The Maude LTL LBMC Tool Tutorial

Kyungmin Bae[1], Santiago Escobar[2], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign, IL, USA
[2] Universidad Politécnica de Valencia, Spain

Abstract. A concurrent system can be naturally specified as a rewrite theory. The Maude LTL logical bounded model checker (LBMC) tool is an LTL model checker for a rewrite theory, which can verify infinite-state systems using narrowing and folding relations, where the system's initial states are represented by terms with logical variables. This tutorial describes the main features and commands of the Maude LTL LBMC tool, illustrated by several examples.

1 Introduction

Rewriting is a very flexible formalism for specifying concurrent systems. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with equations E and rules R specifies a concurrent system whose states are axiomatized as the initial algebra $T_{\Sigma/E}$, and whose concurrent transitions are axiomatized by the rewrite rules R. Rewriting techniques can also be very useful for *model checking verification* of such systems, particularly when they are infinite-state. Specifically, *narrowing*[3] with rules R modulo the equations E offers many advantages as a technique for infinite-state model checking. For the case of *reachability analysis* this was shown in [8], and for the more general case of LTL *model checking* in [4].

A very appealing feature of narrowing-based model checking is the *logical* nature of its state space. That is, we do not represent *concrete states* (i.e., ground terms), but *state patterns*, that is, terms $t(X_1, \ldots, X_n)$ with *logical variables* $X_1, \ldots, X_n$. What $t(X_1, \ldots, X_n)$ stands for is not a single state, but the set of all concrete states that are its ground instances. This is very useful to deal with initial states which are not a single state but a (possibly infinite) set of concrete states, which can often be described by patterns $t(X_1, \ldots, X_n)$. Likewise, the states reached by narrowing are patterns with logical variables. As argued in [4], this logical state space already affords a huge abstraction, since each concrete state is abstracted by a pattern. An even greater abstraction (sometimes making the system finite-state) is afforded by *folding* the state space by means of a folding relation such as, for example, renaming or subsumption. In [4] we gave methods allowing LTL model checking verification when the folded logical state space is finite. But in general a folded logical state space need not be finite.

The Maude LTL logical bounded model checker (LBMC) tool is the first narrowing-based LTL model checker for infinite-state systems we are aware of.

[3] Narrowing [6] generalizes term rewriting by allowing free variables in terms and by performing unification instead of matching.

The concurrent systems it can analyze are rewrite theories $\mathcal{R} = (\Sigma, E, R)$ as the Maude system module [1] where the equational theory E has the finite variant property and satisfies the executability requirements in [2], and where the rules R are *topmost* (always rewrite at top positions). Our tool performs *iterated bounded model checking* on the (folded) logical state space to deal with possibly infinite logical states; only logical states that are reachable within a given depth from initial states are explored, and such a depth is iteratively incremented until a certain bound or until reaching a fixed-point if it exists. Using this method, one of four outcomes is possible: (i) a fixpoint (a finite state space) is reached and the formula is fully verified; (ii) no such fixpoint is reached and the formula is only verified up to a given bound; (iii) a real counterexample is found and reported; or (iv) a possibly spurious counterexample is found and reported.

2 The Maude LTL Logical Bounded Model Checker

The Maude LBMC tool provides a simple user interface in the interface file `symbolic-checker.maude` implemented by extending Full Maude. In order to execute the LBMC tool, we need the core Maude system compiled with the LBMC module, the Full Maude, and the user interface file. Similar to the existing Maude LTL model checker [3], the model checking process assumes a chosen state sort and given atomic propositions, which is defined by the module `SYMBOLIC-CHECKER` in the user interface file `symbolic-checker.maude`:

```
fmod SYMBOLIC-CHECKER is
  protecting QID .
  including SATISFACTION .
  including LTL .
  subsort Prop < Formula .

  sorts CEAssignment CESubstitution .
  subsort CEAssignment < CESubstitution .
  op _<-_ : Qid Universal -> CEAssignment [ctor poly (2) prec 63] .
  op none : -> CESubstitution .
  op _;_ : CESubstitution CESubstitution -> CESubstitution
    [ctor assoc comm id: none prec 65] .
  eq CA:CEAssignment ; CA:CEAssignment = CA:CEAssignment .

  sorts Transition TransitionList ModelCheckResult .
  subsort Transition < TransitionList .
  subsort Bool < ModelCheckResult .
  op {_,_} : State CESubstitution -> Transition [ctor] .
  op nil : -> TransitionList [ctor] .
  op __ : TransitionList TransitionList -> TransitionList
    [ctor assoc id: nil format (d ni d)] .
  op prefix_loop_ : TransitionList TransitionList -> ModelCheckResult
    [ctor format(n n++i n ni i--)] .
endfm
```

Note that the module `SYMBOLIC-CHECKER` includes the module `LTL` for the LTL syntax declarations, and the module `SATISFACTION` for the state and the proposition sorts, where both are defined in the standard Maude LTL model checker.

The sort `ModelCheckResult` in the module `SYMBOLIC-CHECKER` defines the signature of a counterexample from the LTL LBMC model checking, which also includes the boolean sort `Bool`. A counterexample of an LTL formula is an infinite path consisting of two transition lists, where the first one is a finite prefix and the second one is a cycle that forms an infinite path. Each transition is a pair of a logical state with sort `State` and a substitution with sort `CESubstitution` for a narrowing transition. If the given LTL property is a certain class of safety properties in which every counterexample has an informative prefix as explained in [7], our LBMC tool reports a finite counterexample where the second transition list for the cycle is the constant `nil`. For example, the LBMC for an invariant property will always generate a finite counterexample.

Given a chosen kind `[Top]` with a system module `M`, the state propositions for the LBMC are typically defined as the following general pattern in Full Maude after loading the interface file `symbolic-checker.maude`:

```
load symbolic-checker .

(mod M-PRES is
  protecting M .
  including SYMBOLIC-CHECKER .

  subsort Top < State .
  ops prop_1 ... prop_k : -> Prop .
  eq StatePattern |= prop_1 = ValuePattern [variant] .
  ...
endm)
```

The sort `State` is the state sort for the logical model checking, and each state proposition has sort `Prop`. The semantics of each spate proposition is defined by the equations on the operator `_|=_: State Prop -> Bool`, which is defined in the module `SATISFACTION`. Such satisfaction equations should have the finite variant property so that they satisfy the executability requirements in [2]. The equation attribute `[variant]` declares that the corresponding equations have the finite variant property so that variant-based equational unification [5] is used for state propositions. The LBMC tool will only consider the equations with the `[variant]` attribute for the model checking verification.

There are two commands `lmc` [`n`] t `|=` φ and `lfmc` [`n`] t `|=` φ for logical model checking an LTL formula φ from an initial state t with the maximum bound n. Each command uses a different folding relation: the *renaming equivalence* relation for the `lmc` command, and the *subsumption* relation for the `lfmc` command. If a maximum bound n is not specified in the command, infinity is considered as the bound. Note that if we use the renaming equivalence as a folding relation (the `lmc` command), then there are no spurious counter examples but it is more likely not to reach a fixed point, while the subsumption folding relation (the

`lfmc` command) may generate a spurious counterexample. Our LMBC tool also provides the capability to specify the initial pattern constraints given by boolean functions, where the semantics of such boolean functions should be declared by the equations satisfying finite variant property [5]. The initial constraints are given by the postfix "`such that` *Cond*" in logical model checking commands. For example, if a boolean function for a condition is `initCond : S -> Bool` and all the equations for `initCond` has the `[variant]` attribute, we can give a conditional LBMC command such as

`lmc [n] t(X:S) |=` φ `such that initCond(X:S)`

In the rest of this paper, we illustrate the LBMC tool with several examples.

3 Example: Lamport's Bakery Protocol

This section illustrates the LTL logical bounded model checker (LMBC) tool with Lamport's bakery protocol, where the Maude declarations are borrowed from [4]. The Maude functional module `BAKERY-SYNTAX` below describe the states of the bakery protocol as terms of the form i ; j ; $[k_1, m_1] \ldots [k_n, m_n]$ with sort `Conf`, where i is the current number in the bakery's number dispenser, j is the number currently served, and the $[k_1, m_1] \ldots [k_n, m_n]$ are a set of customer processes, each with an identity k_l and in a *mode* m_l, which can be either `idle` (has not yet picked a number), or `wait(n)` (waiting with number `n`), or `crit(n)` (being served with number `n`).

```
(fmod BAKERY-SYNTAX is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  sorts ModeIdle ModeWait ModeCrit Mode .
  subsorts ModeIdle ModeWait ModeCrit < Mode .
  sorts ProcIdle ProcWait Proc ProcIdleSet ProcWaitSet ProcSet .
  subsorts ProcIdle < ProcIdleSet .
  subsorts ProcWait < ProcWaitSet .
  subsorts ProcIdle ProcWait < Proc < ProcSet .
  subsorts ProcIdleSet < ProcWaitSet < ProcSet .
  op idle : -> ModeIdle .
  op wait : Nat -> ModeWait .
  op crit : Nat -> ModeCrit .
  op `[_`,_`] : Nat ModeIdle -> ProcIdle .
  op `[_`,_`] : Nat ModeWait -> ProcWait .
  op `[_`,_`] : Nat Mode -> Proc .
  op none : -> ProcIdleSet .
  op __ : ProcIdleSet ProcIdleSet -> ProcIdleSet [assoc comm id: none] .
  op __ : ProcWaitSet ProcWaitSet -> ProcWaitSet [assoc comm id: none] .
  op __ : ProcSet ProcSet -> ProcSet [assoc comm id: none] .
  sort Conf .
  op _;_;_ : Nat Nat ProcSet -> Conf .
endfm)
```

The three rewrite rules in the system module BAKERY below describes the behavior of the system; the picking of a number, being served, and leaving. Those rules are topmost because they rewrite entire configurations. Note that this system is infinite-state in two ways: (i) the counters i and j are unbounded; and (ii) the number n of customer processes is also unbounded.

```
(mod BAKERY is
  protecting BAKERY-SYNTAX .

  var PS : ProcSet .  vars N M K : Nat .
  rl [wake] : N ; M ; [K, idle] PS  =>  s(N) ; M ; [K, wait(N)] PS .
  rl [crit] : N ; M ; [K, wait(M)] PS  =>  N ; M ; [K, crit(M)] PS .
  rl [exit] : N ; M ; [K, crit(M)] PS  =>  N ; s(M) ; [K, idle] PS .
endm)
```

In the following system module, the atomic proposition ex? specifies the mutual exclusion property of a single state for the Bakery system. The sort Nat in the module SYMBOLIC-CHECKER was renamed to the new name Nat' in BAKERY-SAFETY-SATISFACTION in order to avoid the name conflict with BAKERY.

```
(mod BAKERY-SAFETY-SATISFACTION is
  pr BAKERY .
  pr SYMBOLIC-CHECKER * (sort Nat to Nat') .
  subsort Conf < State .
  ops ex? : -> Prop .

  var WS : ProcWaitSet . var PS : ProcSet . vars N M M1 M2 K1 K2 : Nat .
  eq N ; M ; WS |= ex? = true [variant] .
  eq N ; M ; [K1, crit(M1)] WS |= ex? = true [variant] .
  eq N ; M ; [K1, crit(M1)] [K2, crit(M2)] PS |= ex?
   = false [variant] .
endm)
```

The following logical bounded model checking command verifies that the mutual execution property □ex? is satisfied from any initial state with the pattern N ; N ; [0,idle] [s(0),idle] within the bound 10:

```
Maude> (lmc [10] N:Nat ; N:Nat ; [0, idle] [s(0), idle] |= [] ex? .)
logical model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ; [0,idle] [s(0),idle] |= [] ex?
result:
  no counterexample found within bound 10
```

In this case, if the bound is not specified, the logical model checking command does not terminate , since the subsumption folding relation is not used. Instead, when the subsumption folding relation is applied with the lbmc command, the property can be verified from the initial pattern N ; N ; [0,idle] [s(0),idle] for the Bakery system as follows:

```
Maude> (lfmc N:Nat ; N:Nat ; [0, idle] [s(0), idle] |= [] ex? .)
logical folding model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ; [0,idle] [s(0),idle] |= [] ex?
result:
  true (complete with depth 4)
```

In the above, the computation of the folding graph for the Bakery system from the initial pattern `N ; N ; [0,idle] [s(0),idle]` reaches a fixed point.

However, when the initial pattern is given by `N ; N ; IS:ProcIdleSet`, the bound should be specified for the termination even with the subsumption folding relation, since the folding logical approximation for the initial state is infinite:

```
Maude> (lfmc [50] N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex? .)
logical folding model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result:
  no counterexample found within bound 50
```

Besides safety properties, it is also possible to verify liveness properties using our tool. The following module declares the atomic proposition `ever-wait?` (resp., `ever-crit?`) expressing that at least one process in the Bakery system has been in its waiting (resp. critical) state since the initial configuration.

```
(mod BAKERY-LIVENESS-SATISFACTION is
  pr BAKERY .
  pr SYMBOLIC-CHECKER * (sort Nat to Nat') .
  subsort Conf < State .
  ops ever-wait? ever-crit? : -> Prop .

  vars N M : Nat . vars PS : ProcSet .
  eq s(N) ; M ; PS |= ever-wait? = true [variant] .
  eq 0 ; M ; PS    |= ever-wait? = false [variant] .
  eq N ; s(M) ; PS |= ever-crit? = true [variant] .
  eq N ; 0 ; PS    |= ever-crit? = false [variant] .
endm)
```

Using our LBMC tool, we can then model check a liveness property given by the LTL formula $\Box$(`ever-wait?` $\rightarrow$ $\Diamond$ `ever-crit?`) from the initial pattern `N ; N ; [0,idle] [s(0),idle]` as follows:

```
Maude> (lfmc N ; N ; [0, idle] [s(0), idle]
          |= [] (ever-wait? -> <> ever-crit?) .)
logical folding model check in BAKERY-LIVENESS-SATISFACTION :
  N:Nat ; N:Nat ; [0,idle] [s(0),idle] |=
  [] (ever-wait? -> <> ever-crit?)
result:
  true (complete with 4)
```

If a counterexample is found, the model checker reports it. The following is the logical model checking result of the formula $\bigcirc\Diamond\neg$`ex?` from the initial pattern `N ; N ; [0,idle] [s(0),idle]` with the renaming equivalence relation:

```
Maude> (lmc N:Nat ; N:Nat ; [0, idle] [s(0), idle] |= 0 []~ ex? .)
logical model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ;[0,idle][s(0),idle] |= 0[]~ ex?
result:
  counterexample found at depth 1

prefix
  {N:Nat ; N:Nat ;[0,idle][s(0),idle],none}
  {s(N:Nat); N:Nat ;[0,idle][s(0),wait(N:Nat)],none}
loop
  nil
```

In the above, the counterexample is a finite path without a loop since the formula is a safety property and a finite erroneous path was found.

If the renaming equivalence relation is used for folding, counterexamples are never spurious; but a counterexample in the system with the subsumption folding relation may be spurious. The Maude LBMC tool then tries to construct an actual counterexample from the abstracted counterexample within the resulting bound, and reports it when such a actual counterexample is found:

```
Maude> (lfmc N:Nat ; N:Nat ; [0, idle] [s(0), idle] |= 0 []~ ex? .)
logical folding model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ;[0,idle][s(0),idle] |= 0[]~ ex?
result:
  counterexample found at depth 1

prefix
  {N:Nat ; N:Nat ;[0,idle][s(0),idle],none}
  {s(N:Nat); N:Nat ;[0,idle][s(0),wait(N:Nat)],none}
loop
  nil
```

Otherwise, the tool returns the abstracted counterexample generated from logical model checking, and reports that the counterexample is possibly spurious.

```
Maude> (lfmc N:Nat ; N:Nat ; [0, idle] [s(0), idle] |= <> ~ ex? .)
logical folding model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ;[0,idle][s(0),idle] |= <> ~ ex?
result:
  possibly spurious counterexample found at depth 3

prefix
  nil
loop
  {N:Nat ; N:Nat ;[0,idle][s(0),idle],none}
  {(s(N:Nat); N:Nat ;[0,idle][s(0),wait(N:Nat)],none}
  {(s(N:Nat); N:Nat ;[0,idle][s(0),crit(N:Nat)],none}
```

Note that the above counterexample is an infinite path since the formula is a liveness property. The following result shows another possibly spurious counterexample from a safety property.

```
Maude> (lfmc N ; N ; WS:ProcWaitSet |= [] (ever-wait? -> ever-crit?) .)
logical folding model check in BAKERY-LIVENESS-SATISFACTION :
  N:Nat ; N:Nat ; WS:ProcWaitSet |= [](ever-wait? -> ever-crit?)
result:
  possibly spurious counterexample found at depth 3

prefix
  {0 ; 0 ; WS:ProcWaitSet,
    'WS <- #7:ProcWaitSet[#3:Nat,idle]}
  {s(0) ; 0 ; #7:ProcWaitSet[#3:Nat,wait(0)],
    '#7 <- #15:ProcWaitSet[#10:Nat,idle]}
loop
  {s(s(0)); 0 ; #15:ProcWaitSet[#3:Nat,wait(0)][#10:Nat,wait(s(0))],
    '#15 <-[#18:Nat,idle]}
```

Finally, we can specify initial pattern constraints for the LBMC by means of boolean functions, where the semantics of such boolean functions should be declared by the equations with finite variant property [5], such as:

```
op initCond : ProcWaitSet -> Bool .
eq initCond([K1, wait(M)] [K2, wait(M)] WS) = false [variant] .
eq initCond([K1, wait(M)] IS) = true [variant] .
eq initCond(IS) = true [variant] .
```

For example, the following is the logical model checking result for the formula □ex? from the initial pattern `N:Nat ; M:Nat ; WS:ProcWaitSet` with the constraints `initCond(WS) = true`:

```
Maude> (lfmc N:Nat ; M:Nat ; WS:ProcWaitSet |= []ex?
            such that initCond(WS:ProcWaitSet) .)
logical folding model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; M:Nat ; WS:ProcWaitSet |= []ex?
under the condition :
  initCond(WS:ProcWaitSet)= true
result:
  counterexample found at depth 4

prefix
  {N:Nat ; M:Nat ; WS:ProcWaitSet,
    'WS <- #9:ProcWaitSet[#3:Nat,wait(M:Nat)]}
  {N:Nat ; M:Nat ; #9:ProcWaitSet[#3:Nat,crit(M:Nat)],
    '#9 <-[#12:Nat,idle]}
  {s(N:Nat); M:Nat ;[#3:Nat,crit(M:Nat)][#12:Nat,wait(N:Nat)],
    'N <- M:Nat}
  {s(M:Nat); M:Nat ;[#3:Nat,crit(M:Nat)][#12:Nat,crit(M:Nat)],none}
loop
  nil
```

4 Example: Readers-Writers Problem

This section shows the LTL logical bounded model checking for the readers-writers problem whose Maude specification is from [1]. A state is represented by a tuple `< R, W >` that indicates the number `R` of readers and the number `W` of writers accessing a critical resource. Readers and writers can leave the resource at any time, but writers can only access it if nobody else is using it, and readers only if there are no writers.

```
mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  sort Config .
  op <_‘,_> : Nat Nat -> Config [ctor] .

  vars R W : Nat .
  rl < 0, 0 >     =>  < 0, s(0) > .
  rl < R, s(W) >  =>  < R, W > .
  rl < R, 0 >     =>  < s(R), 0 > .
  rl < s(R), W >  =>  < R, W > .
endm
```

There are two important properties to be verified for this system: (i) at most one writer will be able to access the resource at any given time, and (ii) readers and writers never access the resource simultaneously. The following module specifies the necessary state propositions for those properties in addition to the state sort declaration, with loading the LBMC interface file `symbolic-checker`.

```
load symbolic-checker .

(mod READERS-WRITERS-PROPS is
  pr READERS-WRITERS .
  pr SYMBOLIC-CHECKER * (sort Nat to Nat') .
  subsort Config < State .
  ops one-writer-atmost? exclusion? : -> Prop .

  vars N M : Nat .
  eq < N, 0       > |= one-writer-atmost? = true [variant] .
  eq < N, s(0)    > |= one-writer-atmost? = true [variant] .
  eq < N, s(s(M)) > |= one-writer-atmost? = false [variant] .
  eq < 0,    M    > |= exclusion? = true [variant] .
  eq < N,    0    > |= exclusion? = true [variant] .
  eq < s(N), s(M) > |= exclusion? = false [variant] .
endm)
```

Note that each equation in the module `READERS-WRITERS-PROPS` has the attribute `[variant]`, and the sort `Nat` in the module `SYMBOLIC-CHECKER` was renamed to the new name `Nat'` in `READERS-WRITERS-PROPS` in order to avoid the name conflict with the module `READERS-WRITERS`.

The following shows the logical bounded model checking result of the formula `[] one-writer-atmost?` from the initial pattern `< N, 0 >` with the bound 10 under the renaming equivalence folding relation.

```
Maude> (lmc [10] < N:Nat,0 > |= [] one-writer-atmost? .)
logical model check in READERS-WRITERS-PROPS :
  < N:Nat,0 > |= []one-writer-atmost?
result:
  no counterexample found within bound 10
```

For the mutual exclusion property `[] exclusion?`, the LBMC result with bound 10 under the renaming equivalence folding relation is as follows:

```
Maude> (lmc [10] < N:Nat,0 > |= [] exclusion? .)
logical model check in READERS-WRITERS-PROPS :
  < N:Nat,0 > |= []exclusion?
result:
  no counterexample found within bound 10
```

Using the subsumption folding relation with the `lfmc` command, we can verify both the formula `[] one-writer-atmost?` and `[] exclusion?` from the pattern `< N, 0 >` as follows:

```
Maude> (lfmc < N:Nat,0 > |= []one-writer-atmost? .)
logical folding model check in READERS-WRITERS-PROPS :
  < N:Nat,0 > |= []one-writer-atmost?
result:
  true (complete with depth 3)

Maude> (lfmc < N:Nat,0 > |= [] exclusion? .)
logical folding model check in READERS-WRITERS-PROPS :
  < N:Nat,0 > |= []exclusion?
result:
  true (complete with depth 3)
```

For the faulty property `<> ~ one-writer-atmost?`, the LBMC model checking from the initial pattern `< N, 0 >` with the subsumption folding relation generates a possibly spurious counterexample, so that it cannot disprove the property.

```
Maude> (lfmc < N:Nat,0 > |= <> ~ one-writer-atmost? .)
logical folding model check in READERS-WRITERS-PROPS :
  < N:Nat,0 > |= <> ~ one-writer-atmost?
result:
  possibly spurious counterexample found at depth 3

prefix
  nil
loop
  {< N:Nat,0 >,'N <- s(#1:Nat)}
  {< #1:Nat,0 >,'#1 <-(0).Nat}
  {< 0,s(0)>,none}
```

However, we can use then the renaming equivalence relation with `lmc` command to generate a real counterexample from the initial pattern `< N, 0 >` so as to disprove the faulty property `<> ~ one-writer-atmost?` as follows:

```
Maude> (lmc < N:Nat,0 > |= <> ~ one-writer-atmost? .)
logical model check in READERS-WRITERS-PROPS :
  < N:Nat,0 > |= <> ~ one-writer-atmost?
result:
  counterexample found at depth 4

prefix
  nil
loop
  {< N:Nat,0 >,'N <- s(#1:Nat)}
  {< #1:Nat,0 >,'#1 <- s(#2:Nat)}
```

References

1. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude – A High-Performance Logical Framework*. Springer LNCS Vol. 4350, 2007.
2. F. Durán, S. Eker, S. Escobar, J. Meseguer, and C. Talcott. Variants, unification, narrowing, and symbolic reachability in maude 2.6. In *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10, pages 31–40. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.
3. Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc.* 10^{th} *Intl. SPIN Workshop*, volume 2648, pages 230–234. Springer LNCS, 2003.
4. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Proceedings of the 18th international conference on Term rewriting and applications (RTA'07)*, pages 153–168. Springer-Verlag, 2007.
5. Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *J. Log. Algebr. Program.*, 2011. In Press.
6. Jean-Marie Hullot. Canonical forms and unification. In *CADE*, LNCS vol. 87, pages 318–334. Springer, 1980.
7. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
8. José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.