# The Maude LTL Model Checker and its Implementation

Steven Eker[1], José Meseguer[2], and Ambarish Sridharanarayanan[2]

[1] Computer Science Laboratory, SRI International, Menlo Park, CA 94025
eker@csl.sri.com
[2] CS Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801
{meseguer, srdhrnry}@cs.uiuc.edu

## 1 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. The Maude LTL model checker has been designed with the goal of combining a very expressive and general system specification language (Maude [1]) with an LTL model checking engine that benefits from some of the most recent advances in on-the-fly explicit-state model checking techniques.

Specifically, Maude specifications are *executable logical theories* in rewriting logic [2], a logic that is a flexible logical framework for expressing a very wide range of concurrency models and distributed systems [2]. A rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, with $(\Sigma, E)$ an equational theory specifying a system's *distributed state structure* (for example, a multiset of processes and messages) and with $R$ a collection of rewrite rules specifying the *concurrent transitions* of the system. Since no domain-specific model of concurrency is built into the logic, the range of applications that can be naturally specified is indeed very wide. For example, besides conventional distributed system specifications, properties of signalling pathways in mammalian cells have been model checked [3]. Another advantage of Maude as the system specification language is that integration of model checking with theorem proving techniques becomes quite seamless. The same rewrite theory $\mathcal{R} = (\Sigma, E, R)$ can be the input to the LTL model checker and to several other proving tools in the Maude environment [4]. For a lengthier discussion of the Maude LTL model checker and its LTL satisfiability and tautology procedures, see the companion paper [5].

## 2 LTL Model Checking of Maude Specifications

A Maude module is a rewrite theory $\mathcal{R} = (\Sigma, E, R)$. Fixing a distinguished sort *State*, the initial model $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has an underlying Kripke structure $\mathcal{K}(\mathcal{R}, \mathit{State})$ given by the total binary relation extending its one-step sequential rewrites. To the initial algebra of states $T_{\Sigma/E}$ we can likewise associate equationally-defined *computable state predicates* as atomic predicates for such a Kripke structure. In this way we obtain a language of LTL *properties* of the rewrite theory $\mathcal{R}$.

Maude 2.0 supports on-the-fly LTL model checking for initial states $[t]$, say of sort *State*, of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ such that the set $\{[u] \in T_{\Sigma/E} \mid \mathcal{R} \vdash [t] \to [u]\}$, of all states *reachable* from $[t]$ is *finite*. The rewrite theory $\mathcal{R}$ should satisfy reasonable executability requirements, such as the confluence and termination of the equations $E$ and coherence of the rules $R$ relative to $E$ [1]. In Maude the rewrite theory $\mathcal{R}$ is specified as a module, say `M`. Then, given an initial state, say `init` of sort `State`$_{\mathtt{M}}$, 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 the predefined module `MODEL-CHECKER` as submodules;
- giving a *subsort declaration*, `subsort State`$_{\mathtt{M}}$ `< State .`, where `State` is one of the key sorts in the module `MODEL-CHECKER`;
- 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.
- defining the *semantics* of the state predicates by means of equations.

Once the semantics of each of the state predicates has been defined, we are then ready, given an initial state `init`, to model check any LTL formula, say `form`, involving such predicates. We do so by evaluating in Maude, the expression `init |= form .` Two things can then happen: if the property `form` holds, then we get the result `true`; if it doesn't, we get a counterexample expressed as a finite path followed by a cycle.

## 3 Model Checking Algorithms and Implementation

On-the-fly LTL model checking is performed by constructing a Büchi automaton from the negation of the property formula and lazily searching the synchronous product of the Büchi automaton and the system state transition diagram for a reachable accepting cycle.

**Büchi Automaton Construction.** The negated LTL formula is converted to negative normal form and heuristically simplified by a set of Maude equations, mostly derived from the simplification rules in [6, 7]. Rather than the classical tableaux construction [8], we use a newer technique proposed in [9] based on very weak alternating automata, comprising three basic steps: (1) construct a very weak alternating automaton from the formula, (2) convert the very weak alternating automaton into a generalized Büchi automaton (with multiple fairness conditions on arcs) and (3) convert the generalized Büchi automaton into a regular Büchi automaton. Optimizations and simplifications are performed after

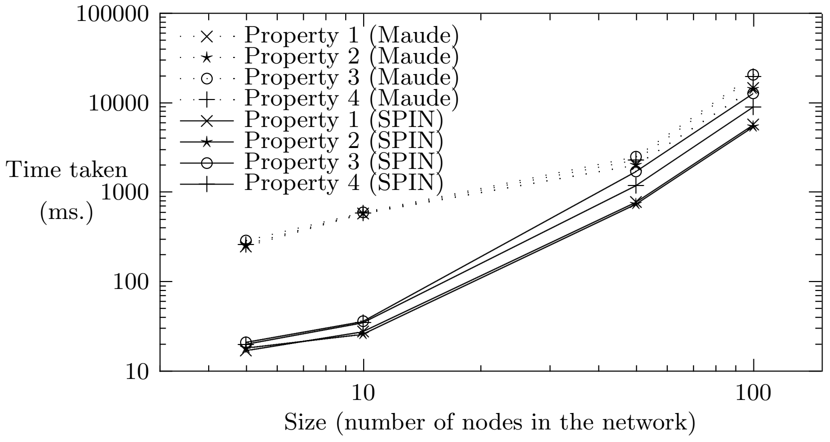

**Fig. 1.** Execution time for the leader-election problem - Maude vs. SPIN

each step and we add some strongly connected component optimizations adapted from those in [7]. Throughout the computation, the pure propositional subformulae labelling the arcs of the various automata are stored as BDDs to allow computation of conjunctions, elimination of contradictions, and combination of parallel arcs by disjunction.

**Searching the Synchronous Product.** We use the double depth first method of [10] to lazily generate and search the synchronous product. For each system state generated we keep five bit vectors to record: (1) which propositions have been tested in the state; (2) which propositions were true in the state; (3) which product pairs (with automaton states) have been seen by the first depth first search; (4) which product pairs are currently on the first depth first search stack; and (5) which product pairs have been seen by the second (nested) depth first search. The full term graph representation of each system state is maintained (in order to test propositions) in a separate hash table which also keeps track of rewrites between system states.

**Performance Evaluation.** We compared the performance of the Maude LTL model checker vis-a-vis the SPIN LTL model checker as follows. Given a system specified in PROMELA, we specify it in Maude, and then compare the running times and the memory consumptions of the two model checkers on the respective specifications. The PROMELA specifications used — a solution to the mutual exclusion problem, a solution to the leader election problem for a unidirectional ring network, and a translation of the $\pi$-calculus description of a mobile handoff scenario — are all available on the SPIN web-page.

In all the above situations, only properties satisfied by the corresponding systems were model checked; no generation of counterexamples was attempted.

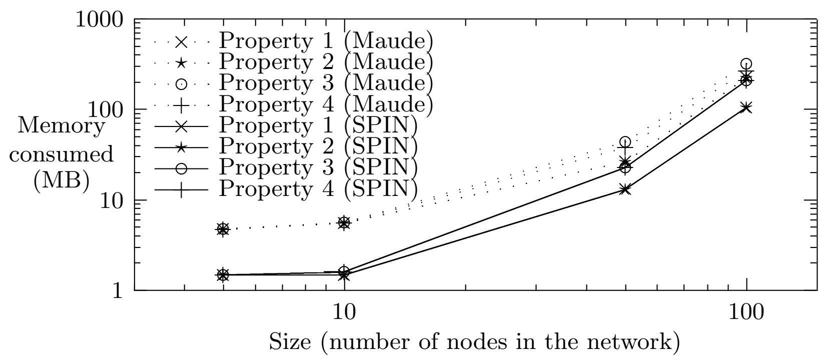

**Fig. 2.** Total memory consumption for the leader-election problem - Maude vs. SPIN

Except in one instance, the default settings for SPIN were used everywhere. The analyses were carried out on a 1.13 GHz Pentium III machine with 384 MB RAM running Red Hat Linux. In most of the cases, both model checkers finished fairly quickly whenever memory was available; lack of memory proved to be the main bottleneck for scalability. The benchmarks showed a comparable performance of SPIN and the Maude LTL model checker, in terms of both speed and memory consumption. The results for the leader election problem are given in Figures 1 and 2. A fuller description of the other algorithms and their respective comparisons are discussed in [5].

## References

1. Clavel, M., et al.: Maude: specification and programming in rewriting logic. Theoretical Computer Science **285** (2002) 187–243
2. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96** (1992) 73–155
3. Eker, S., et al.: Pathway logic: Executable models of biological networks. In: Proc. WRLA'02. Volume 71 of ENTCS., Elsevier (2002)
4. Clavel, M., et al.: Building equational proving tools by reflection in rewriting logic. In: CAFE: An Industrial-Strength Algebraic Formal Method, Elsevier (2000)
5. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Proc. WRLA '02. Volume 71 of ENTCS., Elsevier (2002)
6. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: CONCUR 2000. Number 1877 in LNCS, Springer-Verlag (2000) 153–167
7. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: CAV '00. Number 1633 in LNCS, Springer-Verlag (2000) 247–263
8. Gerth, R., et al.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification Testing and Verification, Chapman & Hall (1995) 3–18
9. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: CAV '01. Number 2102 in LNCS, Springer-Verlag (2001) 53–65
10. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. Design: An International Journal **13** (1998) 289–307