Electronic Notes in Theoretical Computer Science 71 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume71.html 26 pages

The Maude LTL Model Checker

Steven Eker [a], José Meseguer [b], and
Ambarish Sridharanarayanan [b]

[a] *Computer Science Laboratory, SRI International, Menlo Park, CA 94025.*

[b] *CS Department, University of Illinois at Urbana-Champaign, Urbana IL 61801.*

Abstract

The Maude LTL model checker supports on-the-fly explicit-state model checking of concurrent systems expressed as rewrite theories with performance comparable to that of current tools of that kind, such as SPIN. This greatly expands the range of applications amenable to model checking analysis. Besides traditional areas well supported by current tools, such as hardware and communication protocols, many new applications in areas such as rewriting logic models of cell biology, or next-generation reflective distributed systems can be easily specified and model checked with our tool.

Key words: Model checking, concurrent systems, Maude

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. In the conclusions of their excellent model checking book [2], Clarke, Grumberg and Peled state that,

> "... an obvious problem with current systems is how to make the specification language more expressive and easier to use."

While they seem to have mostly the property specification level (2) in mind, we think that it is just as important to greatly increase the expressive power of system specification languages. The point is that 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. For example, PROMELA [10]—SPIN's system specification language—is designed and optimized with the purpose of efficiently model checking distributed algorithm specifications. This is perfectly reasonable, but it comes at the cost of limiting the kinds of systems that can be naturally specified. In particular: (1) processes cannot be nested, in the sense of containing inside other subprocesses, sub-subprocesses, and so on; (2) communication is assumed to happen through FIFO channels; and (3) there is a limited supply of data types in terms of which all other data must be encoded. Nested processes or "objects" are the natural way to specify both rewriting logic models of cell biology that express cell states as *nested soups* of protein complexes in the membranes, cytoplasm, and nucleus [7,6], and also reflective distributed systems involving arbitrary nesting of metaobjects that control groups of other objects below them—the so-called "Russian dolls" model of distributed object reflection [16]. Specifying such nested processes seems hard to accomplish in a language such as PROMELA. By contrast, all the applications just mentioned can be easily specified in Maude (see [7,6,16]). Since the only requirement of the LTL model checker is the finiteness of the reachable states, provided that holds, such applications, and any others expressible as rewrite theories can be model checked by Maude's LTL checker without any modification whatsoever (see [6], resp. [24], for Maude model checking applications to cell biology, resp. active networks).

Furthermore, the Maude LTL checker can model check systems whose states involve data in *data types of infinite cardinality*, such as numbers, lists, or multisets of arbitrary size; in fact, in any algebraic data types. The only assumption is that the set of states *reachable* from a given initial state is finite. This finitary reachability condition can be dropped in the case of the semidecidable search for counterexamples for safety properties of infinite-state systems supported in Maude by its `search` command (this capability is not discussed in this paper, due to space limitations).

There is currently a conscious effort to extend the expressiveness of concurrent system specifications in languages such as SAL [22], that explicitly lifts the finite-state restrictions and supports the specification of infinite-state systems that can then be abstracted into finite-state ones for model checking purposes, with the PVS theorem prover verifying the correctness of the abstractions. Maude's expressiveness goals are similar to SAL's, but with two important differences: (1) Maude is executable, whereas SAL in general is not; and (2) SAL is designed to support fixed synchronous and/or asynchronous communication topologies between a fixed (although possibly parametric) number of communicating objects; whereas Maude allows specification of highly dynamic and possibly nested reflective communication architectures.

In summary, the main contribution of this work is to combine a very expressive executable system specification language, namely Maude [3], 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 [1]. The great generality and flexibility of rewriting logic as a semantic framework [15,19] is of course the reason for Maude's expressiveness. 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 paper is organized as follows. The semantics of linear temporal logic for an arbitrary rewrite theory $\mathcal{R}$ is explained in Section 2. The functionality of the LTL model checker is described in Section 3, and the satisfiability and tautology checker in Section 4. The algorithms and implementation are then described in Section 5. Performance comparisons with SPIN are given in Section 6. Conclusions are drawn in Section 7.

2 The LTL Properties of a Rewrite Theory $\mathcal{R}$

Fixing a distinguished sort *State*, the initial model $\mathcal{T}_\mathcal{R}$ of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has an underlying Kripke structure given by the total binary relation extending its one-step sequential rewrites. Since $T_{\Sigma/E}$ has an algebraic Σ-algebra structure, there is a very expressive first-order language of *state predicates* for such a Kripke structure. We can then associate to the underlying Kripke structure and state predicate language of $\mathcal{R}$ a linear temporal logic in the standard way, that is, the language of LTL *properties* of the rewrite theory $\mathcal{R}$. We make all this precise in what follows, beginning with some background material on Kripke structures and LTL.

A binary relation $R \subseteq A \times A$ on a set A is called *total* iff for each $a \in A$ there is at least one $a' \in A$ such that $(a, a') \in R$. If R isn't total, it can be made total by defining, $R^\bullet = R \cup \{(a, a) \in A^2 \mid \not\exists a' \in A \; (a, a') \in R\}$.

Definition. A *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_\mathcal{A}, L)$ such that A is a set, called the set of *states*, $\rightarrow_\mathcal{A}$ is a total binary relation on A, called the *transition relation*, and $L : A \longrightarrow \mathcal{P}(AP)$ is a function, called the *labeling function* associating to each state $a \in A$ the set $L(a)$ of those *atomic propositions* in AP that *hold* in the state a.

Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory [17], with (Σ, E) its underlying membership equational theory [18], and let *State* be a sort in Σ such that, for each $[t] \in T_{\Sigma/E,State}$, if $[\alpha] : [t] \longrightarrow [t']$ is a rewrite proof in $\mathcal{T}_\mathcal{R}$, then $[t'] \in T_{\Sigma/E,State}$. We can then associate to $\mathcal{R}$ and *State* the Kripke structure $\mathcal{K}(\mathcal{R}, State) = (T_{\Sigma/E,State}, \rightarrow^\bullet_\mathcal{R}, L_\mathcal{R})$, where $\rightarrow_\mathcal{R}$ is the *one-step sequential $\mathcal{R}$-rewriting relation* on $T_{\Sigma/E,State}$, that is, the set of all pairs $([t], [t']) \in T^2_{\Sigma/E,State}$ such that there is a one-step sequential rewrite proof (see [17]) $[\alpha] : [t] \rightarrow [t']$ in $\mathcal{T}_\mathcal{R}$; its associated total relation $\rightarrow^\bullet_\mathcal{R}$ adds a self-loop for each *deadlock state*, that is, for each $[t] \in T_{\Sigma/E,State}$ that cannot be further rewritten. The labeling function $L_\mathcal{R}$ is of the form $L_\mathcal{R} : T_{\Sigma/E,State} \longrightarrow \mathcal{P}(SPred_0(\Sigma, State))$, where $SPred_0(\Sigma, State)$ is the set of first-order formulae, P, called *state predicates*, in the first order language with equality[1] $FOL(\Sigma)$ that have a single, fixed

[1] This language is of course many-kinded; furthermore, we allow the use of sorted variables as abbreviations for kinded variables that satisfy the corresponding sort predicate.

free variable x of sort *State*, that is, $fvars(P) = \{x\}$. The function $L_{\mathcal{R}}$ then maps each state $[t] \in T_{\Sigma/E,State}$ to the set of state predicates that hold in $[t]$, that is, $L_{\mathcal{R}}([t]) = \{P \in SPred_0(\Sigma, State) \mid T_{\Sigma/E} \models_{FOL} P(x \mapsto t)\}$. Note that the definition of $L_{\mathcal{R}}([t])$ just given does not depend on the choice of the representative t in $[t]$.

Given a set AP of atomic predicates, the language $LTL(AP)$ of *linear temporal logic formulae* on AP can be defined in the usual way as the (unsorted) free algebra $T_{\Sigma_{LTL}}(AP)$ on the set AP for the signature Σ_{LTL} with constants $\top$ (true),$\perp$ (false), unary operators $\neg$ (negation), $\bigcirc$ (next), $\Diamond$ (eventually), and $\Box$ (henceforth), and with binary operators $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), $\mathcal{U}$ (until), and $\mathcal{R}$ (release). The *models* of $LTL(AP)$ are Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ having AP as their atomic propositions, that is, with $L : A \longrightarrow \mathcal{P}(AP)$. Since LTL formulae are implicitly universally quantified on infinite computation paths, the *satisfaction relation* $\mathcal{A}, a \models_{LTL} \varphi$ holds between a Kripke structure $\mathcal{A}$, one of its states a, and a formula $\varphi \in LTL(AP)$ iff for all infinite computation paths π of $\mathcal{A}$ beginning at state a the satisfaction relation $\mathcal{A}, a, \pi \models_{LTL} \varphi$ holds (see for example [2], where boldface capital letters are used for all the temporal operators).

In particular, given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with a sort *State* satisfying the above assumptions, a state $[t] \in T_{\Sigma/E,State}$, and an LTL formula $\varphi \in LTL(SPred_0(\Sigma, State))$, the satisfaction relation, $\mathcal{K}(\mathcal{R}, State), [t] \models_{LTL} \varphi$, states that the (Kripke structure associated to the) rewrite theory $\mathcal{R}$ and the initial state $[t]$ satisfy the LTL property φ. When no confusion can arise, we may leave implicit the choice of the sort *State* and abbreviate the above satisfaction relation by, $\mathcal{R}, [t] \models_{LTL} \varphi$. This gives a precise semantics to the LTL properties satisfied by a rewrite theory $\mathcal{R}$. We are of course interested in both deductive techniques to prove such properties, and, whenever possible, in *model checking decision procedures* to decide satisfaction of an LTL property φ in a given initial state $[t]$.

It is worth pointing out that $SPred_0(\Sigma, State)$ is a quite general set of *parameterless* state predicates. It can be further generalized to a more general set $SPred(\Sigma, State)$ of *state predicates with parameters* by relaxing the requirement $fvars(P) = \{x\}$ to the weaker requirement $\{x\} \subseteq fvars(P)$. That is, for $P \in SPred(\Sigma, State)$ we call the variables in $fvars(P) - \{x\}$ its *parameters*. We can then define a more general temporal logic $LTL(SPred(\Sigma, State))$ where the state predicates can have parameters, and a satisfaction relation $\mathcal{R}, a \models_{LTL} \varphi$, where now a is an assignment of values in $T_{\Sigma/E}$ for all the variables of the formula φ, including its parameters. This goes beyond the strictly "propositional" semantics treated in [2], but the generalization is unproblematic. Even when (Σ, E) is confluent and terminating, the satisfaction of a state predicate $P \in LTL(SPred(\Sigma, State))$ may in general be undecidable. This is acceptable for deductive uses, but not for model checking ones. As further explained in Section 3, one of the ways in which the Maude LTL model checker allows defining very general properties is by allowing state predicate definitions in a rich subclass of *decidable* parametric state predicates in $SPred(\Sigma, State)$.

3 The Maude LTL Model Checker

Maude 2.0 supports on-the-fly LTL model checking for initial states $[t]$, say of sort *State*, of a finitary 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 the already mentioned requirement that all such reachable states $[u]$ have also sort *State*. Furthermore, the equational theory (Σ, E) should be confluent and terminating (perhaps *modulo* some axioms such as associativity, commutativity, or identity) and the rules R should be *coherent* relative to the equations E [25]. Note that many rewrite theories of interest may have an *infinite* number of states, yet the states reachable from any given initial state may still be finite.

A rewrite theory $\mathcal{R}$ satisfying the above assumptions can be specified in Maude by a system module, say `M`. Then, given an initial state, say `init` of sort $\texttt{State}_{\texttt{M}}$, 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 (we can include other submodules as well if we wish, for example to introduce auxiliary data types and functions);
- giving a *subsort declaration*, `subsort` $\texttt{State}_{\texttt{M}}$ `< State .`, where `State` is one of the key sorts in the module `MODEL-CHECKER` (this declaration can be omitted if $\texttt{State}_{\texttt{M}} = \texttt{State}$);
- 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.
- 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 a set of (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 $\texttt{State}_\texttt{M}$, 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(u1i,...,uni)`, $1 \leq i \leq k$ are likewise *patterns* of sort `Prop`;
- each condition `Ci`, $1 \leq i \leq k$, is a conjunction of equalities and memberships; such conditions may involve *auxiliary functions*, either imported from auxiliary modules, or defined by additional equations in our module `CHECK-M`.

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. Such LTL formulas are *ground terms*[2] of sort `Formula` in `CHECK-M`; we do so by giving the Maude command,

```
  reduce init |= form .
```

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

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

This is because, if an LTL formula φ is not satisfied by a finite Kripke structure, it is always possible to find a counterexample for φ having the form of a *path of transitions followed by a cycle.* The first argument of the above constructor is the path leading to the cycle, and the second is the cycle itself. Each transition is represented as a *pair*, consisting of a state and a rule label.

Note that we have defined the syntax and semantics of the state predicates in such a way that *their state argument is left implicit.* For example, a parameterized state predicate `p` with parameters of sorts `S1`, ..., `Sm` is defined by an operator,

```
  op p : S1 ... Sm -> Prop .
```

instead than by an operator

```
  op p : State S1 ... Sm -> Prop .
```

Note that the semantic equations in the first syntax,

```
  ceq exp1 |= p(u11,...,un1) = true if C1 .
  ...
  ceq expk |= p(u1k,...,unk) = true if Ck .
```

correspond exactly to the equations,

```
  ceq p(exp1,u11,...,un1) = true if C1 .
  ...
  ceq p(expk,u1k,...,unk) = true if Ck .
```

[2] As explained above, we allow the definition of *parameterized* state predicates; however, in any LTL formula presented to the model checker such parameterized state predicates must be *instantiated* by adequate ground terms for each of their parameters.

in the second syntax.

This observation helps clarify a further very convenient feature about how state predicates are specified in the LTL model checker, namely that *only the positive cases have to be specified*; that is, if a state predicate *ground expression* of the form `exp |= p(w1,...,wk)` (equivalent to `p(exp,w1,...,wk)` in the second syntax) cannot be simplified to `true`, then it is *assumed to be false*[3]

How general is the above-described method of defining state predicates? Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory expressed as a module `M` and satisfying all the assumptions already stated earlier in this section. Let p be defined by the above k equations in its explicit state formulation, with the i^{th} equation involving the set of variables X_i, and assume that all auxiliary functions that might be needed in the conditions have already been defined in `M`. Then, the state predicate p yields a *definitional extension* of the first-order language $FOL(\Sigma)$ associating to p the following disjunctive formula, which is *decidable* for any ground instance in $T_{\Sigma,E}$:

$$\begin{aligned} p(x, y_1, \ldots, y_n) = true \quad &\text{iff} \\ &((\exists X_1)\ x = exp_1\ \wedge y_1 = u_{11} \wedge \ldots \wedge y_n = u_{n1} \wedge C_1)\ \vee \\ &\ldots \vee\ ((\exists X_k)\ x = exp_k\ \wedge y_1 = u_{1k} \wedge \ldots \wedge y_n = u_{nk} \wedge C_k). \end{aligned}$$

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

```
fmod LTL is
  sorts Prop Formula .
  subsort Prop < 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 O_ : 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] .
```

[3] However, the user is allowed to give definitions of the form,
`ceq exp |= p(u1,...,un) = bexp if C .`
with `bexp` an arbitrary Boolean expression; also `C` my be empty, so that an *unconditional* equation (`eq`) suffices.

```
  op _W_ : Formula Formula -> Formula [prec 65] .   *** weak until
  op _|->_ : Formula Formula -> Formula [prec 65] . *** leads-to

  vars f g : Formula .
  eq f -> g = ~ f \/ g .
  eq f <-> g = (f -> g) /\ (g -> f) .
  eq <> f = True U f .
  eq [] f = False R f .
  eq f W g = (f U g) \/ [] f .
  eq f |-> g = [](f -> (<> g)) .

  *** negative normal form
  eq ~ True = False .
  eq ~ False = True .
  eq ~ ~ f = f .
  eq ~ (f \/ g) = ~ f /\ ~ g .
  eq ~ (f /\ g) = ~ f \/ ~ g .
  eq ~ O f = O ~ f .
  eq ~(f U g) = (~ f) R (~ g) .
  eq ~(f R g) = (~ f) U (~ g) .
endfm
```

The equations in this module do two things: (1) they express all defined LTL operators in terms of the basic operators True, False, negation, conjunction, disjunction, next, O, until, U, and release, R; and (2) they 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*.

We illustrate the use of the Maude model checker with Dekker's algorithm, one of the earliest correct solutions to the mutual exclusion problem. 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 right away. In case of a tie (both variables set to 1) the tie is broken using a variable turn that takes values in $\{1, 2\}$.

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 .
```

where the fragments of code for the critical section and for the remaining part of the program are respectively abstracted as constants `crit` and `rem`. We assume that `crit` is *terminating*, but no such assumption is made about `rem`. In the Maude specification in Appendix A this is achieved by declaring subsorts and constants,

```
 subsorts LoopingUserStatement < UserStatement < Program .
 op crit : -> UserStatement .
 op  rem : -> LoopingUserStatement .
```

and by semantic rules where a `UserStatement` not in `LoopingUserStatement` always terminates, but a `LoopingUserStatement` may not terminate. The code of process 2 is entirely symmetric. The Maude specification of the semantics of a simple parallel language supporting the above features is given in Appendix A. The two processes are defined in a module `DEKKER` that imports the module `PARALLEL` defining the semantics of the parallel language.

To specify relevant properties of Dekker's algorithm we define three state predicates parameterized by the process id: `enterCrit`, when the process enters its critical section, `in-rem`, when it is in its `rem` part, 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 in-rem 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 {[I, rem ; R] | S, M, J} |= in-rem(I) = true .
 eq {S, M, J} |= exec(J) = true .
endm
```

We can then verify that the *mutual exclusion property* is satisfied:

```
reduce in CHECK : initial |= []~ (enterCrit(1) /\ enterCrit(2)) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1156 in 40ms cpu (40ms real) (28900 rewrites/second)
```

```
result Bool: true
```

The *strong liveness property* that executing infinitely often implies entering one's critical section infinitely often fails. The Maude LTL model checker returns a counterexample.

```
reduce in CHECK : initial |= []<> exec(1) -> []<> enterCrit(1) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 16 system states.
rewrites: 148 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: 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 ;  ...
```

Since `rem` may not terminate, the weaker liveness property that if *both* `p1` and `p2` execute infinitely often, then both enter their critical sections infinitely often also fails.

```
reduce in CHECK : initial |= []<> exec(1) /\ []<> exec(2) ->
                           []<> enterCrit(1) /\ []<> enterCrit(2) .
ModelChecker: Property automaton has 7 states.
ModelCheckerSymbol: Examined 236 system states.
rewrites: 1463 in 60ms cpu (60ms real) (24383 rewrites/second)
result ModelCheckResult: 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 ;  ...
```

What *does* hold is the more subtle weak liveness property that if p1 and p2 both get to execute infinitely often, then if p1 is infinitely often out of its `rem` section, then p1 enters its critical section infinitely often. Of course, the symmetric statement holds true for p2.

```
reduce in CHECK : initial |= []<> exec(1) /\ []<> exec(2) ->
                       []<> ~ in-rem(1) -> []<> enterCrit(1) .
ModelChecker: Property automaton has 5 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1661 in 70ms cpu (70ms real) (23728 rewrites/second)
result Bool: true
```

The above Dekker algorithm example illustrates a general capability to model check in Maude *any* program (or abstraction of a program, having finitely many reachable states) in *any* programing language: we just have to define in Maude the language's rewriting semantics and the state predicates.

4 The Satisfaction Solver and Tautology Checker

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

which is satisfied by a four-state model with `a` holding in the first state, `b` holding in the second, `c` not holding in the third but holding in the fourth, and the fourth state going back to the third.

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) .
```

so that 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 $X = \{x_1, \ldots, x_n, \ldots\}$ of variables, the *semantic* LTL *equality* relation $\equiv_{LTL}$ is the binary relation on $LTL(X)$ defined by,

$$\varphi \equiv_{LTL} \psi \quad \Leftrightarrow \quad \varphi \leftrightarrow \psi \textit{ is a tautology.}$$

The key observation (proved in Appendix B) is that $\equiv_{LTL}$ is a *substitution-closed* Σ_{LTL}*-congruence*, and therefore (see, e.g., [4] Thm. IV-1.2) closed under equational deduction. That is, defining $E_{LTL} = \{\varphi = \psi \mid \varphi, \psi \in LTL(X) \wedge \varphi \equiv_{LTL} \psi\}$, for any $\mu, \nu \in LTL(X)$ we have,

$$E_{LTL} \vdash \mu = \nu \quad \Leftrightarrow \quad \mu \equiv_{LTL} \nu.$$

5 Model Checking Algorithms and Implementation

On-the-fly LTL model checking consists of two major steps [11]. First, we construct a Büchi automaton associated to the negation of the temporal logic formula that recognizes the language of counterexamples. Second, we 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.

5.1 Büchi Automaton Construction

Given an LTL formula ϕ we wish to construct a Büchi automaton that accepts the language of ω-words satisfying $\neg\phi$. The first phase is to put $\neg\phi$ in negative normal form as explained in Section 3, possibly simplifying the formula in order to produce a smaller automaton.

The LTL simplification is done in Maude by including the optional module `LTL-SIMPLIFIER`. Such simplification is necessarily heuristic; the general idea being to reduce the number of temporal operators and to push propositional operators inside of temporal operators. Purely propositional subformulae can be handled by propositional techniques and need not give rise to additional automaton states. There are several simplification schemes in the literature. The main technique we use is the method of Etessami and Holzmann [8], with a minor refinement in the case of the $\bigcirc$ operator. This method is based on syntactically classifying subsets of LTL formulae as *pure eventuality formulae* or *pure universality formulae* and performing rewrites that are only valid for formulae that belong to the appropriate subset. We introduce a third set of LTL formulae (*pure formulae*) that is the intersection of the other two sets. These notions, together with their syntactic basis, can be mapped rather neatly onto the order-sorted fragment of Maude's type system by introducing additional sorts to represent the formulae `True` and `False`.

```
fmod LTL-SIMPLIFIER is
  inc LTL .
  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
          PE-Formula PU-Formula < Formula .
  op True : -> TrueFormula [ditto] .
  op False : -> FalseFormula [ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ditto] .
  op _/\_ : PU-Formula PU-Formula -> PU-Formula [ditto] .
  op _/\_ : PureFormula PureFormula -> PureFormula [ditto] .
  op _\/_ : PE-Formula PE-Formula -> PE-Formula [ditto] .
  op _\/_ : PU-Formula PU-Formula -> PU-Formula [ditto] .
  op _\/_ : PureFormula PureFormula -> PureFormula [ditto] .
  op O_ : PE-Formula -> PE-Formula [ditto] .
  op O_ : PU-Formula -> PU-Formula [ditto] .
  op O_ : PureFormula -> PureFormula [ditto] .
  op _U_ : PE-Formula PE-Formula -> PE-Formula [ditto] .
  op _U_ : PU-Formula PU-Formula -> PU-Formula [ditto] .
  op _U_ : PureFormula PureFormula -> PureFormula [ditto] .
  op _U_ : TrueFormula Formula -> PE-Formula [ditto] .
  op _U_ : TrueFormula PU-Formula -> PureFormula [ditto] .
  op _R_ : PE-Formula PE-Formula -> PE-Formula [ditto] .
  op _R_ : PU-Formula PU-Formula -> PU-Formula [ditto] .
  op _R_ : PureFormula PureFormula -> PureFormula [ditto] .
  op _R_ : FalseFormula Formula -> PU-Formula [ditto] .
  op _R_ : FalseFormula PE-Formula -> PureFormula [ditto] .
```

Using the rule numbers from the original paper, the simplification rules can be expressed as Maude equations:

```
  vars p q r s : Formula .
  var pe : PE-Formula .
  var pu : PU-Formula .
  var pr : PureFormula .

  *** Rules 1, 2 and 3; each with its dual.
  eq (p U r) /\ (q U r) = (p /\ q) U r .
  eq (p R r) \/ (q R r) = (p \/ q) R r .
  eq (p U q) \/ (p U r) = p U (q \/ r) .
  eq (p R q) /\ (p R r) = p R (q /\ r) .
  eq True U (p U q) = True U q .
  eq False R (p R q) = False R q .

  *** Rules 4 and 5 do most of the work.
  eq p U pe = pe .
  eq p R pu = pu .

  *** An extra rule in the same style.
  eq O pr = pr .
```

For further simplification we also include the rewrite rules of Somenzi and Bloem [23] that are not subsumed by the Etessami and Holzman method.

These include conditional rules that involve an auxiliary binary relation $\leq$.

```
  *** Four pairs of duals.
  eq O p /\ O q = O (p /\ q) .
  eq O p \/ O q = O (p \/ q) .
  eq O p U O q = O (p U q) .
  eq O p R O q = O (p R q) .
  eq True U O p = O (True U p) .
  eq False R O p = O (False R p) .
  eq (False R (True U p)) \/ (False R (True U q)) =
                              False R (True U (p \/ q)) .
  eq (True U (False R p)) /\ (True U (False R q)) =
                              True U (False R (p /\ q)) .

  *** <= relation on formulae
  op _<=_ : Formula Formula -> Bool [prec 75] .

  eq p <= p = true .
  eq False <= p  = true .
  eq p <= True = true .

  ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
  ceq p <= (q \/ r) = true if p <= q .
  ceq (p /\ q) <= r = true if p <= r .
  ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .

  ceq p <= (q U r) = true if p <= r .
  ceq (p R q) <= r = true if q <= r .
  ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
  ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
  ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
  ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

  *** condition rules depending on <= relation
  ceq p /\ q = p if p <= q .
  ceq p \/ q = q if p <= q .
  ceq p /\ q = False if p <= ~ q .
  ceq p \/ q = True if ~ p <= q .
  ceq p U q = q if p <= q .
  ceq p R q = q if q <= p .
  ceq p U q = True U q if p =/= True /\ ~ q <= p .
  ceq p R q = False R q if p =/= False /\ q <= ~ p .
  ceq p U (q U r) = q U r if p <= q .
  ceq p R (q R r) = q R r if q <= p .
endfm
```

The classical method of constructing a Büchi automaton from an LTL formula in negative normal form is the tableau construction of a generalized

Büchi automaton (with multiple fairness conditions on states) followed by a counter-based conversion to a regular Büchi automaton [11]. Instead we used a newer technique, due to Gastin and Oddoux [9], based on very weak alternating automata. It consists of 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. After each step, the resulting automaton is optimized by removing unreachable states and then iteratively eliminating subsumed arcs and combining equivalent states until no further simplifications can be made.

We modify Gastin and Oddoux's approach slightly as follows. Throughout the computation, the pure propositional subformulae labeling the arcs of the various automata are stored as binary decision diagrams (BDDs) to allow computation of conjunctions and elimination of contradictions. This also allows us to combine parallel arcs by disjunction.

During the optimization of the generalized Büchi automaton we use some of the strongly connected component optimizations from [23], adapted to the case where fairness is defined on arcs rather than on states. We partition the generalized Büchi automaton into strongly connected components (SCCs). An SCC C is *fair* if each fairness condition is satisfied by at least one arc within C. An SCC C is *alive* if it is fair or if there is a arc from C to an alive SCC; otherwise C is *dead*. Dead SCCs and any arcs entering them can trivially be eliminated. Fairness information can be deleted from all arcs not lying within a fair SCC.

Let $<$ be an arbitrary linear ordering on the fairness conditions. A fairness condition α is *redundant* w.r.t. a fair SCC C iff there exists a another fairness condition β such that every arc lying within C that satisfies β also satisfies α and either $\alpha < \beta$ or there exists an arc that satisfies α but not β. If a fairness condition α is redundant w.r.t. all fair SCCs it can be deleted. Otherwise a fairness condition α that is redundant w.r.t. a fair SCC C can be added to every arc lying within C. We apply the subsumed arc elimination and equivalent state combination optimization step both before and after the strongly connected component optimizations.

5.2 Searching the Synchronous Product

We use the double depth first method of Holzmann et al. [12,13] to lazily search the synchronous product of our Büchi automaton and the state transition diagram. Although this method examines more states than the naive depth first algorithm for cycle detection, it avoids the costly examination of each cycle for fairness. We keep a full history of each system state generated, and for each such system state we keep five bit vectors to record:

(i) which propositions have been tested in the state;

(ii) which propositions were true in the state;

(iii) which product pairs (with automaton states) have been seen by the first depth first search;

(iv) which product pairs are currently on the first depth first search stack; and

(v) which product pairs have been seen by the second (nested) depth first search.

Of course we also need to keep to full term graph representation of each system state in order to test propositions; however this is not maintained in the core of the model checker (which contains no Maude-dependent code). Rather it is maintained in a separate hash table which also keeps track of rewrites between system states to avoid repeating work and is accessed via abstract state numbers. This outer, Maude-dependent layer also adds self loops to deadlocked states.

5.3 LTL Satisfiability Solving

The LTL satisfiability problem is decided as a by-product of the strongly connected component optimization of the generalized Büchi automaton in the Büchi automaton construction method sketched above. A formula is satisfiable if and only if in the derived generalized Büchi automaton there is a fair SCC C reachable from an initial state (in practice all non-reachable states will have already been eliminated).

In the negative case we just return *false*. In the positive case we need to compute a witness. We construct a shortest path from an initial state to a state $s \in C$ and find a fair cycle within C by finding shortest paths within C that satisfy additional fairness conditions until all fairness conditions are satisfied, followed by a shortest path back to s. All of the shortest paths are found by breadth first search. Also we try to shorten the path to C by folding the last part of it into the chosen cycle where possible. These heuristics seldom find the shortest path and fair cycle combination for our generalized Büchi automaton but seem useful enough in practice since our generalized Büchi automaton is not usually optimal in any sense.

For each arc with label v in the path and fair cycle we find a conjunction of literals that imply v by extracting a prime implicant from the BDD representation of v.

6 Performance Evaluation

SPIN [1], developed at Bell Labs in the formal methods and verification group, is a widely distributed formal verification tool. Some of its key features are: support for verification of distributed systems, on-the-fly nature, a built-in LTL model checker, and support for rendezvous and buffered message passing and shared memory. Maude also supports concurrent systems verification and has a fast LTL model checker. We compare the performance of the SPIN LTL model checker vis-a-vis the Maude LTL model checker.

SPIN uses a high-level language called PROMELA [10] to specify systems descriptions. We compare the performance of the 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. The results of such a comparison are given below.

The PROMELA specifications used are available from the main SPIN home page [1] as part of the distribution. Systems were chosen such that properties they satisfied were expressible using many operators. Peterson's solution to the mutual exclusion problem is a straightforward test of the basic features and speed of the model checkers; the property is one of global satisfaction of a predicate. Dolev et al.'s solution [1] to the leader election problem for a unidirectional ring network is an interesting example where properties involving different LTL operators, such as $\Box$, $\Diamond$ and $\mathcal{U}$, can be model checked. The third system is a mobile handoff scenario, involving an interesting LTL formula.

Except where stated, the default settings for SPIN were used everywhere. In all the above situations, only properties satisfied by the corresponding systems were model checked; no generation of counterexamples was attempted. The reason for this choice is that, since the two different model checkers implement two different search strategies, which model checker generates a counterexample first may vary from case to case depending on the particular search strategy of each tool, rather than on the speed of the model checker itself, which is what we are trying to estimate. A property that actually does hold for the specification forces the exploration of the entire synchronous product search space, resulting in more meaningful benchmarks.

The analyses were carried out on two machines, one a dual 1GHz Pentium III machine with 1GB RAM running Red Hat Linux 7.1 and the other a single 1.13 GHz Pentium III machine with 384MB RAM running Red Hat Linux 7.3. 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 in both cases. In all cases, the memory benchmark refers to the total memory footprint of the program, including the memory occupied by the code.

6.1 Peterson's Algorithm

Peterson's algorithm [21] is a simpler solution to the mutual exclusion problem than the algorithm originally proposed Dekker. The LTL property verified here is that the total number of processes in their critical sections is always either 0 or 1. This can be stated as $S \models \Box P$, where P holds true when the number of processes in their respective critical sections is 0 or 1. For the case $n = 5$, where n is the number of processes, both the SPIN and the Maude

Size[4]	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

model checkers ran out of memory.

6.2 Leader Election in a Unidirectional Ring

We compare the performance of Maude and SPIN for the algorithm described in [5]. Many different LTL properties can be verified here; for instance,

(i) $S \models \sim \Box\,(|\mathit{leaders}| = 0)$ (the number of leaders is not always zero.)

(ii) $S \models \Diamond\,(|\mathit{leaders}| > 0)$ (the number of leaders eventually becomes positive, i.e., the algorithm succeeds in electing somebody.)

(iii) $S \models \Diamond\,\Box\,(|\mathit{leaders}| = 1)$ (the number of leaders eventually settles down to unity.)

(iv) $S \models \Box\,((|\mathit{leaders}| = 0)\,\mathcal{U}\,(|\mathit{leaders}| = 1))$ (the "strong until" $\mathcal{U}$ signifies that the number of leaders remains zero up to a point when it becomes one, and once it becomes one it remains one.)

Here again, up to $n = 100$, where n is the number of nodes in the ring, both model checkers succeeded in proving all the four properties; however for the next case tried, $n = 200$, both model checkers ran out of memory for all four properties.

6.3 Mobile Handoff

This is a translation of the π-calculus description presented in [20]. The LTL property here is a progress property of the form $(\sim \Box\Diamond(r)) \to \Box(\Diamond p \to \Diamond q)$, where p, q and r are properties of the system. Further details are available in [20]. A PROMELA model of the system as well as a simplified version are available as part of the SPIN distribution.

[4] With 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

7 Conclusions

We have presented the Maude LTL model checker and its LTL satisfiability and tautology checkers. This tool opens up new application areas for model checking which are hard to specify in the system specification languages of existing model checking tools. This substantial widening in generality and scope of application areas has been achieved without sacrificing performance, which is comparable to that of current explicit-state model checkers with considerably more restricted system specification languages. 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 the-

orem proving support for proving such abstractions correct.

Acknowledgement

The first two authors' work has been partially supported by DARPA through Air Force Research Laboratory Contract F30602-97-C-0312, NSF grants CCR-9900326 and CCR-9900334, Office of Naval Research Contract N00012-99-C-0198, and DARPA through Air Force Research Laboratory Contract F30602-02-C-0130. The last two authors' work is also supported in part by the ONR Grant N00014-02-1-0715. We specially thank Miguel Palomino for finding a gap in an earlier proof of the Theorem in Appendix B and suggesting the correct proof. We thank the members of the Maude team for their help in the design of this tool and in particular Narciso Martí-Oliet and Alberto Verdejo for carefully reading the manuscript and suggesting improvements. We also thank Leonardo De Moura, Amir Pnueli, and Tomás Uribe for very helpful discussions on temporal logic and model checking issues, Kousha Etessami for clarifying for us one of the ideas in his paper [8], Sam Owre for clarifying several aspects of the SAL language, and the anonymous referees for their constructive criticism on an earlier version of this paper.

References

[1] Spin - Formal Verification. 2002.

[2] E. M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 2001.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, `http://maude.cs.uiuc.edu/`.

[4] P. M. Cohn. *Universal Algebra.* Harper & Row, 1965.

[5] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.

[6] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. L. Talcott. Pathway logic: Executable models of biological networks. This volume.

[7] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway logic: Symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, January 2002. `http://www.csl.sri.com/papers/lincoln-pathway-logic-psb-2002/`.

[8] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Eleventh International Conference on Concurrency Theory (CONCUR 2000)*, number 1877 in LNCS, pages 153–167. Springer-Verlag, 2000.

[9] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Thirteenth Conference on Computer Aided Verification (CAV '01)*, number 2102 in LNCS, pages 53–65. Springer-Verlag, 2001.

[10] R. Gerth. Concise Promela Reference. 1997.

[11] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[12] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *Proc. Second SPIN Workshop, August 1996, Vol. 32 of Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 81–89, Rutgers, Piscataway, NJ, 1997. American Mathematical Society.

[13] G J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *Design: An International Journal*, 13(3):289–307, nov 1998.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems*. Springer-Verlag, 1992.

[15] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[16] J. Meseguer and C. Talcott. Semantic models for distributed object reflection. In *Proceedings of ECOOP'02, Málaga, Spain, June 2002*, pages 1–36. Springer LNCS 2374, 2002.

[17] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[18] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, ed., *Proc. WADT'97*, 18–61, Springer LNCS 1376, 1998.

[19] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, pages 347–398. Springer-Verlag, 1999.

[20] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4(6):497–543, 1992.

[21] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, june 1981.

[22] N. Shankar. Symbolic analysis of transition systems. In Y. Gurevich, Ph. W. Kutter, M. Odersky, and L. Thiele, eds., *Abstract State Machines: Theory and Applications (ASM 2000)*, number 1912 in Lecture Notes in Computer Science, pages 287–302, Monte Verità, Switzerland, mar 2000. Springer-Verlag. `http://www.csl.sri.com/papers/asm2000/`.

[23] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Twelfth Conference on Computer Aided Verification (CAV '00)*, number 1633 in LNCS, pages 247–263. Springer-Verlag, 2000.

[24] M.-O. Stehr and C. L. Talcott. Plan in Maude—specifying an active network programming language. This volume.

[25] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.

A The Semantics of Dekker's Algorithm

```
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

***(Equality test comparing the contents of a named memory
location to a given machine integer.)

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

***(Syntax for a trival sequential programming language.
We can abstract certain terminating, or potentially
nonterminating, program fragments as constants of sorts
UserStatement and LoopingUserStatement.)

fmod SEQUENTIAL is
  inc TESTS .
  sorts UserStatement LoopingUserStatement Program .
  subsort LoopingUserStatement < 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

***(Processes have a process identifier and a program.
The machine state is a soup of processes, a shared memory and a
process identifier. The latter records the id of the last process
to execute and is needed to talk about fairness. The operational
semantics of the programming language running on this machine is given
by just 6 rules.  The first two rules deal with terminating and
potentially nonterminating user statements.)

mod PARALLEL is
  inc SEQUENTIAL .
  inc TESTS .

  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 .
  var L : LoopingUserStatement .
  vars I J : Pid .
  var M : Memory .
  var Q : Qid .
  vars N X : Int .
  var T : Test .

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

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


***(The classical Dekker's algorithm for mutual exclusion between two
processes using 3 variables, 'c1, 'c2 and 'turn, in shared memory.
crit is used to represent the critical section (which must be
terminating) and rem is used to represent the remainder (non-critical)
part of each program, which may be nonterminating.)

mod DEKKER is
  inc PARALLEL .
  subsort Int < Pid .
  op crit : -> UserStatement .
  op  rem : -> LoopingUserStatement .
  ops p1 p2 : -> Program .
  op initialMem : -> Memory .
  op initial : -> MachineState .

  eq p1 =
        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 .

  eq p2 =
        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 .

 eq initialMem = ['c1, 0] ['c2, 0] ['turn, 1] .
 eq initial = { [1, p1] | [2, p2], initialMem, 0 } .
endm
```

B Semantic LTL Equality

Theorem. The semantic LTL equality relation $\equiv_{LTL}$ is a substitution-closed Σ_{LTL}-congruence.

Proof: The proof that $\equiv_{LTL}$ is an equivalence relation follows easily from the LTL semantics of Boolean connectives, which gives us,

$$A, a, \pi \models_{LTL} \varphi \leftrightarrow \psi \quad \Leftrightarrow \quad (A, a, \pi \models_{LTL} \varphi \;\; \Leftrightarrow \;\; A, a, \pi \models_{LTL} \psi).$$

The above equivalence also gives an easy proof that $\equiv_{LTL}$ is a Σ_{LTL}-congruence[5]. The fact that $\equiv_{LTL}$ is substitution-closed, that is, that for each substitution $\theta : X \to LTL(X)$ we have, $\varphi \equiv_{LTL} \psi \;\Rightarrow \theta(\varphi) \equiv_{LTL} \theta(\psi)$ follows easily from the following, more general proposition (by applying it to the LTL formula $\phi = \varphi \leftrightarrow \psi$, which by hypothesis is a tautology). The proof of the proposition is by induction on the structure of LTL formulas.

Proposition. Let $\mathcal{A} = (A, \to_{\mathcal{A}}, L_{\mathcal{A}})$ be a Kripke structure on a set Y of atomic propositions, and let $\theta : X \to LTL(Y)$ be a substitution. Then, for each $\phi \in LTL(X)$, each path π in $\mathcal{A}$, and each natural number n we have,

$$\mathcal{A}, \pi(n), \pi \circ s^n \models_{LTL} \theta(\phi) \quad \Leftrightarrow \quad \theta_\pi(\mathcal{A}), \pi(n), \pi \circ s^n \models_{LTL} \phi.$$

where $\theta_\pi(\mathcal{A})$ is the Kripke structure on atomic propositions X with $\theta_\pi(\mathcal{A}) = (A, \to_{\mathcal{A}}, L_{\theta_\pi(\mathcal{A})})$, where $L_{\theta_\pi(\mathcal{A})} : A \longrightarrow \mathcal{P}(X)$ is defined on the elements $\pi(n)$ of π by, $L_{\theta_\pi(\mathcal{A})}(\pi(n)) = \{x \in X \mid \mathcal{A}, \pi(n), \pi \circ s^n \models_{LTL} \theta(x)\}$, and where for all other $a \in A$, $L_{\theta_\pi(\mathcal{A})}(a)$ can be an arbitrary set of propositions in X. q.e.d. q.e.d.

[5] Note, however, that, as shown in [14], adding past operators such as $\ominus$ would actually destroy the congruence property of $\equiv_{LTL}$.