Logic Programming and Model Checking

C. R. Ramakrishnan
Scott A. Smolka

SUNY, Stony Brook

SAS/PLILP/ALP'98, Pisa, Italy
September '98

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.

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

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
Model Checking

\[ S \models f \]

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

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)

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).
  \[ \mu X.(\text{bit32.carry_out} \land \text{tt} \lor \neg\text{bit32.carry_out}).X \]

Computation Trees

Automaton

Computation Tree

\[ S \]
CTL Model Checker as a Logic Program

\texttt{trans(S1, A, S2): transition relation (from system specifications)}
\texttt{models(S, F): Does system state S model formula F?}
\texttt{models(S, ef(F)) :- models(S, F).}
\texttt{models(S, ef(F)) :- trans(S, _, T), models(T, ef(F)).}
\texttt{models(S, af(F)) :- models(S, F).}
\texttt{models(S, af(F)) :- findall(T, trans(S, _, T), LS),}
\texttt{ 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 μ-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.

### XMC Implementation

- The rules describing the semantics of CCS and modal μ-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.

### Outline

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

---

### 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.
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) :- path(X,Z), arc(Z,Y).
path(X,Y) :- arc(X,Y).
arc(a,a).
arc(a,b).
arc(b,c).
```

Calls

- `path(a,V)`
- `arc(a,Y)`

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).

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.

Outline

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

<table>
<thead>
<tr>
<th>Syntax of Basic CCS</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Milner’89</th>
<th>XMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xleftarrow{} Pname$</td>
<td>$P \xleftarrow{} Pname$</td>
</tr>
<tr>
<td>$\alpha P \xleftarrow{} Pname \cdot \alpha \in \text{Action}$</td>
<td>$\text{in}(\text{Port}) \circ Pexp$</td>
</tr>
<tr>
<td>$P \xleftarrow{} (P + Pexp)$</td>
<td>$\text{out}(\text{Port}) \circ Pexp$</td>
</tr>
<tr>
<td>$P \xleftarrow{} Pexp \cdot Pexp$</td>
<td>$Pexp # Pexp$</td>
</tr>
<tr>
<td>$P \xleftarrow{} Pexp \cdot Pexp$</td>
<td>$Pexp \mid Pexp$</td>
</tr>
<tr>
<td>$P \xleftarrow{} Pexp \cdot L$</td>
<td>$Pexp \mid {\text{port list}}$</td>
</tr>
<tr>
<td>$P \xleftarrow{} Pexp \cdot [\text{port map}]$</td>
<td>$Pexp \circ [\text{port map}]$</td>
</tr>
</tbody>
</table>

Example:

\[ p_1 \xleftarrow{} a \cdot (b? \cdot p_2 + c? \cdot p_1) \]

\[ p_1 \xleftarrow{} \text{out}(a) \circ (\text{in}(b) \circ (p_2 \# \text{in}(c) \circ p_1)) \]
Semantics of CCS

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

\[ \alpha.E \xrightarrow{\alpha} E \]
\[ E \rightarrow F \xrightarrow{=} E + F \rightarrow E' \]
\[ F \rightarrow F' \]
\[ E \rightarrow E', F \rightarrow F' \xrightarrow{=} E | F \rightarrow E' | F' \]
\[ P \rightarrow P' (A \overset{\text{def}}{=} P) \]
\[ P \rightarrow P' (A \overset{\text{def}}{=} P) \]
\[ P \rightarrow P' (A \overset{\text{def}}{=} P) \]
\[ P \rightarrow P' (A \overset{\text{def}}{=} P) \]
\[ P \rightarrow P' (A \overset{\text{def}}{=} P) \]

Encoding the Semantics of CCS

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

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

Choice \( \text{trans}(P_1 \# P_2, A, Q) \)

Restriction \( \text{trans}(P \setminus L, A, Q \setminus L) ; \neg \text{member}(A, L) \)

Relabelling \( \text{trans}(P @ F, A, Q @ F) ; \text{map}(F, B, A) \)

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

\( \text{df} ← \text{boxMinus}(\text{nil}, \text{df}) \setminus \text{diamMinus}(\text{nil}, \text{tt}) \)
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)) :-
    findall(T, trans(S, A, T), L), map_models(L, F).
models(S, boxMinus(A, F)) :-
    findall(T, (trans(S, B, T), A ⊆ B), L), map_models(L, F).

Fixed Points

Minimal model of the logic program ≡ least fixed point.
models(S, Fname) :-
    Fname ∈= Fexp,
    models(S, Fexp).
Greatest fixed points can be computed using nu(F) ≡ ¬ nu(¬ F).
models(S, Fname) :-
    Fname ←= Fexp,
    negate(Fexp, NFexp),
    not models(S, NFexp).

where negate(F, NF) is such that NF ≡ ¬ F and NF itself doesn’t contain ‘¬’.

Nested Fixed Points

- XSB computes 2-valued models for (dynamically) stratified programs
  – 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.

Optimizations: Literal Reordering

trans(P '||' Q, tau, P1 '||' Q1) :-
    trans(P, A, P1),
    trans(Q, B, Q1),
    complement(A, B).
trans(P '||' Q, tau, P1 '||' Q1) :-
    trans(P, A, P1),
    complement(A, B),
    trans(Q, B, Q1).
Optimizations: Clause Resolution Factoring

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

Clause Resolution Factoring

```prolog
:- table trans/3.
trans(Pname, A, Q) :- Pname := Pexp, trans(Pexp, A, Q).
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(P1 # P2, A, Q) :- trans(P1, A, Q); trans(P2, A, Q).
```

Optimizations: Specialization based on modes

```prolog
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))).
```

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.
Value Passing (contd.)

Example:

\[
\text{channel}(N, B) ::= \\
\quad \text{length}(B, L) o \\
\quad \text{if}(L == 0) \\
\quad \quad , \text{read_only}(N, B) \\
\quad \quad , \text{if}(L == N) \\
\quad \quad , \text{write_only}(N, B) \\
\quad \quad , \text{read_only}(N, B) \# \text{write_only}(N, B) \\
\).
\]

\[
\text{write_only}(N, [X|N]) ::= \text{out}(X)) o \text{channel}(N, R).
\]

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)

<table>
<thead>
<tr>
<th>Program</th>
<th>System</th>
<th>Time (sec)</th>
<th>Space (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader5</td>
<td>SPIN</td>
<td>8.1</td>
<td>9.60</td>
</tr>
<tr>
<td></td>
<td>XMC</td>
<td>5.5</td>
<td>0.78</td>
</tr>
<tr>
<td>sieve6</td>
<td>SPIN</td>
<td>1.8</td>
<td>2.31</td>
</tr>
<tr>
<td></td>
<td>XMC</td>
<td>10.4</td>
<td>1.23</td>
</tr>
</tbody>
</table>

A More Challenging Example

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Tool</th>
<th>Completed?</th>
<th>Memory (MB)</th>
<th>Time (min:sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>W=1 *fixed</td>
<td>SPIN</td>
<td>Yes</td>
<td>749</td>
<td>0:10</td>
</tr>
<tr>
<td></td>
<td>XMC</td>
<td>Yes</td>
<td>18.4</td>
<td>0:03</td>
</tr>
<tr>
<td>W=1 *fixed</td>
<td>SPIN</td>
<td>Yes</td>
<td>820</td>
<td>1:02</td>
</tr>
<tr>
<td></td>
<td>XMC</td>
<td>Yes</td>
<td>128</td>
<td>0:46</td>
</tr>
<tr>
<td>W=2 *fixed</td>
<td>SPIN</td>
<td>Yes</td>
<td>751</td>
<td>0:12</td>
</tr>
<tr>
<td></td>
<td>XMC</td>
<td>Yes</td>
<td>68</td>
<td>0:11</td>
</tr>
<tr>
<td>W=2 *fixed</td>
<td>SPIN</td>
<td>Yes</td>
<td>1789</td>
<td>6:23</td>
</tr>
<tr>
<td></td>
<td>XMC</td>
<td>Yes</td>
<td>688</td>
<td>3:48</td>
</tr>
</tbody>
</table>
Alternating Fixed Points in XMC: Performance

Process $M_k$:

- Formula $F$: $\nu X, \mu Y.([\nu X.((a) X) \land X] \lor Y)$

<table>
<thead>
<tr>
<th>Instance</th>
<th>CMC</th>
<th>FAM</th>
<th>XMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{500} \parallel F$</td>
<td>33.84</td>
<td>2.88</td>
<td>1.61</td>
</tr>
<tr>
<td>$M_{1000} \parallel F$</td>
<td>138.51</td>
<td>11.64</td>
<td>2.76</td>
</tr>
<tr>
<td>$M_{1500} \parallel F$</td>
<td>312.10</td>
<td>26.61</td>
<td>4.08</td>
</tr>
</tbody>
</table>

Synchronous CCS

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

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

  $E \rightarrow E \quad \alpha.E \rightarrow E$

  $E \rightarrow E' \quad E + F \rightarrow F' (\alpha \neq \pi) \quad F \rightarrow F'$

  $E \rightarrow E', F \rightarrow F' \quad E \times F \rightarrow E' \times F' \quad P \rightarrow P'$

  $A \rightarrow B (A \neq B, \alpha \neq \pi)$

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:
  
  $^* \times ^* = ^*$

  $\alpha \times \overline{\alpha} = ^*$

- The algebraic laws are applied only when actions are compared, as in $\alpha.E \rightarrow E$ 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:

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

Inference rules for Compositional Model Checking

Sample rules:

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

\[
\frac{(P_1 \setminus L) \mid P_2 \vdash [\alpha]F}{(P_1[f_L] \mid P_2)[f_1(L)] \vdash [\alpha]F} \quad f_L \text{ renames elements of } L \text{ to new names}
\]

\[
\frac{(\beta.P_1)[f] \mid P_2 \vdash [\alpha]F}{(f(\beta).P_1)[f] \mid P_2 \vdash [\alpha]F} \quad \beta.P_1 \mid P_2 \vdash [\alpha]F \quad \beta \neq \alpha = \tau
\]

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.

Outline

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

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?

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 scheme:

- 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.
Prospects and Limitations

- Combines model checking and induction
  without compromising model checking speeds
- Searches for induction proofs whose structure is identical to the
cursive 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

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

Real-time systems

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

Local Model Checking of real-time systems

Ongoing work.

- Add one more inference rule to the model checker:

  \[
  \frac{R \models F}{\text{refine } R_i, \forall i \vdash R_i \models F}
  \]

- \text{refine} 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 \text{refine} is mutually recursive with \text{arc}.
Outline

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

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

Future Directions: Ease of Use

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

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.
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