The Maude LTL Model Checker and its Implementation

Steven Eker, SRI

José Meseguer and Ambarish Sridharanarayanan, UIUC

Introduction (I)

Maude is a executable specification language based on term rewriting (semantics given by rewriting logic).

Equations are assumed to be confluent and terminating and are used to define algebraic data types.

Rewrite rules need not be terminating or confluent and naturally express concurrency and nondeterminism (need to satisfy coherence however).

Execution of rewrite rules generates rewrite graphs where states are terms are reduced (i.e. in normal form) w.r.t. the equations.

Introduction (II)

The Maude model checker provides LTL model checking for finite rewrite graphs.

Parameterized families of state predicates are defined by Maude equations.

Rewriting takes place modulo the axioms of associativity, commutativity and identity for chosen functions symbols, generalizing list, set and multiset rewriting.

Expressiveness of term rewriting extends model checking beyond traditional domains such as hardware and communication protocols to areas such nested and/or mobile processes and discrete cell biology models.

Allows deep embedding of other languages.

Dekker (I)

```
fmod MEMORY is inc INT . inc QID .
  sorts Memory .
  op none : -> Memory .
  op __ : Memory Memory -> Memory [assoc comm id: none] .
  op [_,_] : Qid Int -> Memory .
endfm

fmod TESTS is inc MEMORY .
  sort Test .
  op _=_ : Qid Int -> Test .
  op eval : Test Memory -> Bool .

  var Q : Qid .  var M : Memory .  vars N N' : Int .
  eq eval(Q = N, [Q, N'] M) = N == N' .
endfm
```

Dekker (II)

```
fmod SEQUENTIAL is inc TESTS .
  sorts UserStatement Program .
  subsort UserStatement < Program .
  op skip : -> Program .
  op _;_ : Program Program -> Program
    [prec 61 assoc id: skip] .
  op _:=_ : Qid Int -> Program .
  op if_then_fi : Test Program -> Program .
  op while_do_od : Test Program -> Program .
  op repeat_forever : Program -> Program .
endfm
```

Dekker (III)

```
mod PARALLEL is inc SEQUENTIAL .
  sorts Pid Process Soup MachineState .
  subsort Process < Soup .
  op [_,_] : Pid Program -> Process .
  op empty : -> Soup .
  op _|_ : Soup Soup -> Soup
    [prec 61 assoc comm id: empty] .
  op {_,_,_} : Soup Memory Pid -> MachineState .

  vars P R : Program .  var S : Soup .
  var U : UserStatement .  vars I J : Pid .
  var M : Memory . var Q : Qid .
  vars N X : Int . var T : Test .
```

Dekker (IV)

```
rl {[I, U ; R] | S, M, J} => {[I, R] | S, M, I} .

rl {[I, (Q := N) ; R] | S, [Q, X] M, J} =>
  {[I, R] | S, [Q, N] M, I} .

rl {[I, if T then P fi ; R] | S, M, J} =>
  {[I, if eval(T, M) then P else skip fi ; R] |
    S, M, I} .

rl {[I, while T do P od ; R] | S, M, J} =>
  {[I, if eval(T, M) then (P ; while T do P od) else
       skip fi ; R] | S, M, I} .

rl {[I, repeat P forever ; R] | S, M, J} =>
  {[I, P ; repeat P forever ; R] | S, M, I} .
```

Dekker (V)

```
mod DEKKER is inc PARALLEL .
  subsort Int < Pid .
  ops crit rem : -> UserStatement .
  ops p1 p2 : -> Program .
  op initialMem : -> Memory .
  op initial : -> MachineState .
  eq initialMem = ['c1, 1] ['c2, 1] ['turn, 1] .
  eq initial = { [1, p1] | [2, p2], initialMem, 0 } .
```

Dekker (VI)

```
eq p1 =
  repeat
    'c1 := 0 ;
    while 'c2 = 0 do
      if 'turn = 2 then
        'c1 := 1 ;
        while 'turn = 2 do skip od ;
        'c1 := 0
      fi
    od ;
    crit ;
    'turn := 2 ; 'c1 := 1 ;
    rem
  forever .
```

Dekker (VII)

```
mod CHECK is inc DEKKER .  inc MODEL-CHECKER .
  subsort MachineState < State .
  ops enterCrit exec : Pid -> Prop .
  var M : Memory .  vars R : Program .
  var S : Soup .  vars I J : Pid .

  eq {[I, crit ; R] | S, M, J} |= enterCrit(I) = true .
  eq {S, M, J} |= exec(J) = true .
endm

red modelCheck(initial,
               [] ~ (enterCrit(1) /\ enterCrit(2))) .

red modelCheck(initial, ([]<> exec(1) /\ []<> exec(2)) ->
    ([]<> enterCrit(1) /\ []<> enterCrit(2))) .
```

Implementation

On-the-fly LTL model checking consists of two major steps.

1. Construct a Büchi automaton for the negation of the temporal logic formula that recognizes the language of counterexamples.
2. Lazily form the synchronous product of the Büchi automaton with the Kripke structure $\mathcal{K}(\mathcal{R}, State)$ associated to the rewrite theory $\mathcal{R}$, searching for an accepting cycle which is reachable from the initial state.

We make use of *Binary Decision Diagrams* (BDDs) to deal with pure propositional formulae used to label arcs in the various automata we use.

LTL Formula Preprocessing

- Put $\neg\phi$ in negative normal.
- Etessami and Holzman method using notions of *pure eventuality formulae* and *pure universality formulae*.
- Syntactic definitions that map neatly on to Maude's sort system.
- Method requires 8 unconditional equations; we add an extra equation:

  ```
      var pr : PureFormula .
      eq O pr = pr .
  ```

  where PureFormula is the intersection of PE-Formula and PU-Formula.
- Also use the rules from Somenzi and Bloem method that are not subsumed by Etessami and Holzman.

Büchi Automaton Construction

Modified Gastin and Oddoux's algorithm:

1. Construct a very weak alternating automaton V from ψ.
2. Remove unreachable states from V
3. Convert V into a generalized Büchi automaton G with multiple fairness conditions on arcs.
4. Iterate (combine parallel arcs, delete subsumed arcs, combine equivalent states).
5. SCC optimizations: delete dead components, simplify fairness conditions.
6. Repeat step (4).
7. Convert G into a regular Büchi automaton B.
8. Regular version of step (4).

Searching the Synchronous Product

Use double depth first method (Holzmann et al.). Store the Kripke structure as it is generated with extra 5 bit vectors per state.

- Propositions tested in state.
- Propositions true in state.
- Product pairs (with automaton states) seen by first DFS.
- Product pairs currently on first DFS stack.
- Product pairs seen by second DFS.

Synchronous product search code manipulates abtract state indices and is Maude-independent.

Performance Comparisons with SPIN (I)

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	42	250	1.58	4.74
10	43	570	1.58	5.58
50	821	1,920	13.11	26.6
100	4,443	14,200	104.86	226.3

Leader election - property 1

Performance Comparisons with SPIN (II)

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	40	250	1.58	4.73
10	49	580	1.58	5.54
50	825	1,880	13.11	26.6
100	5,928	13,800	104.86	223.7

Leader election - property 2

Performance Comparisons with SPIN (III)

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	46	270	1.58	4.73
10	50	560	1.68	5.58
50	1,804	2,310	22.94	44.46
100	10,709	19,480	104.86	330.3

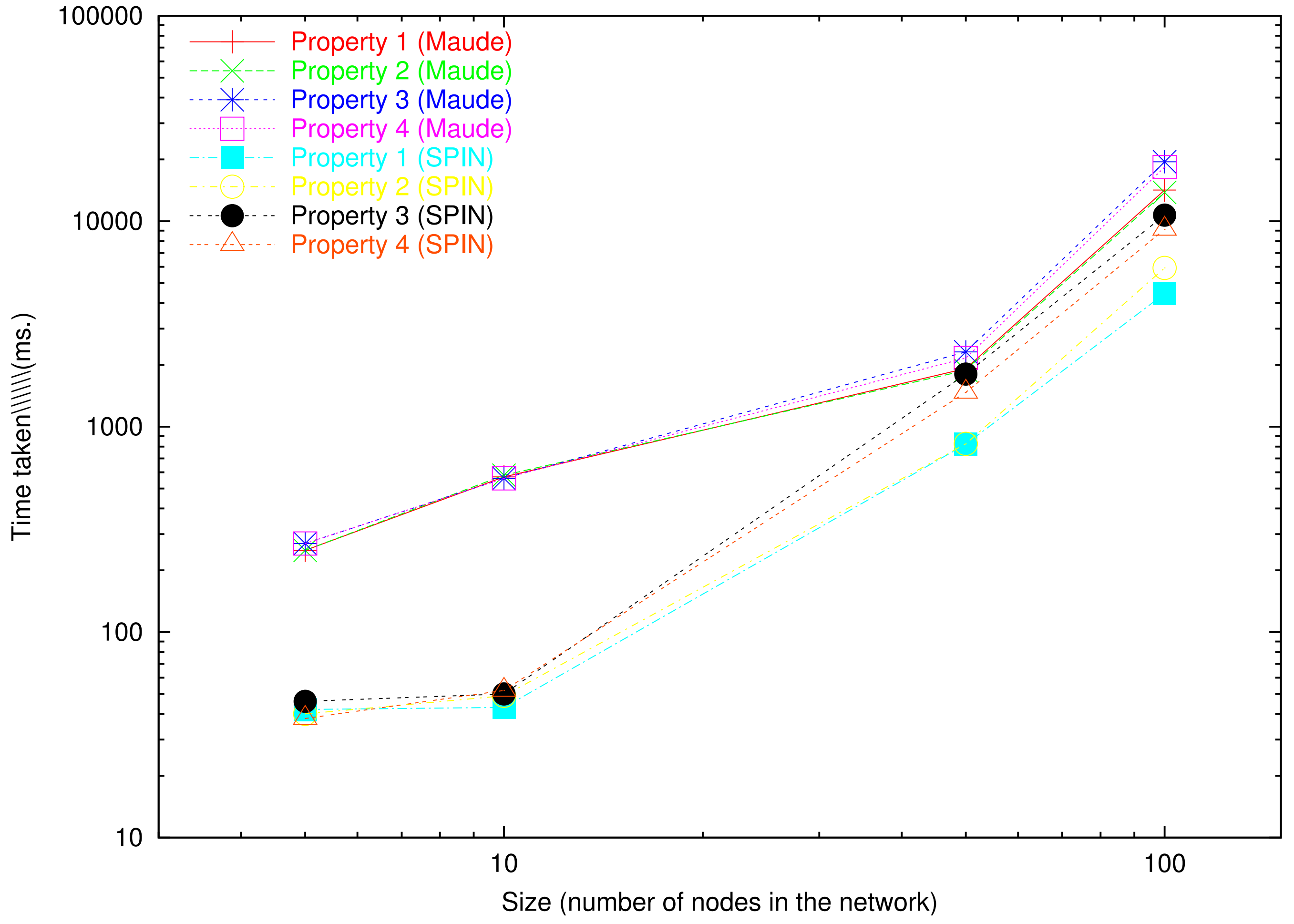
Leader election - property 3

Performance Comparisons with SPIN (IV)

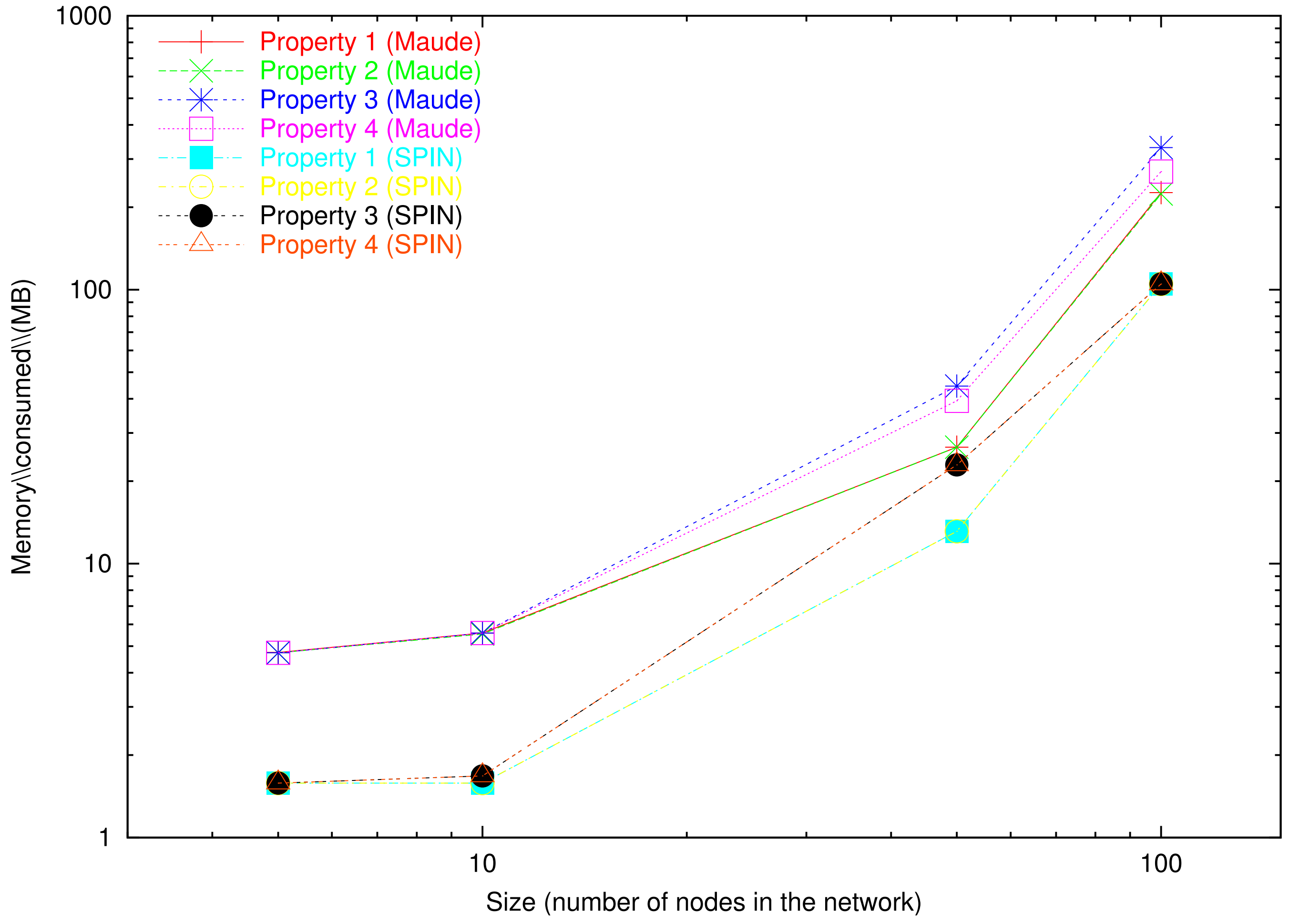
Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	38	270	1.58	4.73
10	52	560	1.68	5.58
50	1,468	2,160	22.94	39.3
100	9,134	18,320	104.86	270.1

Leader election - property 4

Performance Comparisons with SPIN (V)



Performance Comparisons with SPIN (VI)



Conclusions and Future Directions

The Maude model checker combines a very expressive system specification language with reasonable cpu/memory performance.

A number of research issues should be explored in the future, including:

- further improvements in the Büchi automata constructions;
- special treatment of fairness properties, instead of expressing them as LTL formulae;
- model checking of properties restricting the set of computation paths by means of suitable *strategy expressions*;
- development of general abstraction techniques for rewrite theories, and theorem proving support for proving such abstractions correct.