

Logic Programming and Model Checking

C. R. Ramakrishnan
Scott A. Smolka

SUNY, Stony Brook

SAS/PLILP/ALP'98, Pisa, Italy

September '98

Logic Programming-Based Model Checking

LMC Project: Experimental Software Systems project to explore 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.

PLILP'98

Logic Programming and Model Checking

1

LMC Project Members

Rance Cleaveland	C. R. Ramakrishnan
Baoqiu Cui	I. V. Ramakrishnan
Yifei Dong	Abhik Roychoudhury
Xiaoqun (Vic) Du	Scott A. Smolka
Narayan Kumar	Terrance Swift
Madhavan Mukund	David S. Warren
Y.S. Ramakrishna	

PLILP'98

Logic Programming and Model Checking

2

Outline

1. Brief introduction to Model Checking
2. Tabling and XSB
3. Encoding Model Checkers as Logic Programs
4. Beyond Finite-State Model Checking
5. Future Directions

PLILP'98

Logic Programming and Model Checking

3

Model Checking

$$S \models f \quad ?$$

- System specifications typically written in a process calculus (e.g., CCS)
- Property specifications expressed in temporal logic

PLILP'98

Logic Programming and Model Checking

4

Process Specifications

- Systems specified using various formalisms, including **CCS** (Calculus of Communicating Systems).
- Processes perform basic operations, such as communicating over a port (CCS), or setting a shared variable (CSP).
- Process can be composed using
 - **Parallel composition**
 - **Choice** (non-determinism)
 - **Prefix** (sequence)
 - **Restriction and Relabeling** (modules)

PLILP'98

Logic Programming and Model Checking

5

Property Specifications

Properties of interest are expressed in temporal logics:

- Linear-time and Branching-time Temporal Logics (e.g., LTL, CTL, CTL*).
- **AF** (`bit32.carry_out`): On every path there exists a state (future) where `bit32.carry_out` is true.
- **Modal Logics** (e.g., Modal mu-calculus).

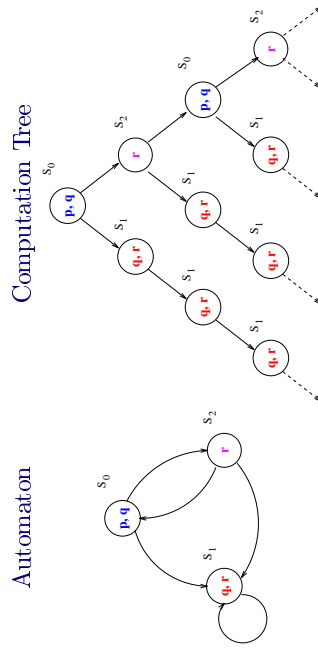
$$\mu X. (\text{bit32.carry_out}) . \# \vee [-\text{bit32.carry_out}] . X$$

PLILP'98

Logic Programming and Model Checking

6

Computation Trees



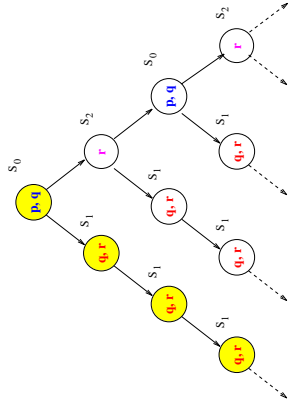
PLILP'98

Logic Programming and Model Checking

7

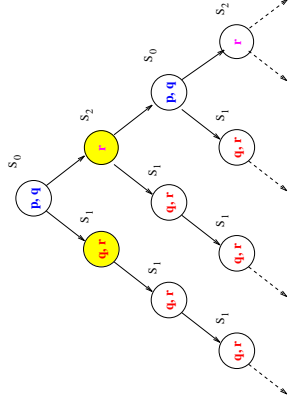
Temporal Logic Model Checking - I

$$S_0 \models EG q$$



Temporal Logic Model Checking - II

$$S_0 \models AF r$$



CTL Model Checker as a Logic Program

`trans(S1, A, S2)`: transition relation (from system specifications)

`models(S, F)`: Does system state S model formula F?

`models(S, ef(F))` :- `models(S, F)`.

`models(S, ef(F))` :- `trans(S, -, T), models(T, ef(F))`.

`models(S, af(F))` :- `models(S, F)`.

`models(S, af(F))` :- `forall(T, trans(S, -, T), LS),`
`all_models(LS, af(F))`.

...

Local vs. Global Model Checking

- ▷ Local Model Checking: Goal-directed evaluation.
 Explores only those states necessary to prove (or disprove) a formula.
 - ▷ Global Model Checking: Bottom-up evaluation.
 Explores the entire state space of the system.
- !! State space of the system can grow exponentially with the number of concurrent processes.**

XMC

Value passing CCS, full modal mu-calculus.

Derived from high-level specifications of the semantics of process language and temporal logic.

The Approach:

- Represent the equations describing the operational semantics of CCS (in terms of labeled transition systems) a logic program.
- Encode the SOS semantics of modal- μ calculus as another logic program.
- Specify the system in CCS, and the temporal formula in modal μ -calculus (EDB facts).
- Query: Is the formula true in the initial state of the CCS program? Evaluate the query using tabled resolution.

PLILP'98

Logic Programming and Model Checking

12

XMC Implementation

- The rules describing the semantics of CCS and modal mu-calculus are transformed using several source-level optimizations. *e.g.*,
 - ▷ Clause resolution factoring
 - ▷ Literal Reordering
 - ▷ Mode-based specialization
- **System size:** < 200 lines of Tabled Prolog code.

PLILP'98

Logic Programming and Model Checking

13

Outline

- Brief introduction to Model Checking
 - Tabling and XSB
- Encoding Model Checkers as Logic Programs
- Beyond Finite-State Model Checking
- Future Directions

PLILP'98

Logic Programming and Model Checking

14

What is Tabled Resolution?

Memoize the results of computations 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.

PLILP'98

Logic Programming and Model Checking

15

Tabled Resolution

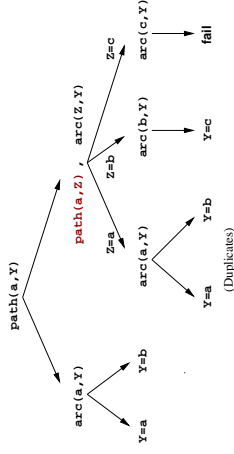
Record goals in *call table* and their provable instances in *answer table*.

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

```
path(x,y) :- arc(x,y).
path(x,y) :- path(x,z), arc(z,y).
arc(a,a).
arc(a,b).
arc(b,c).
```



Calls
`path(a,v)`
Answers
`path(a,a)`
`path(a,b)`
`path(a,c)`

XSB

Full-fledged Prolog system + Tabling, using *SLG* resolution.

- For positive programs \equiv OLDT resolution
- For programs with negation, computes the well-founded semantics.
 - For predicates with *unknown* truth value, generates the set of dependencies that lead to this conclusion.

Complete for datalog programs: computes minimal models.

XSB Tabled Logic Programming System

- Prolog performance comparable with state-of-the-art emulated systems.
- Computes minimal models for in-memory programs an order of magnitude faster than best-known deductive database systems.
- Fastest known system for computing well-founded models of normal logic programs.

Efficient organization of tables using trie data structures.

Useful features of XSB

- Goal-directed evaluation: *Local model checking* “for free”.
- Conservative extension of Logic Programming:
 - **WAM-based engine**: Traditional Logic Programming optimizations can be easily incorporated.
 - **Metaprogramming**: Representation and manipulation of constraints (can be used to represent infinite state spaces).

PLILP'98

Logic Programming and Model Checking

20

Negation and XSB

XSB evaluates the well-founded semantics.

- Computes two-valued models for (dynamically) stratified programs:
 - Direct encoding for a fragment of modal mu-calculus: nested (but non-alternating) fixed point formulae.
- Generates residual program that captures the dependencies between the predicates that have *unknown* values in the WFS.
 - Alternating fixed point formulae evaluated by post-processing of the residual program to find the “preferred” stable model.

PLILP'98

Logic Programming and Model Checking

21

Outline

- Brief introduction to Model Checking
- Tabling and XSB
 - Encoding Model Checkers as Logic Programs
- Beyond Finite-State Model Checking
- Future Directions

PLILP'98

Logic Programming and Model Checking

22

Syntax of Basic CCS

Milner'89	XMC
$P_{exp} \rightarrow P_{name}$	$P_{exp} \rightarrow P_{name}$
$\alpha.P_{exp} \quad \alpha \in Action$	$in(Port) \circ P_{exp}$
$P_{exp} + P_{exp}$	$out(Port) \circ P_{exp}$
$P_{exp} P_{exp}$	$P_{exp} \# P_{exp}$
$P_{exp} \setminus L$	$P_{exp} P_{exp}$
$P_{exp}[f]$	$P_{exp} \setminus \{port\ list\}$
	$P_{exp} @ [port\ map]$
$P_{def} \rightarrow P_{name} \stackrel{def}{=} P_{exp}$	$P_{def} \rightarrow P_{name} ::= P_{exp}$

Example:

$$p1 \stackrel{def}{=} a! . (b? . p2 + c? . p1)$$

$$p1 ::= out(a) \circ (in(b) \circ p2 \# in(c) \circ p1)$$

PLILP'98

Logic Programming and Model Checking

23

Semantics of CCS

From Milner's C&C book (p. 46):

$$\frac{}{\alpha.E \xrightarrow{\alpha} E}$$

$$\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}$$

$$\frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}$$

$$\frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$\frac{E \xrightarrow{\alpha} E', F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E'|F'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad (\alpha, \bar{\alpha} \notin L)$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

PLILP'98

Logic Programming and Model Checking

24

Encoding the Semantics of CCS

trans: Single-step Transition Relation: $State \times Action \times State$

Prefix $\text{trans}(A \circ P, A, P)$.

Choice $\text{trans}(P1 \# P2, A, Q) \text{ :- trans}(P1, A, Q)$.
 $\text{trans}(P1 \# P2, A, Q) \text{ :- trans}(P2, A, Q)$.

Restriction $\text{trans}(P \setminus L, A, Q \setminus L) \text{ :- trans}(P, A, Q)$,
 not member(A, L).

Relabelling $\text{trans}(P \circ F, A, Q \circ F) \text{ :- trans}(P, B, Q)$,
 map(F, B, A).

PLILP'98

Logic Programming and Model Checking

25

Semantics of CCS (contd.)

Parallel $\text{trans}(P \mid Q, A, P1 \mid Q) \text{ :- trans}(P, A, P1)$.

composition $\text{trans}(P \mid Q, A, P \mid Q1) \text{ :- trans}(Q, A, Q1)$.

$\text{trans}(P \mid Q, \text{tau}, P1 \mid Q1) \text{ :-}$
 $\text{trans}(P, A, P1)$,
 $\text{trans}(Q, B, Q1)$,
 complement(A, B).

complement(in(A), out(A)).
 complement(out(A), in(A)).

Definition $\text{trans}(\text{Fname}, A, Q) \text{ :- Phame} ::= \text{Pexp}$,
 $\text{trans}(\text{Pexp}, A, Q)$.

PLILP'98

Logic Programming and Model Checking

26

Modal Mu-calculus: Syntax

Fexp --> Fname
 | tt
 | Fexp \wedge Fexp
 | Fexp \vee 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) \wedge diamMinus(nil, tt)

PLILP'98

Logic Programming and Model Checking

27

Modal Mu-Calculus: Semantics

```
models(S, tt).
models(S, F1  $\vee$  F2) :- models(S, F1) ; models(S, F2).
models(S, F1  $\wedge$  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  $\setminus$  = B, models(T, F).
models(S, box(A, F)) :-
  forall(T, trans(S, A, T), L), map_models(L, F).
models(S, boxMinus(A, F)) :-
  forall(T, (trans(S, B, T), A  $\setminus$  = B), L), map_models(L, F).
```

PLILP'98

Logic Programming and Model Checking

28

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 $\text{nu}(F) \equiv \neg \mu(\neg F)$.

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

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

PLILP'98

Logic Programming and Model Checking

29

Nested Fixed Points

- XSB computes 2-valued models for (dynamically) stratified programs
 \implies 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 independently, based on the residual program generated by XSB.

PLILP'98

Logic Programming and Model Checking

30

Optimizations: Literal Reordering

```
trans(P  $\vee$  ||' Q, tau, P1  $\vee$  ||' Q1) :-
  trans(P, A, P1),
  trans(Q, B, Q1),
  complement(A, B).
   $\Downarrow$ 
trans(P  $\vee$  ||' Q, tau, P1  $\vee$  ||' Q1) :-
  trans(P, A, P1),
  complement(A, B),
  trans(Q, B, Q1).
```

PLILP'98

Logic Programming and Model Checking

31

Optimizations: Clause Resolution Factoring

- Share operations across program-clause and answer-clause resolution steps.
- Clause-level (instead of predicate-level) tabling.

PLILP'98

Logic Programming and Model Checking

32

Clause Resolution Factoring

```
:- table trans/3.

trans(Pname, A, Q) :- Pname ::= Pexp, trans(Pexp, A, Q).
trans(A o P, A, P).
trans(P1 # P2, A, Q) :- trans(P1, A, Q) ; trans(P2, A, Q).
↓
:- table trans_rec/3.

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

trans(Pname, A, Q) :- trans_rec(Pname, A, Q).
trans(A o P, A, P).
trans(P1 # P2, A, Q) :- trans(P1, A, Q) ; trans(P2, A, Q).
```

PLILP'98

Logic Programming and Model Checking

33

Optimizations: Specialization based on modes

```
trans(P \ L, A, Q \ L) :- trans(P, A, Q), not member(A, L).
↓
trans(P \ L, A, Q \ L) :-
  (var(A) -> (trans(P, A, Q), not member(A, L))
  ; (not member(A, L), trans(P, A, Q))).
```

PLILP'98

Logic Programming and Model Checking

34

Value Passing Language

Exchange data values along channels.

- Communication primitives and process names are *terms*.
- *if-then-else* primitive to “test” values.
- Internal computation specified by (possibly user defined) Prolog predicates.

PLILP'98

Logic Programming and Model Checking

35

Value Passing (cont.d.)

Example:

```
channel(N, B) ::=
length(B, L) o
if(L == 0
    , read_only(N, B)
    , if (L == N
        , write_only(N, B)
        , read_only(N, B) # write_only(N, B)
    )) .
write_only(N, [X|R]) ::= out(put(X)) o channel(N, R) .
...
```

Extending XMC with Value Passing

- Add rules for `if` and internal computation.
- Variables in processes are Prolog variables:
 - **No code needed to manipulate variables**— substitutions, renaming etc. are done as and when needed by the LP engine
- Values are passed between processes at the time of synchronization by *unification*.
 - Can be used to simulate shared variables.

Performance on Value Passing examples

Comparison with SPIN (without partial order reduction)

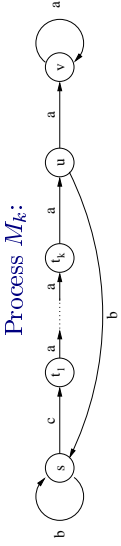
Program	System	Time (sec)	Space (MB)
leader5	SPIN	8.1	9.60
	XMC	5.5	0.78
sieve6	SPIN	1.8	2.31
	XMC	10.4	1.23

A More Challenging Example

Livelock detection in i-Protocol (from GNU UUCP stack)

Version	Tool	Completed?	Memory (MB)	Time (min:sec)
W=1 *fixed	SPIN	Yes	749	0:10
	XMC	Yes	18.4	0:03
W=1 fixed	SPIN	Yes	820	1:02
	XMC	Yes	128	0:46
W=2 *fixed	SPIN	Yes	751	0:12
	XMC	Yes	68	0:11
W=2 fixed	SPIN	Yes	1789	6:23
	XMC	Yes	688	3:48

Alternating Fixed Points in XMC: Performance



Formula F : $\nu X. \mu Y. ([-], ((a) \# \wedge X) \vee Y)$

Instance	CMC	FAM	XMC
$M_{500} \models F$	33.84	2.88	1.61
$M_{1000} \models F$	138.51	11.64	2.76
$M_{1500} \models F$	312.10	26.61	4.08

Synchronous CCS

Concurrent processes composed using *product* operation ' \times ' that allows components to proceed synchronously.

Idling action ' $*$ ' used to build asynchronous behavior over products.

$$E \xrightarrow{*} E \quad \alpha.E \xrightarrow{\alpha} E$$

$$\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E' + F} (\alpha \neq *) \quad \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} E + F'} (\alpha \neq *)$$

$$\frac{E \xrightarrow{\alpha} E', F \xrightarrow{\beta} F'}{E \times F \xrightarrow{\alpha \times \beta} E' \times F'} \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} (A \stackrel{def}{=} P, \alpha \neq *)$$

SCCS Model Checking

• **trans** relation can be defined, similar to CCS, based on the operational rules.

• Additional rules needed to ensure that the following algebraic laws hold:

$$* \times * = *$$

$$\alpha \times \bar{\alpha} = *$$

• The algebraic laws are applied only when actions are compared, as in $\alpha.E \xrightarrow{\alpha} E$ and can be enforced "lazily".

Compositional Model Checking

- Verify property of a system
 - ▷ based on the properties of its individual components,
 - ▷ and not by constructing the global transition system.
- Enables *modular* verification of systems.
- Decomposes a verification problem into potentially simpler verification tasks for the subcomponents.
- Facilitates *reuse* of verified components.

Compositional Model Checking for CCS

Model checking rules (models) will be directly examine the structure of the processes, instead of the transition relation, **trans**.

Examples:

$$\frac{P_1 + P_2 \vdash [\alpha]F}{P_1 \vdash [\alpha]F \wedge P_2 \vdash [\alpha]F}$$

$$\frac{P_1 + P_2 \vdash \langle \alpha \rangle F}{P_1 \vdash \langle \alpha \rangle F} \quad \frac{P_1 + P_2 \vdash \langle \alpha \rangle F}{P_2 \vdash \langle \alpha \rangle F}$$

PLILP'98

Logic Programming and Model Checking

44

Inference rules for Compositional Model Checking

Sample rules:

$$\frac{(P_1 + P_2) \mid P_3 \vdash [\alpha]F}{P_1 \mid P_3 \vdash [\alpha]F \wedge P_2 \mid P_3 \vdash [\alpha]F} \quad \frac{(P_1 \mid P_2) \mid P_3 \vdash [\alpha]F}{P_1 \mid (P_2 \mid P_3) \vdash [\alpha]F}$$

$$\frac{(P_1 \setminus L) \mid P_2 \vdash [\alpha]F}{(P_1[f_L] \mid P_2) \setminus f_L(L) \vdash [\alpha]F} \quad f_L \text{ renames elements of } L \text{ to new names}$$

$$\frac{(\beta.P_1)[f] \mid P_2 \vdash [\alpha]F}{(f(\beta).P_1[f]) \mid P_2 \vdash [\alpha]F}$$

$$\frac{\beta.P_1 \mid P_2 \vdash [\alpha]F}{P_2 \vdash [\alpha](F/\beta.P_1) \wedge P_1 \mid P_2 \vdash F \text{ if } \alpha = \beta \wedge P_2 \vdash [\beta]F/P_1 \text{ if } \beta \neq \alpha = \tau}$$

PLILP'98

Logic Programming and Model Checking

45

Implementation of Compositional Model Checking

- Single models predicate, with auxiliary definitions for
 - ▷ inventing new names (f_L), and
 - ▷ pushing relabels and restrictions into the formula.
- Intermediate global states are not materialized, and hence there is a potential for saving space.
- Worst case behavior is same as that of the original model checker: verification time is proportional to the size of global state space.
- Can verify certain infinite-state systems where the original model checker fails to terminate (attempting to construct the global state space).

Current work: extensions with value passing.

PLILP'98

Logic Programming and Model Checking

46

Outline

- Brief introduction to Model Checking
- Tabling and XSB
- Encoding Model Checkers as Logic Programs
 - Beyond Finite-State Model Checking
- Future Directions

PLILP'98

Logic Programming and Model Checking

47

Beyond Finite-state Model Checking

- Verification of infinite families of systems:
Combining model checking with *deductive methods*, such as induction.
- Verification of Real-time systems:
Model checking with real-time temporal logics.

PLILP'98

Logic Programming and Model Checking

48

Induction

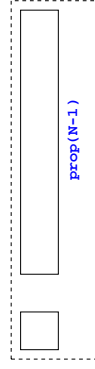
- Induction is needed for verifying properties of all members of an infinite family of finite-state systems.
- Infinite families include systems comprised of processes connected by recursively defined topologies, such as rings and trees.
- Can we exploit the recursive definition of infinite families to obtain an induction proof for a nontrivial class of systems and properties?

PLILP'98

Logic Programming and Model Checking

49

Induction



$$\frac{\text{prop}(N)}{\vdash \forall N \quad \Phi(N)}$$

$$\frac{}{\vdash \Phi(0) \wedge \forall M(\Phi(M) \vdash \Phi(M+1))}$$

$\text{phi}(0).$
 $\text{phi}(M+1) :- \text{phi}(M)$

PLILP'98

Logic Programming and Model Checking

50

Approach to Induction

Start with a query where the induction variables (parameter of the infinite family) are unbound.

Harness XSB's ability to generate conditional answers to construct induction schema:

- Stop when induction hypothesis is reached
- ▷ Hypothesis \equiv theorem \Rightarrow hypothesis is reached when the current call, containing induction variables, is same as a subgoal previously encountered.
- ▷ Generate conditional answer instead of *failing*.
- Make induction structure explicit by *folding*.

PLILP'98

Logic Programming and Model Checking

51

Prospects and Limitations

- Combines model checking and induction
without compromising model checking speeds
 - Searches for induction proofs whose structure is identical to the recursive structure of the infinite family's definition.
 - Searches only for proofs that do not require strengthening of induction hypotheses.
- Additional inference rules are needed to strengthen hypotheses.
- Current status: A preliminary implementation capable of proving nontrivial, yet simple, theorems: e.g.,:
 - associativity of append,
 - liveness formulas on token rings,
 - correctness of carry lookahead addition

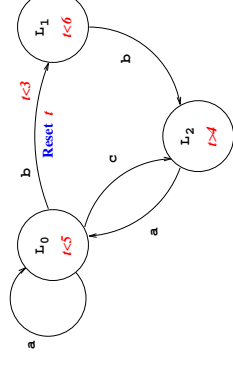
PLILP'98

Logic Programming and Model Checking

52

Real-time systems

Timed Automata



- Clock variables that take values over a discrete time domain
- Constraints on clocks at (locations), and on arcs
- Clocks may be reset on when an arc is traversed

PLILP'98

Logic Programming and Model Checking

53

Mapping real-time to finite-state systems

- State of the system \equiv Location \times Clock values
 - For a suitable set of constraints, timed systems can be reduced to finite state systems.
- Constraints on clock variables must be of the form $X < Y + c$, where c is a constant.
- For such constraints, the state space is split into a finite number of *indistinguishable* regions.
 - Two approaches to model checking:
 - ▷ Construct the equivalent finite state system *a priori*
 - ▷ Start with an assumption that states can be distinguished only based on the location;
Refine regions when model checking

PLILP'98

Logic Programming and Model Checking

54

Local Model Checking of real-time systems

Ongoing work.

- Add one more inference rule to the model checker:

$$\frac{R \vdash F}{R \text{ refinesto } \{R_i\}, \forall_i R_i \vdash F}$$

- **refinesto** relation can be specified as Horn clauses, *with constraints*.
- Refinement of a region creates new arcs;
Conditions (and destination states) of arcs lead to refinement;
Hence **refinesto** is mutually recursive with **arc**.

PLILP'98

Logic Programming and Model Checking

55

Outline

- Brief introduction to Model Checking
- Tabling and XSB
- Encoding Model Checkers as Logic Programs
- Beyond Finite-State Model Checking
 - Future Directions

PLILP'98

Logic Programming and Model Checking

56

Future Directions: Efficiency

- Optimizing state-space search: e.g., partial-order reduction
- Source-level Optimizations: bisimulation & preorder reductions
- Reducing state space using *abstraction*
- Representation: combining “symbolic” data structures with current methods

PLILP'98

Logic Programming and Model Checking

57

Future Directions: Ease of Use

- Modeling languages should support
 - State encapsulation
 - Support for types
- Integration with GCCS
- Interactive justification for proofs

PLILP'98

Logic Programming and Model Checking

58

Justifier and Proof traces

- Query execution searches for proofs using the inference rules.
- The lemmas used in the proof are remembered in the tables.
- The proof can be reconstructed from the tables (without searching).
- The inference rules directly encode the semantics of the process language and temporal logic:
 - ▷ An user can explore the proof tree interactively in terms of the semantic rules of process language and temporal logic, without having to know the operational details of the model checking algorithm.

PLILP'98

Logic Programming and Model Checking

59

Moral of the Story

- Rapid “prototyping” of model checkers
- ... without sacrificing efficiency
- Platform for integrating model checking and deduction
- Constraints + Tabling for verification of infinite-state systems
- High-level (declarative) debugging