

Model Checking Based on Logic Programming

C.R. Ramakrishnan

SUNY, Stony Brook

GULP Summer School

Sept. '98

Model Checking

Technique used to verify properties of system specifications.

- Concurrent systems are specified as automata or using various process calculi (e.g., CCS).

```
data_channel = in?Dat . data_channel
              + drop! . data_channel
```

- Properties of interest are expressed using temporal logics (e.g., CTL).

EF drop_enabled

Temporal logic formulas typically involve fixed point operators for specifying properties over infinite execution sequences.

Logic Programming-Based Model Checking

Harness the power of (tabled) logic programming for model checking.

- Semantics of process languages and property logics can be succinctly specified as Horn clauses.
- Complete LP evaluation techniques (bottom-up, OLDT) can be used to derive implementations directly from these specifications.
- Logic programming/deductive database optimizations can be used to transform these specifications for efficient evaluation.
- Power of Constraints and Inference can be exploited to design systems for verifying properties of infinite-state systems.

Outline

▷ Introduction to Model Checking

Temporal and Modal Logics

- Introduction to Tabled LP & XSB
OLDT & SLG resolution, implementation
- Encoding Model Checkers as Logic Programs
CTL & modal mu-calculus model checkers
- Current & Future Research
Compositional model checkers, verification of infinite-state systems

Model Checking

- A concurrent system is viewed as a parallel composition of (sequential) processes. Each process is a (finite state) automaton.
 - ▷ The set of *global* states of the system is a subset of the product of all *local* states of each process.
- Temporal logic formulas are used to express specific properties of the system's execution behavior. *e.g.*,
 - ▷ Properties that must hold at all states of the system: *e.g.*, freedom from deadlocks.
 - ▷ Properties that must hold at some state on every execution path: *e.g.*, losslessness of buffered channels.

Process Specifications

- Systems specified using various formalisms, including CCS (Calculus of Communicating Systems).
- Processes perform basic operations, such as communicating over a port (CCS [Milner '89]), or setting a shared variable (CSP [Hoare '85]).
- 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 [Manna&Pnueli'91, '95]:
- 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 [Kozen '83]).
$$\mu X.(\text{bit32.carry_out})\#t \vee [\text{bit32.carry_out}].X$$

Fundamentals of Formal Methods

Specification Logics

Recommended reading: Allen Emerson's chapter "Temporal and Modal Logic" in Handbook of TCS v. B (J. van Leeuwen, ed.)

Process Algebras

Recommended reading: Robin Milner's book, "Communication and Concurrency".

Model Checking

Recommended reading: Manna&Pnueli's books, Ed Clarke's upcoming text.

Online Resources

Starting Points:

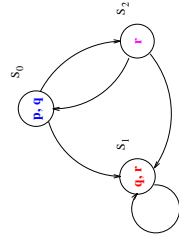
- Oxford University's Formal Methods Archive
<http://www.comlab.ox.ac.uk/archive/formal-methods.html>
- FM Educational Resources
<http://www.cs.indiana.edu/formal-methods-education/>
- Formal Methods Europe
<http://www.csr.nc1.ac.uk/projects/FME/index.html>
- SRI's FM Page
<http://www.csl.sri.com/fm.html>

Introduction to Model Checking

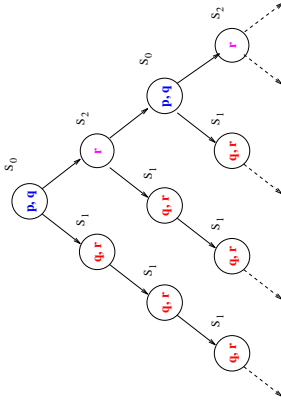
- Overview
- Modeling Reactive systems
 ω -automata
- Specification Logics:
 LTL, CTL, CTL*
- Process Calculi:
 CCS
- Model Checking:
 - CTL model checking
 - LTL model checking
 - Compositional proofs

Computation Trees

Automaton

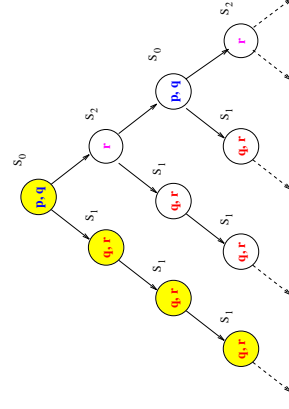


Computation Tree

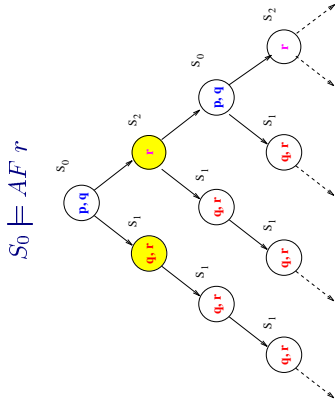


Temporal Logic Model Checking - I

$$S_0 \models EGq$$



Temporal Logic Model Checking - II



Büchi Automata

Nondeterministic, Finite State Automata on *infinite inputs*.
 Acceptance condition: designated (“good”) states are visited infinitely often.

Proposition: Emptiness checking of Büchi automata is in P.

Rationale: States that can be visited infinitely often will be part of some SCC.

Properties of Büchi Automata

Recognizers for ω -regular languages.

Closed under

- Union
- Intersection
- Complementation (difficult).

There are nondeterministic automata which recognize languages that cannot be recognized by deterministic automata.

Further reading: W. Thomas’s chapter “Automata on Infinite Objects” in the Handbook of TCS (v. B).

Temporal Logics

Linear-time: At each moment, there is only one future moment.

- Linear-time Temporal Logic (LTL) [Pnueli'77]

Branching-time: Time has tree-like structure.

At each moment, time splits into alternate courses representing different possible futures.

- Computation Tree Logic (CTL) [Clarke/Emerson'81]
- CTL* [Emerson/Halpern'86]

LTL

Propositional Linear Temporal Logic

- p : base propositions
- $\phi_1 \vee \phi_2, \phi_1 \wedge \phi_2, \neg\phi$
- $F\phi$: Eventually ϕ (also written as $\langle\phi\rangle$)
- $G\phi$: Henceforth ϕ (also written as $\Box\phi$)
- $X\phi$: Nexttime ϕ
- $\phi_1 U \phi_2$: ϕ_1 until ϕ_2
- $\phi_1 B \phi_2$: ϕ_1 before ϕ_2

Some Properties of LTL Operators

Equivalences:

- $F\phi \equiv \text{true } U \phi$
- $G\phi \equiv \neg F\neg\phi$
- $\phi_1 B \phi_2 \equiv \neg((\neg\phi_1) U \phi_2)$

Examples:

- FGp : Eventually Always p .
The system eventually reaches a stable state
- GFp : Always Eventually p .
Infinitely often p

LTL: Semantics

Notations:

- P : set of propositions
- ϕ : LTL formula over P
- Model M of formula ϕ : sequences over 2^P .
- $M(i)$: i -th element of M

Some of the description is from Mukund's tutorial on LTL model checking; see <http://www.smi.ernet.in/~mukund>

LTL: Semantics (contd.)

$M, i \models \phi$: ϕ is true at the time instant i in model M .

1. $M, i \models p$ iff $p \in M(i)$, for atomic proposition p
2. $M, i \models \neg\phi$ iff $M, i \not\models \phi$
3. $M, i \models \phi_1 \vee \phi_2$ iff $M \models \phi_1$ or $M \models \phi_2$
4. $M, i \models X\phi_1$ iff $M, i+1 \models \phi_1$
5. $M, i \models \phi_1 U \phi_2$ iff $\exists k > i$ s.t.
 - $\forall i \leq j < k$ $M, j \models \phi_1$, and
 - $M, k \models \phi_2$

LTL and Büchi Automata

Check satisfiability of LTL formulas by verifying the emptiness of the corresponding Büchi automata.

Straightforward translation (a la regular expressions to finite automata) is inefficient:

Complementation of Büchi automaton has exponential lower bound.

Vardi-Wolper construction: [Vardi/Wolper '86] A more practical translation (still exponential).

Automata-Theoretic Approach to Model Checking

- System specifications treated as finite automata, S .
 - Fairness constraints modelled as Büchi acceptance conditions.
- Negation of property to be verified translated into a “witness” automaton, F .
- Check for emptiness of $S \cap F$.

Branching Time Logics

CTL*

State Formulas: which are true in specific states

- Base propositions
- Negation, conjunction of state formulas
- $E\psi$ where ψ is a path formula

Path Formulas: which are true along specific paths

- State formulas
- Negation, conjunction of path formulas
- $X\psi, \psi_1 U \psi_2$, where ψ 's are path formulas

Semantics of CTL*

Model of a CTL* formula is a computation tree.

- $M, s \models \phi_1 \vee \phi_2$ iff $M, s \models \phi_1$ or $M, s \models \phi_2$.
- $M, s \models E\psi$ iff there is a path π starting at s such that $M, \pi \models \psi$.
- $M, \pi \models \phi$ iff $M, s \models \phi$, where s is the first state in π .
- $M, \pi \models X\psi$ iff $M, \pi^1 \models \psi$
- ...

CTL

- CTL is a sublogic of CTL*
where X, U (and their duals) must be immediately preceded by E (or its dual).
- LTL is a sublogic of CTL*
- CTL and LTL have different expressive powers, but strictly less expressive than CTL*.
e.g., $A(FGp) \vee AG(EFp)$

Modal Mu-Calculus

- Expressive logic that can be used to encode a variety of temporal logics.
“Assembly language” of temporal logics
- Explicit fixed point operators and nesting of fixed points makes the language powerful, at the same time, difficult to read.
- $EFp \equiv \mu X.p \vee \langle - \rangle X$
- $EGp \equiv \nu Y.p \wedge \langle - \rangle Y$

Syntax of Modal Mu-calculus

- formula variables X_1, X_2, \dots
- atomic propositions p_1, p_2, \dots
- negation, conjunction and disjunction of formulae
- Existential modality: $\langle a \rangle f$
- Universal modality: $[a]f$
- Least fixed point: $\mu X.f(X)$
- Greatest fixed point: $\nu X.f(X)$

Semantics of Modal Mu-Calculus

Model of a formula is a set of states in a Kripke structure, M .
An *environment*, e , assigns a set of *states* to a formula variable.

- $\llbracket p \rrbracket_{Me} =$ set of states of M where p is true
- $\llbracket X \rrbracket_{Me} = e(X)$
- $\llbracket f \wedge g \rrbracket_{Me} = \llbracket f \rrbracket_{Me} \cap \llbracket g \rrbracket_{Me}$
- $\llbracket f \vee g \rrbracket_{Me} = \llbracket f \rrbracket_{Me} \cup \llbracket g \rrbracket_{Me}$

Semantics of Modal Mu-Calculus

- $\llbracket \langle a \rangle f \rrbracket_{M \varepsilon} = \{s \mid \exists s' \text{ s.t. } s \xrightarrow{a} s', s' \in \llbracket f \rrbracket_{M \varepsilon}\}$
 - $\llbracket [a]f \rrbracket_{M \varepsilon} = \{s \mid \forall s', s \xrightarrow{a} s' \implies s' \in \llbracket f \rrbracket_{M \varepsilon}\}$
 - $\llbracket \mu X.f \rrbracket_{M \varepsilon}$ is the least fixed point of the functional F :
- $$F(S) = \llbracket f \rrbracket_{M \varepsilon}(X \longleftarrow S)$$
- $\llbracket \nu X.f \rrbracket_{M \varepsilon}$ is the greatest fixed point of the functional F :
- $$F(S) = \llbracket f \rrbracket_{M \varepsilon}(X \longleftarrow S)$$

Flavors of Modal Mu-Calculus

- $\mu X. \langle - \rangle X \vee \langle a \rangle tt$:
Single fixed point (eventually “ a ” is enabled)
- $\mu X. \langle - \rangle X \vee (\nu Y. \langle \tau \rangle Y)$:
Nested, but non-alternating fixed point (eventually, always τ is enabled).
- $\nu X. \mu Y. ([-]. \langle a \rangle \# \wedge X) \vee Y$:
Alternating fixed point (alternation depth-2).

CTL formulas translate into nested but non-alternating fixed points
CTL* and LTL formulas can always be translated to mu-calculus formulas with alternation depth ≤ 2 .

Temporal Logic Model Checkers

- **SPIN**: (Lucent)
LTL formulas, Promela process description language.
Depth-first search, Bitstate hashing, partial order methods.
- **SMV**: (CMU)
CTL formulas, state machine language.
Symbolic model checker (BDDs).
- **Concurrency Workbench**: (Edinburgh/NCSU)
alternation-free modal mu-calculus, CCS.
Semantic state-minimization techniques.
- **Concurrency Factory**: (Stony Brook/NCSU)
full modal mu-calculus, value passing CCS.

Temporal Logic Model Checkers (cont'd.)

- **Mur ϕ** : (Stanford)
guarded rules for specifying systems, subset of LTL.
- **Kronos & HyTech**: (Berkeley)
real-time and hybrid systems.
and a host of others.

Behavior Conformance Checkers

- **COSPAN:** (Lucent)
Equivalence between omega automata.
- **FDR:** (Oxford)
Refinement between CSP programs.
- **Concurrency Workbench:** (Edinburgh/NCSU)
Refinement between CCS programs; Observational equivalence.

Combination Checkers

- **PVS:** (SRI)
Theorem prover, with a mu-calculus model checker as a decision procedure.
- **STeP:** (Stanford)
Deductive model checking.
Tableau-based methods, constraints, equational reasoning.

Logic Programming-Based Model Checking

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.

XMC Implementation

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

Outline

- Introduction to Model Checking
Temporal and Modal Logics
- ▷ **Introduction to Tabled LP & XSB**
OLDT & SLG resolution, implementation
- Encoding Model Checkers as Logic Programs
CTL & modal mu-calculus model checkers
- Current & Future Research
Compositional model checkers, verification of infinite-state systems

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*.

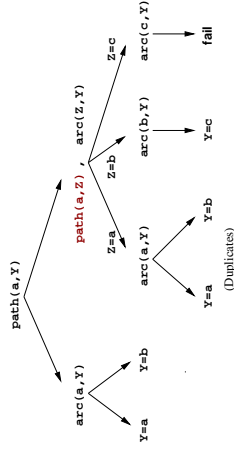
On encountering a goal G ,

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



Calls
path(a,y)

Answers
path(a,a)
path(a,b)
path(a,c)

Tabled Resolution: An Operational Overview

- Each answer table is *produced* by one computation unit (producer).
- Each answer table has one or more *consumers*.
- Derivations form a proof *forest*, with each tree in the forest corresponding to an answer table.
- Each step of resolution modifies the proof forest using one of five tabling operations.

Tabling Operations

Program Clause Resolution: Extend selected proof tree using one step of OLD-resolution.

New Subgoal: Create new consumer; if needed create new producer for the subgoal.

New Answer: Place answer computed by resolution in the corresponding goal's answer table.

Answer Clause Resolution: Given a goal, an answer table and a set of answers that have already been consumed from that table, resolve the goal against any new answers in the table.

Completion: Remove consumers when all answers are deemed to have been consumed (using SCC check).

Tabled Resolution: Negation

Programs with negation computed using “conditional answers”.
Selected literal is $\neg Q$.

- If there is no table for Q , create one, and start proof tree for Q
- If answer table for Q exists, which
 - holds an answer, fail out of $\neg Q$.
 - holds no answers, and is marked “complete”, then succeed out of $\neg Q$.
 - holds no answers, but is still not marked as complete, *delay* Q ; continue with next literal selection.

Tabled Resolution: Simplification

- New answer is computed for Q .
Delete all answers with $\neg Q$ in delay set.
- Answer table for Q is completed with no answers for Q .
Delete $\neg Q$ from all delay sets.

Tabulation in XSB

- Tables maintained using *trie*-like data structures.
- Search in proof forest done by “freezing” portions of traversal (abstract machine) stacks.
- Scheduling strategies (which control how computation among various proof trees are interleaved) ensure that no *delays* are done for (dynamically) stratified programs.
- If no 2-valued minimal model exists, the delayed literals can be used to glean the relationships between literals with *unknown* values (under well founded semantics).

Tabled Resolution: Sources of Information

- OLD T resolution: Tamaki/Sato '86.
- SLG resolution: Chen/Warren '96.
- SLG WAM abstract machine: Swift/Warren '94
- Tabulation as Program Transformation: Degerstedt's thesis, '96.
- Warren's talks on Tabled Logic Programming
<http://www.cs.sunysb.edu/~warren>

XSB Tabled Logic Programming System

- Prolog performance comparable with state-of-the-art interpreted 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

- Introduction to Model Checking
Temporal and Modal Logics
- Introduction to Tabled LP & XSB
OLDT & SLG resolution, implementation
- ▷ **Encoding Model Checkers as Logic Programs**
CTL & modal mu-calculus model checkers
- Current & Future Research
Compositional model checkers, verification of infinite-state systems

CTL Model Checker as a Logic Program

$\text{trans}(S1, A, S2)$: transition relation (from system specifications)

$\text{models}(S, F)$: Does system state S model formula F ?

$\text{models}(S, \text{ef}(F))$:- $\text{models}(S, F)$.

$\text{models}(S, \text{ef}(F))$:- $\text{trans}(S, -, T), \text{models}(T, \text{ef}(F))$.

$\text{models}(S, \text{af}(F))$:- $\text{models}(S, F)$.

$\text{models}(S, \text{af}(F))$:- $\text{findall}(T, \text{trans}(S, -, T), \text{LS}),$
 $\text{all_models}(\text{LS}, \text{af}(F))$.

$\text{models}(S, \text{eg}(F))$:- $\text{negate}(F, \text{NF}), \text{not models}(S, \text{af}(\text{NF}))$

$\text{models}(S, \text{ag}(F))$:- $\text{negate}(F, \text{NF}), \text{not models}(S, \text{ef}(\text{NF}))$

Modal Mu-calculus: Syntax

```
Fexp --> Fframe
| tt
| Fexp ^ Fexp
| Fexp v Fexp
| diam(A, Fexp)
| diamMinus(A, Fexp)
| box(A, Fexp)
| boxMinus(A, Fexp)
```

Definition -->

```
Fframe == Fexp
| Fframe += Fexp
```

An Example: *deadlock freedom*

```
df == boxMinus(nil, df) ^ diamMinus(nil, tt)
```

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).
```

Modal Mu-Calculus: Semantics (contd.)

```
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).  
  
map_models([], _).  
map_models([S|Ss], F) :- models(S, F), map_models(Ss, F).
```

Fixed Points

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

```
models(S, Fname) :-  
  def(Fname, mu(Fexp)),  
  models(S, Fexp).
```

Greatest fixed points can be computed using $\mu(F) \equiv \neg \mu(\neg F)$.

```
models(S, Fname) :-  
  def(Fname, nu(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 '.

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.

Syntax of CCS

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

Example:

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

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

Encoding the Semantics of CCS

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

Prefix $trans(A \circ P, A, P)$.

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

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

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

Semantics of CCS

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

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

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

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

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

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

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

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

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

Semantics of CCS (contd.)

Parallel $trans(P | Q, A, P1 | Q) :- trans(P, A, P1).$
 composition $trans(P | Q, A, P | Q1) :- trans(Q, A, Q1).$
 $trans(P | Q, \tau, P1 | Q1) :-$

$trans(P, A, P1),$
 $trans(Q, B, Q1),$
 $complement(A, B).$

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

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

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.

See Dawson et al '95.

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).
```

Optimizations: Multi-version code

```
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)).
```

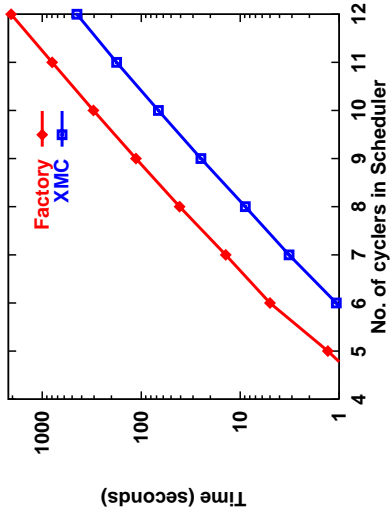
Performance of XMC

Time as well as space performance of XMC is:

- Comparable to Concurrency Factory for basic CCS.
- Two orders of magnitude better than the Concurrency Factory for value passing CCS.
- Competitive with SPIN (without partial order reduction).

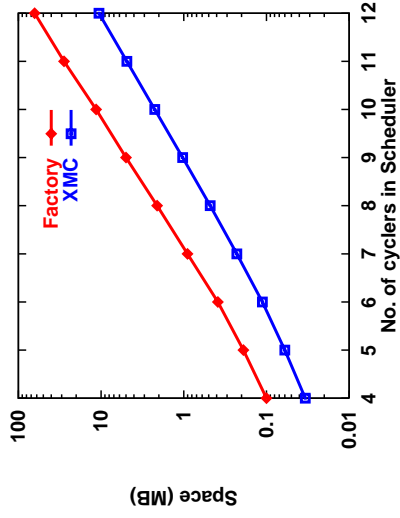
Performance of XMC (Time)

XMC and Concurrency Factory on Milner's Scheduler of Cyclers (basic CCS).



Performance of XMC (Space)

XMC and Concurrency Factory on Milner's Scheduler of Cyclers.



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.

Example:

```
channel ::= in(get(X)) o out(put(X)) o channel.  
max(X) ::= in(get(Y)) o if(X < Y,  
           out(put(Y)) o max(Y),  
           max(X)).
```

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.

Model Checking in XSB: A Complete Example

Specification of Alternating Bit Protocol (with a bug):

```
dat_chan ::= in(sRin(Dat)) o
(out(sRout(Dat)) # out(drop)) o dat_chan.

ack_chan ::= in(rSin) o
(out(rSout) # out(drop)) o ack_chan.

ctr ::= in(up) o (in(down) # in(up) o out(error)) o ctr.

psender(Seq) ::= out(send) o out(datOut(Seq)) o
(in(ackIn) o NSeq is 1-Seq o out(up) o psender(NSeq)
# out(timeout) o psender(Seq)).

sender ::= (out(up) o psender(0))
@ [datOut(X) / sRin(X), ackIn / rSout].
```

Specification of ABP (contd.)

```
precvr(Seq) ::= in(datIn(RecSeq)) o out(recv) o
if(RecSeq == Seq,
(out(down) o NSeq is 1-Seq o out(ackOut) o precvr(NSeq)),
out(ackOut) o precvr(Seq)).

recvr ::= precvr(0) @ [ datIn(X) / sRout(X), ackOut / rSin].

abp ::=
(sender | dat_chan | ack_chan | recvr | ctr)
\ {sRin(_), sRout(_), rSin, rSout, up, down}.
```

Model Checking in XSB (contd.)

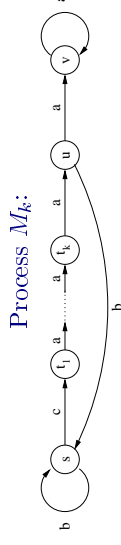
```
$ xsb
XSB v1.8.2, ...
| ?- [xmc].
[ccs loaded].
[umacculus loaded].
[xlc loaded].
| ?- xlc(abp).
[abp compiled].
| ?- ld(abp).
[abp.xlo dynamically loaded].
| ?- mck(abp, dropped_packet).
yes.
| ?-
```

Performance on Value Passing examples

- Nearly two orders of magnitude faster than Concurrency Factory.
- Factory's model checker first converts value passing into basic CCS.
- 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

Alternating Fixed Points in XMC: Performance



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

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".

Outline

- Introduction to Model Checking
Temporal and Modal Logics
- Introduction to Tabled LP & XSB
OLDT & SLG resolution, implementation
- Encoding Model Checkers as Logic Programs
CTL & modal mu-calculus model checkers
- ▷ **Current & Future Research**
Compositional model checkers, verification of infinite-state systems

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:

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

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

Inference rules for Compositional Model Checking

Sample rules:

$$\frac{(P_1 + P_2) \mid P_3 \vdash \langle \alpha \rangle F}{P_1 \mid P_3 \vdash \langle \alpha \rangle F \wedge P_2 \mid P_3 \vdash \langle \alpha \rangle F}$$

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

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

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

Implementation of Compositional Model Checking

- Single models predicate, with auxiliary definitions for
 - ▷ inventing new names (fL), 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).

Currently extending compositional proofs for value passing

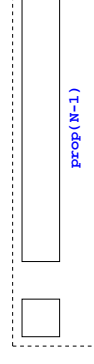
Beyond Finite-state Model Checking

- 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?

Induction



$\text{prop}(N)$

$\vdash \forall N \quad \Phi(N)$

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

$\text{phi}(0)$.

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

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:

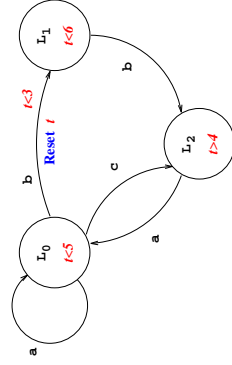
- Stop when induction hypothesis is reached
- ▷ Hypothesis \equiv theorem \Rightarrow hypothesis is reached when the current call, containing induction variables, is same as a subgoal previously encountered.
- ▷ Generate conditional answer instead of *failing*.
- Make induction structure explicit by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

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

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

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

Local Model Checking of real-time systems

Ongoing work.

- Add one more inference rule to the model checker:
$$\frac{R \dashv F}{R \text{ refines } \{R_i\}, \forall i \quad R_i \dashv F}$$
- **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**.

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 language extensions for
 - State encapsulation
 - Procedural code for sequential machines
 - Support for types
 - Static checks for errors in specifications
- Justification for proofs
Help isolate problem in specification

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.

Summary

Logic Programming with Tabling is an ideal platform for implementing model checkers.

- High-level implementation \Rightarrow enables development of model checkers for a wide variety of process languages and temporal logics, using different techniques.
- Horn clauses viewed as inference rules \Rightarrow facilitates combination of model checking and deduction.
- Constraints and Tabling enables verification of infinite-state systems
- High-level implementation of model checking \Rightarrow high-level tools for debugging the process specifications.