Program Verification: Lecture 20

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Decidability of Propositional LTL

It is well-known that, for any computable Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$, any state $a \in A$ such that the set

$$Reach_{\mathcal{A}}(a) = \{x \in A \mid \exists \pi \in Path(\mathcal{A})\ \exists n \in \mathbb{N}\ \ s.t.\ \pi(0) = a \wedge \pi(n) = x\}$$

of states reachable from a in $\mathcal{A}$ is finite, and any LTL formula $\varphi \in LTL(AP)$, where $L : A \longrightarrow \mathcal{P}(AP)$, there is a decision procedure that can effectively decide the satisfaction relation,

$$\mathcal{A}, a \models_{LTL} \varphi.$$

Furthermore, if $\mathcal{A}, a \not\models_{LTL} \varphi$, the decision procedure will exhibit a counterexample, that is, a path not satisfying φ.

Decidability of Propositional LTL (II)

A decision procedure of this kind is called a model checking algorithm, since it checks whether φ holds in the model $\mathcal{A}$ with initial state a. Detailed discussion of such algorithms for a variety of temporal logics such as LTL, CTL, and CTL^* is beyond the scope of this course; see the excellent text "Model Checking" by Clark, Grumberg, and Peled. There are two rough classes of model checking algorithms:

- explicit-state model checking algorithms, that explicitly search the state space of $\mathcal{A}$ to find a counterexample;
- symbolic model checking algoritms, that use a symbolic representation of sets of states (BDDs or other representations) to compute the fixpoint of the transition relation, i.e., the set $Reach_{\mathcal{A}}(a)$.

The Maude Model Checker

Suppose that, given a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, we have:

- chosen a kind k in `M` as our kind of states;
- defined some state predicates Π and their semantics in a module, say `M-PREDS`, protecting `M` by the method already explained in this lecture.

Then, as explained earlier, this defines a Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\Pi}$ on the set of atomic propositions AP_{Π}. Given an initial state $[t] \in T_{\Sigma/E,k}$ and an LTL formula $\varphi \in LTL(AP_{\Pi})$ we would like to have a procedure to decide the satisfaction relation,

The Maude Model Checker (II)

$$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi.$$

By applying the general LTL decidability results to our Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$, this satisfaction relation becomes decidable if two conditions hold:

1. The set of states in $T_{\Sigma/E,k}$ that are reachable from $[t]$ by rewriting is finite.

2. The rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by `M` plus the equations D defining the predicates Π are such that:

The Maude Model Checker (III)

- both E and $E \cup D$ are (ground) Church-Rosser and terminating, perhaps modulo some axioms A, and

- R is (ground) coherent relative to E (again, perhaps modulo some axioms A).

Under these assumptions, both the state predicates Π and the transition relation $\rightarrow^1_{\mathcal{R}}$ are computable and, given the finite reachability assumption, we can then settle the above satisfaction problem using a model checking procedure. Specifically, Maude uses an on-the-fly LTL model checking procedure of the style described by Clark, Grumberg, and Peled.

The Maude Model Checker (III)

The basis of this procedure is the following. Each LTL formula φ has an associated Büchi automaton B_φ whose acceptance ω-language is exactly that of the traces satisfying φ. We can then reduce the satisfaction problem

$$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$$

to the emptiness problem of the language accepted by the synchronous product of $B_{\neg\varphi}$ and (the Büchi automaton associated to) $(\mathcal{K}(\mathcal{R}, k)_\Pi, [t])$. The formula φ is satisfied iff such a language is empty. The model checking procedure checks emptiness by looking for a counterexample, that is, an infinite computation belonging to the language recognized by the synchronous product.

The Maude Model Checker (IV)

This makes clear our interest in obtaining the negative normal form of a formula $\neg\varphi$, since we need it to build the Büchi automaton $B_{\neg\varphi}$.

For efficiency purposes we need to make $B_{\neg\varphi}$ as small as possible. The following module `LTL-SIMPLIFIER` (also in the `model-checker.maude` file) tries to further simplify the negative normal form of the formula $\neg\varphi$ in the hope of generating a smaller Büchi automaton $B_{\neg\varphi}$. This module is optional (the user may choose to include it or not when doing model checking) but tends to help building a smaller $B_{\neg\varphi}$.

The Maude Model Checker (V)

```
fmod LTL-SIMPLIFIER is
  including LTL .

  *** The simplifier is based on:
  ***   Kousha Etessami and Gerard J. Holzman,
  ***   "Optimizing Buchi Automata", p153-167, CONCUR 2000, LNCS 1877.
  *** We use the Maude sort system to do much of the work.

  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
  PE-Formula PU-Formula < Formula .

  op True : -> TrueFormula [ctor ditto] .
  op False : -> FalseFormula [ctor ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
```

```
op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op O_ : PE-Formula -> PE-Formula [ctor ditto] .
op O_ : PU-Formula -> PU-Formula [ctor ditto] .
op O_ : PureFormula -> PureFormula [ctor ditto] .
op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .

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 .

*** We also use the rules from:
***   Fabio Somenzi and Roderick Bloem,
*** "Efficient Buchi Automata from LTL Formulae",
***   p247-263, CAV 2000, LNCS 1633.
*** that are not subsumed by the previous system.
```

```
*** 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 formula
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 Maude Model Checker (VI)

Suppose that all the requirements listed above to perform model checking are satisfied. How do we then model check a given LTL formula in Maude for a given initial state $[t]$ in a module `M`? We define a new module, say `M-CHECK`, according to the following pattern:

```
mod M-CHECK is
  protecting M-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER . *** optional
  op init : -> k .           *** optional
  eq init = t .              *** optional
endm
```

The declaration of a constant `init` of the kind of states is not necessary: it is a matter of convenience, since the initial state `t` may be a large term.

The Maude Model Checker (VII)

The module `MODEL-CHECKER` is as follows.

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

*** transitions and results
sorts RuleName Transition TransitionList ModelCheckResult .
subsort Qid < RuleName .
subsort Transition < TransitionList .
subsort Bool < ModelCheckResult .
ops unlabeled deadlock : -> RuleName .
op {_,_} : State RuleName -> Transition [ctor] .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList [ctor assoc id: nil] .
op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult [special ( ... )] .
endfm
```

The Maude Model Checker (VIII)

Its key operator is `modelCheck` (whose `special` attribute has been omitted here), which takes a state and an LTL formula and returns either the Boolean `true` if the formula is satisfied, or a counterexample when it is not satisfied.

Let us illustrate the use of this operator with our `MUTEX` example. Following the pattern described above, we can define the module

```
mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a,wait] [b,wait] .
  eq initial2 = * [a,wait] [b,wait] .
endm
```

The Maude Model Checker (X)

We are then ready to model check different LTL properties of `MUTEX`. The first obvious property to check is mutual exclusion:

```
Maude> red modelCheck(initial1,[] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []~ (crit(a) /\ crit(b))) .
rewrites: 18 in 10ms cpu (10ms real) (1800 rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,[] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []~ (crit(a) /\ crit(b))) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The Maude Model Checker (XII)

We can also model check the strong liveness property that if a process waits infinitely often, then it is in its critical section infinitely often:

```
Maude> red modelCheck(initial1,([] <> wait(a)) -> ([] <> crit(a))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(a) -> []<> crit(a)) .
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial1,([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(b) -> []<> crit(b)) .
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,([] <> wait(a)) -> ([] <> crit(a))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(a) -> []<> crit(a)) .
rewrites: 68 in 10ms cpu (10ms real) (6800 rewrites/second)
```

```
result Bool: true

Maude> red modelCheck(initial2,([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(b) -> []<> crit(b)) .
rewrites: 68 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The Maude Model Checker (XIII)

Of course, not all properties are true. Therefore, instead of a success we can get a counterexample showing why a property fails. Suppose that we want to check whether, beginning in the state `initial1`, process `b` will always be waiting. We then get the counterexample:

```
Maude> red modelCheck(initial1,[] wait(b)) .
reduce in MUTEX-CHECK : modelCheck(initial1, []wait(b)) .
rewrites: 14 in 10ms cpu (10ms real) (1400 rewrites/second)
result ModelCheckResult:
  counterexample({$ [a,wait] [b,wait],'a-enter}
                 {[a,critical] [b,wait],'a-exit}
                 {* [a,wait] [b,wait],'b-enter},
                 {[a,wait] [b,critical],'b-exit}
                 {$ [a,wait] [b,wait],'a-enter}
                 {[a,critical] [b,wait],'a-exit}
                 {* [a,wait] [b,wait],'b-enter})
```

The Maude Model Checker (XIV)

The main counterexample term constructors are:

```
op {_,_} : State RuleName -> Transition .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList [ctor assoc id: nil]
op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor]
```

A counterexample is a pair consisting of two lists of transitions: the first is a finite path beginning in the initial state, and the second describes a loop. 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. Note that each transition is represented as a pair, consisting of a state and the label of the rule applied to reach the next state.

Model Checking TOK-RING

Consider the following TOK-RING module,

```
(fth NZNAT* is
   protecting NAT .
   op * : -> NzNat .
 endfth)

(fmod NAT/{N :: NZNAT*} is
   sort Nat/{N} .
   op `[_`] : Nat -> Nat/{N} .
   op _+_ : Nat/{N} Nat/{N} -> Nat/{N} .
   op _*_ : Nat/{N} Nat/{N} -> Nat/{N} .
   vars I J : Nat .
   ceq [I] = [I rem *] if I >= * .
   eq [I] + [J] = [I + J] .
   eq [I] * [J] = [I * J] .
 endfm)
```

```
(omod TOK-RING{N :: NZNAT*) is
   protecting NAT/{N} .
   sort Mode .
   subsort Nat/{N} < Oid .
   ops wait critical : -> Mode .
   msg tok : Nat/{N} -> Msg .
   op init : -> Configuration .
   op make-init : Nat/{N} -> Configuration .
   class Proc | mode : Mode .
   var I : Nat .
   ceq init = tok([0]) make-init([I]) if s(I) := * .
   ceq make-init([s(I)])
     = < [s(I)] : Proc | mode : wait > make-init([I])
     if I < * .
   eq make-init([0]) = < [0] : Proc | mode : wait > .
   rl [enter] : tok([I]) < [I] : Proc | mode : wait >
     => < [I] : Proc | mode : critical > .
   rl [exit] : < [I] : Proc | mode : critical >
     => < [I] : Proc | mode : wait > tok([s(I)]) .
 endom)
```

Model Checking **TOK-RING** (II)

The `TOK-RING` module satisfies the following two properties:

- mutual exclusion, and
- guaranteed reentrance, that is:
  - each process eventually reaches its critical section, and
  - it does so again after $2 \times n$ steps.

There isn't a single LTL formula stating each of these properties: they are parametric on n. However, in Full Maude we can specify these properties by parametic formula definitions as follows:

Model Checking **TOK-RING** (III)

```
(omod CHECK-TOK-RING{N :: NZNAT*} is
  inc TOK-RING{N} .
  inc MODEL-CHECKER .
  subsort Configuration < State .

  op inCrit : Nat/{N} -> Prop .
  op twoInCrit : -> Prop .

  var  I : Nat .
  vars X Y : Nat/{N} .
  var  C : Configuration .
  var  F : Formula .

  eq < X : Proc | mode : critical > C |= inCrit(X) = true .
  eq < X : Proc | mode : critical > < Y : Proc | mode : critical > C
       |= twoInCrit = true .
```

```
 op guaranteedReentrance : -> Formula .
 op allProcessesReenter : Nat -> Formula .
 op nextIter_ : Formula -> Formula .
 op nextIterAux : Nat Formula -> Formula .

 ceq guaranteedReentrance = allProcessesReenter(I) if s(I) := * .

 eq allProcessesReenter(s(I))
   = (<> inCrit([s(I)])) /\
      [] (inCrit([s(I)]) -> (nextIter inCrit([s(I)]))) /\
     allProcessesReenter(I) .
 eq allProcessesReenter(0) = (<> inCrit([0])) /\
      [] (inCrit([0]) -> (nextIter inCrit([0]))) .

 eq nextIter F = nextIterAux(2 * *, F) .
 eq nextIterAux(s I, F) = O nextIterAux(I, F) .
 eq nextIterAux(0, F) = F .

endom)
```

Model Checking **TOK-RING** (IV)

We cannot model check these properties directly in their parameterized form. However, for each nozero value n we can check the corresponding instance of these properties. For example, for $n = 5$ we define in Full Maude the view,

```
(view 5 from NZNAT* to NAT is
   op * to term 5 .
 endv)
```

Then we can model check the mutual exclusion property for 5 processes as follows:

```
(red in CHECK-TOK-RING{5} : modelCheck(init,[] ~ twoInCrit) .)
result Bool :
  true
```

Model Checking `TOK-RING` (V)

In the same way, we can model check the guaranteed reentrance property for $n = 5$ by giving to Full Maude the command,

```
(red in CHECK-TOK-RING(5) : modelCheck(init,[] guaranteedReentrance) .)
result Bool :
  true
```