

The Maude LTL Model Checker

Steven Eker, SRI

José Meseguer and Ambarish Sridharanarayanan, UIUC

Introduction

A model checker typically supports two different levels of specification:

1. a **system specification** level, in which the concurrent system to be analyzed is formalized; and
2. a **property specification** level, in which the properties to be model checked—for example, temporal logic formulae—are specified.

Increasing the expressive power of system specification languages is important because many potential application areas—beyond traditional ones such as hardware and communication protocols—can be very hard to express in current model checking system specification languages.

Introduction (II)

For example, nested processes or “objects” are the natural way to specify both rewriting logic models of cell biology, and reflective distributed systems involving arbitrary nesting of metaobjects—the so-called “Russian dolls” model.

All this is easy to do in Maude, but seems hard to accomplish in other model checkers.

Furthermore, the Maude LTL checker can model check systems whose states involve data in **data types of infinite cardinality**; in fact, in **any algebraic data types**. The only assumption is that the set of states **reachable** from a given initial state is finite.

Introduction (III)

The main contribution of this work is to combine Maude's very expressive executable system specification language with an **explicit-state on-the-fly** linear temporal logic (LTL) model checker with time and space performance comparable to that of current high-performance model checkers of that kind such as SPIN.

The high expressive power at the system specification level has been achieved **without sacrificing performance** by taking into account the latest research developments in optimized Büchi automata constructions and in explicit-state model checking algorithms.

The Maude Model Checker

Given a rewrite theory \mathcal{R} specified in Maude by a system module, say `M`, and given an initial state, say `init` of sort `StateM`, we can **model check** different LTL properties beginning at this initial state by doing the following:

- defining a new module, say `CHECK-M`, that includes the modules `M` and `MODEL-CHECKER` as submodules;

The Maude Model Checker (II)

- giving a **subsort declaration**, `subsort StateM < State .`, where `State` is one of the key sorts in the module `MODEL-CHECKER` (this declaration can be omitted if `StateM = State`);
- defining the **syntax** of the **state predicates** we wish to use by means of constants and operators of sort `Prop` (a subsort of the sort `Formula` (i.e., LTL formulas) in the module `MODEL-CHECKER`); we can define **parameterless** state predicates as **constants** of sort `Prop`, and **parameterized** state predicates by operators from the sorts of their parameters to the `Prop` sort.

The Maude Model Checker (III)

- defining the **semantics** of the **state predicates** by means of equations involving the operator

`op _|=_ : State Prop -> Result [special ...] .`

in MODEL-CHECKER. The sort `Result` is a supersort of `Bool`. We define the semantics of each state predicate, say a parameterized state predicate `p`, by giving (possibly conditional) equations of the form:

`ceq exp1 |= p(u11,...,un1) = true if C1 .`

`...`

`ceq expk |= p(u1k,...,unk) = true if Ck .`

where:

- the `expi`, $1 \leq i \leq k$, are **patterns** of sort `StateM`, that is, terms, possibly with variables, and involving only

constructors, so that any of their instances by simplified ground terms cannot be further simplified;

- the terms $p(u_{1i}, \dots, u_{ni})$, $1 \leq i \leq k$ are likewise **patterns** of sort **Prop**;
- each condition C_i , $1 \leq i \leq k$, is a conjunction of equalities and memberships.

The Maude Model Checker (IV)

Note that **only the positive cases of state predicates have to be specified**; that is, if a state predicate **ground expression** of the form $\text{exp} \models p(w_1, \dots, w_k)$ cannot be simplified to **true**, then it is **assumed to be false**.

We are then ready, given an initial state **init**, to model check any LTL formula, say **form**; such LTL formulas are **ground terms** of sort **Formula** in **CHECK-M**; we do so by giving the Maude command,

```
red init |= form .
```

assuming, as already mentioned, that the set of reachable states is finite.

The Maude Model Checker (V)

Two things can then happen: if the property **form** holds we get the result **true**; if it doesn't, we get a counterexample, expressed with the syntax,

```
op counterExample : TransitionList TransitionList -> Result [ctor] .
```

This is because any counterexample to an LTL formula can be expressed as a **path of transitions followed by a cycle**.

The LTL Syntax

The model checker's LTL syntax is defined by the following functional module LTL imported by MODEL-CHECKER

```
fmod LTL is sort Formula .

  *** primitive LTL operators
  ops True False : -> Formula [ctor] .
  op ~_ : Formula -> Formula [ctor prec 53] .
  op _/\_ : Formula Formula -> Formula [comm ctor gather (E e) prec 55] .
  op _\/_ : Formula Formula -> Formula [comm ctor gather (E e) prec 59] .
  op 0_ : Formula -> Formula [ctor prec 53] .
  op _U_ : Formula Formula -> Formula [ctor prec 65] .
  op _R_ : Formula Formula -> Formula [ctor prec 65] .

  *** defined LTL operators
  op _->_ : Formula Formula -> Formula [gather (e E) prec 61] .
  op _<->_ : Formula Formula -> Formula [prec 61] .
  op <>_ : Formula -> Formula [prec 53] .
  op []_ : Formula -> Formula [prec 53] .
  op _W_ : Formula Formula -> Formula [prec 65] .
  op _|->_ : Formula Formula -> Formula [prec 65] . *** leads-to

  vars f g : Formula .
  eq f -> g = ~ f \/_ g .
```

eq $f \leftrightarrow g = (f \rightarrow g) \wedge (g \rightarrow f)$.

eq $\langle \rangle f = \text{True} \vee f$.

eq $[] f = \text{False} \wedge f$.

eq $f \vee g = (f \vee g) \wedge [] f$.

eq $f \rightarrow g = [](f \rightarrow (\langle \rangle g))$.

*** negative normal form

eq $\sim \text{True} = \text{False}$.

eq $\sim \text{False} = \text{True}$.

eq $\sim \sim f = f$.

eq $\sim (f \wedge g) = \sim f \vee \sim g$.

eq $\sim (f \vee g) = \sim f \wedge \sim g$.

eq $\sim 0 f = 0 \sim f$.

eq $\sim(f \vee g) = (\sim f) \wedge (\sim g)$.

eq $\sim(f \wedge g) = (\sim f) \vee (\sim g)$.

endfm

The LTL Syntax (II)

Note that the equations in this module do two things:

- express all defined LTL operators in terms of the basic operators **True**, **False**, negation, conjunction, disjunction, next, **O**, until, **U**, and release, **R**
- transform the LTL formula using only those basic operators into an equivalent one in **negative normal form**, that is, the negations are **pushed all the way down into the state predicates**

Dekker's Mutex Algorithm

One of the earliest correct solutions to the mutual exclusion problem was given by Dekker with his algorithm. The algorithm assumes processes that execute concurrently on a shared memory machine and communicate with each other through shared variables.

There are two processes, $p1$ and $p2$. Process 1 sets a Boolean variable $c1$ to 1 to indicate that it wishes to enter its critical section. Process $p2$ does the same with variable $c2$. If one process, after setting its variable to 1 finds that the variable of its competitor is 0, then it enters its critical section rightaway. In case of a tie (both variables set to 1) the tie is broken using a variable $turn$ that takes values in $\{1, 2\}$.

Dekker's Mutex Algorithm (II)

The code of process 1 is as follows,

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

Dekker's Mutex Algorithm (III)

The code of process 2 is entirely symmetric:

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


The Semantics of Dekker's Algorithm

To subject Dekker's algorithm to a model checking analysis we first need somehow to specify precisely the **semantics** of the parallel language in which it is written.

This can be done by **specifying such a semantics as a rewrite theory** in Maude for a simple parallel language expressive enough to write Dekker's algorithm in it.

A model of the memory and the syntax of the programs used by the processes is defined in functional modules **MEMORY**, **TESTS**, and **SEQUENTIAL**. The rewriting semantics of the language is then defined in the **PARALLEL** module. The **DEKKER** module defines the algorithm and the state predicates (see Appendix A).

Model Checking Dekker's Algorithm

We need to define two state predicates parameterized by the process id: `enterCrit`, when the process is about to enter its critical section, and `exec`, when the process has just executed.

```
mod CHECK is inc DEKKER . inc MODEL-CHECKER .  
  inc LTL-SIMPLIFIER . *** optional  
  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
```

Model Checking Dekker's Algorithm (II)

The **mutual exclusion property** is satisfied:

```
=====
reduce in CHECK : initial |= []~ (enterCrit(1) /\ enterCrit(2)) .
ModelChecker: Property automaton has 2 states.
ModelSymbol: Examined 245 system states.
rewrites: 1052 in 30ms cpu (30ms real) (35066 rewrites/second)
result Bool: true
=====
```

Model Checking Dekker's Algorithm (III)

But the **strong liveness property** that executing infinitely often implies entering one's critical section infinitely often fails, as witnessed by the counterexample (fragment)

```
reduce in CHECK : initial |= []<> exec(1) -> []<> enterCrit(1) .
```

```
ModelChecker: Property automaton has 3 states.
```

```
ModelSymbol: Examined 16 system states.
```

```
rewrites: 148 in 10ms cpu (10ms real) (14800 rewrites/second)
```

```
result Result: counterExample(
```

```
{[1,repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn  
  = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ;  
  'turn := 2 ; 'c1 := 0 ; rem forever] | [2,repeat 'c2 := 1 ; while 'c1 = 1  
do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od  
; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],[ 'c1,0] [ 'c2,0] [ 'turn,1],0},  
unlabeled}
```

```
...
```

Model Checking Dekker's Algorithm (IV)

However, the **weaker liveness property** that if **both** p1 and p2 execute infinitely often then both enter their critical sections infinitely often is true:

=====

```
reduce in CHECK : initial |= []<> exec(1) /\ []<> exec(2) -> []<> enterCrit(1)
    /\ []<> enterCrit(2) .
```

ModelChecker: Property automaton has 7 states.

ModelSymbol: Examined 245 system states.

rewrites: 1502 in 60ms cpu (60ms real) (25033 rewrites/second)

result Bool: true

Sat Solver

A formula $\varphi \in LTL(AP)$ is **satisfiable** iff there is a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ with $L : A \longrightarrow \mathcal{P}(AP)$, a state $a \in A$, and a computation path π beginning at a such that $\mathcal{A}, a, \pi \models_{LTL} \varphi$.

Satisfiability of a formula $\varphi \in LTL(AP)$ is a **decidable** property. In Maude, the satisfiability decision procedure is supported by the predefined functional module SAT-SOLVER.

Sat Solver (II)

One can define the desired atomic predicates in a module extending SAT-SOLVER, such as, for example,

```
fmod TEST is
  inc SAT-SOLVER .
  ops a b c d e p q r : -> Prop .
endfm
```

The user can then decide the satisfiability of an LTL formula involving those atomic propositions by applying the operator,

```
op satSolve : Formula -> [SatSolveResult] [special ... ]
```

to the given formula and evaluating the expression. The resulting solution of sort `SatSolveResult` is then either `false`, if no model exists, or a finite model satisfying the formula.

Sat Solver (III)

Such a model is described by a pair of finite paths of states: an initial path leading to a cycle. Each state is described by a conjunction of atomic propositions or negated atomic propositions, with the propositions not mentioned in the conjunction being “don’t care” ones.

For example, we can evaluate,

```
Maude> red satSolve(a /\ (0 b) /\ (0 0 ((~ c)/\ [](c \/ (0 c))))) .  
reduce in TEST : satSolve(0 0 (~ c /\ [](c \/ 0 c)) /\ (a /\ 0 b)) .  
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)  
result SatSolveResult: model(a ; b, (~ c) ; c)
```


Tautology Checker

We call $\varphi \in LTL(AP)$ a **tautology** iff $\mathcal{A}, a, \pi \models_{LTL} \varphi$ holds for every Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ with $L : A \longrightarrow \mathcal{P}(AP)$, every state $a \in A$, and every path π in \mathcal{A} beginning at a . It then follows easily that φ is a tautology iff $\neg\varphi$ is unsatisfiable.

Therefore, the module SAT-SOLVER can also be used as a tautology checker. This is accomplished by using the following operators and equations in SAT-SOLVER:

```
op tautCheck : Formula -> [TautCheckResult] .
op $invert : SatSolveResult -> TautCheckResult .
var F : Formula .      vars L C : FormulaList .
eq tautCheck(F) = $invert(satSolve(~ F)) .
eq $invert(false) = true .
eq $invert(model(L, C)) = counterexample(L, C) .
```

Tautology Checker (II)

The `tautCheck` function returns either `true` if the formula is a tautology, or a finite model that does not satisfy the formula. For example, we can evaluate:

```
Maude> red tautCheck( (p U r) /\ (q U r) <-> ((p /\ q) U r) ) .
reduce in TEST : tautCheck((p U r) /\ (q U r) <-> p /\ q U r) .
rewrites: 31 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The tautology checker gives us also a **decision procedure for semantic LTL equality**. Assuming a countable set $X = \{x_1, \dots, x_n, \dots\}$ of variables, the **semantic LTL equality** relation \equiv_{LTL} is the binary relation on $LTL(X)$ defined by,

$$\varphi \equiv_{LTL} \psi \iff \varphi \leftrightarrow \psi \text{ is a tautology.}$$

As shown in Appendix C, \equiv_{LTL} is **closed under equational deduction**.

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 0 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 (I)

Use double depth first method (Holzmann et al.).

- First DFS looks for accepting state-pair p .
- Second DFS starts from p and looks for a state-pair on the first DFS stack to complete an accepting cycle.

Avoids a fairness check for every cycle in product.

Searching the Synchronous Product (II)

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 abstract state indices and is Maude-independent.

LTL Satisfiability Solving

Satisfiability is decided by SCC optimization phase of Büchi automaton construction. A formula is satisfiable iff there is a fair SCC reachable from initial state in its generalized Büchi automaton. In positive case construct a witness:

1. Find shortest path from initial state to a state s in a fair SCC.
2. Iterate(find shortest path that satisfies at least additional fairness condition).
3. Find a shortest path back to s .
4. Shorten path from initial state to s by folding it into cycle if formulae allow.
5. For each arc in initial path and fair cycle, extract a prime implicant from the BDD representation of its formula.

Performance Comparisons with SPIN

We have compared the performance of the SPIN and Maude model checkers as follows. Given a system specified in PROMELA, we specify it in Maude, and then compare, for a given model checking problem, the running times as well as memory consumptions of SPIN and of the Maude LTL model checker on the respective specifications.

Only properties satisfied by the corresponding systems were model checked to force the exploration of the entire search space.

Except where stated, the default settings for SPIN were used everywhere. The analyses were carried out on a dual 1GHz Pentium III machine with 1GB RAM running Red Hat Linux 7.1 and on a single 1.13 GHz Pentium III machine with 384MB RAM.

Size ^a	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
2	16	10	1.49	4.14
3	201	550	2.67	6.37
4	155,737	72,860	214.80	176.18

Table 1: Peterson's algorithm

^aWith the default settings, the Spin model checker ran out of memory for size 4. The size 4 benchmark was obtained after turning on the Spin option -DCOLLAPSE.

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	17	250	1.49	4.74
10	28	580	1.49	5.56
50	770	2,020	13.11	26.27
100	5,702	14,500	104.86	227.33

Table 2: Leader election - property 1

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	18	260	1.49	4.73
10	26	590	1.49	5.56
50	734	2,010	13.11	26.35
100	5,409	14,400	104.86	223.65

Table 3: Leader election - property 2

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	21	290	1.49	4.75
10	36	600	1.60	5.58
50	1,706	2,510	22.94	43.46
100	12,811	20,420	209.72	320.72

Table 4: Leader election - property 3

Size	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
5	20	260	1.49	4.73
10	35	590	1.60	5.57
50	1,193	2,310	22.94	37.96
100	9,088	19,440	209.72	265.12

Table 5: Leader election - property 4

Property	Average time taken (ms.)		Average memory usage (MB)	
	Spin	Maude	Spin	Maude
Literal model : Safety	137	520	2.21	3.56
Simplified model : Safety	39	120	1.60	3.56
Literal model : Progress	90	200	1.49	3.56
Simplified model : Progress	26	90	1.49	3.56

Table 6: Mobile handoff

Conclusions and Future Directions

We have presented the Maude LTL model checker and its LTL satisfiability and tautology checkers. 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.