2,A,T1)^trans(S2, A, T2),
      nbisim(T1, T2)).

forall(Bv, Fv^Ant, Cons) :-
   findall((Fv,Cons), Ant, L),
   all_true(Fv, L).

all_true(_, []).
all_true(Fv, [(Fv, Cons)|Rest]) :-
   Cons,
   all_true(Fv, Rest).
```

Bisimulation for Symbolic LTS

Given an extended LTS $L = (S, T)$, $\mathcal{R}_l$ is a ***early*** bisimulation relation if $\mathcal{R}_l$ is the largest relation such that

$$\forall s_1, s_2 \in S \quad s_1 \; \mathcal{R}_l \; s_2 \quad \Rightarrow \quad \forall (s_1, a, t_1) \in T$$

$$\boxed{\forall \sigma} \quad \exists (s_2, a', t_2) \in T \text{ such that}$$

$$a\sigma \; \succeq \; a'\sigma \quad \wedge \quad t_1\sigma \; \mathcal{R}_l \; t_2\sigma$$

$\wedge$... *the symmetric case...*

$\mathcal{R}_e$ is a ***late*** bisimulation relation if $\mathcal{R}_e$ is the largest relation such that

$$\forall s_1, s_2 \in S \quad s_1 \; \mathcal{R}_e \; s_2 \quad \Rightarrow \quad \forall (s_1, a, t_1) \in T$$

$$\exists (s_2, a', t_2) \in T \quad \boxed{\forall \sigma} \text{ such that}$$

$$a\sigma \; \succeq \; a'\sigma \quad \wedge \quad t_1\sigma \; \mathcal{R}_e \; t_2\sigma$$

$\wedge$... *the symmetric case...*

Implementing Symbolic Bisimulation Checking — I

```
:- table nbisim/2.

nbisim(S1, S2) :-
   strans(S1, A1, C1, T1),
   C1,
   no_matching_trans(S2, S1, A1, T1).
nbisim(S1, S2) :- nbisim(S2, S1).

no_matching_trans(S2, S1, A1, T1) :-
   forall((A2,C2,T2),
      (S1,A1,T1,S2)^strans(S2, A2, C2, T2),
      nsimulate(A1,T1, A2,C2,T2) ).

bisim(S1, S2) :- tnot(nbisim(S1,S2)).
```

Implementing Symbolic Bisimulation Checking — II

```
nsimulate(A1,T1, A2,C2,T2) :-
    (C2,
        copy_term((A1,T1,A2,T2), (B1,U1,B2,U2)), % Late
        (A1,T1,A2,T2) = (B1,U1,B2,U2), % Early
        ( B2 =>= B1, nbisim(U1, U2)
        ; B2 />= B1)
    )
    ; not(C2)
    .
```

[TAPD'00]

Organization

- Verification by query evaluation
- An overview of Tabled Logic Programming
- Representing transition systems
- Model checking modal mu-calculus
- Infinite-state systems and Constraint LP
- Induction proofs via program transformation
- Symbolic bisimulation for value-passing systems
- ***Justification of verification proofs***

Justifier

- Constructs sufficient evidence of proof/disproof *after* verification run by inspecting lemmas in memo tables.
- Adds no overhead (time or space) to prover.
- Presents abstractions of proof/disproof tree to user; user may “walk” the tree interactively.
- *Can be used to construct tree of MSCs.*

[PPDP’00]

Justifying Logic Programs: The Basic Idea

Given a goal p, show one step in its derivation (or evidence of lack of derivation).

p **is true:** get a clause $H :- B$ such that $mgu(H, p) = \theta$, and

every literal q_j in $B\theta$ is true, and

q_j is not an ancestor in the justification.

p **is false:** Find all clauses $H_i :- B_i$ such that $mgu(H_i, p) = \theta_i$.

Pick q_{ij} from each $B_i\theta_i$ such that q_{ij} is false, and $\forall\ k < j \ \ q_{ik}$ is true.

p **is unknown:** Find all clauses $H_i :- B_i$ such that $mgu(H_i, p) = \theta_i$.

Pick q_{ij} from each $B_i\theta_i$ such that q_{ij} is unknown, and $\forall\ k \neq j \ \ q_{ik}$ is true/unknown.

Justifying Meta Programs

- Allow arbitrary combination of tabled and non-tabled goals.
- Permit user-specifiable justifications for library predicates (e.g., '$\forall$' is justified in terms of '$\neg\exists\neg$'
- Convert logic program proof graphs to higher-level structures using graph (tree) transformations

Uniform "core" method for showing proofs, counter examples and bisimulation games.

Summary

LP-based formulation and implementation of verification techniques offer

Elegance: Succinct

Efficiency: As fast as existing systems

Expressiveness: Value-passing languages, symbolic (constraint-based) evaluation

Extensibility: Mix-and-match logics and tool interfaces

Unaddressed issues: Space consumption, control of search, special-purpose data structures, ...