Appendix 2 to Lecture 22: LTL Model Checking for **`PARALLEL`**

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

LTL Model Checking for **PARALLEL**

Lecture 19 illustrated by means of `PARALLEL` how a rewriting semantics $\mathcal{R}_\mathcal{L}$ for a concurrent imperative language $\mathcal{L}$, $\mathcal{R}_\mathcal{L}$ specified in Maude automatically endows $\mathcal{L}$ with a parser, an interpreter and a model checker for invariants using Maude's `search` command.

LTL allows us to specify additional properties. In particular, so-called liveness properties (including fairness properties), that intrinsically require infinite behaviors in order to be defined and fall out of the scope of `search`-based model checking.

This Appendix will serve two purposes. It will first illustrate a general method to endow any concurrent language $\mathcal{L}$ having a rewriting logic semantics with an LTL model checker for $\mathcal{L}$ in Maude. Second, it will also allow us to see some interesting fairness properties specified in LTL.

The Rewriting Semantics of `PARALLEL`: A Slight Change

The rewriting semantics of `PARALLEL` was given in Lecture 19 and will not be repeated here. There is, however, a slight change in its semantics necessitated by the fact that LTL is a state-based temporal logic. That is, state predicated $p \in \Pi$ define unary predicates on the states.

There is for this reason a mismatch between fairness properties, which typically are about actions, i.e., transitions, and not about states. This requires sometimes to slightly modify the specification to represent in the state itself some actions.

The slight modification needed (replacing the state constructor `{_,_}` by `{_,_,_}`) to record the actions of processes, can be best understood by comparing the rules for `PARALLEL` with those given here:

The Rewrite Rules for **PARALLEL** in Lecture 19

```
mod PARALLEL is
  inc SEQUENTIAL .
  inc TESTS .

  sorts Pid Process Soup MachineState .
  subsort Process < Soup .
  subsort Int < Pid .
  op [_,_] : Pid Program -> Process .
  op empty : -> Soup .
  op _|_ : Soup Soup -> Soup [prec 61 assoc comm id: empty] .
  op {_,_} : Soup Memory -> 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 .                       var E : Expression .
  rl  {[I, U ; R] | S, M} => {[I, R] | S, M} .

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

  rl {[I, (Q := E) ; R] | S, [Q, X] M} =>
        {[I, R] | S, [Q,eval(E,[Q, X] M)] M} .

  crl {[I, (Q := E) ; R] | S, M} =>
        {[I, R] | S, [Q,eval(E,M)] M} if Q in M =/= true .

  rl {[I, if T then P fi ; R] | S, M} =>
        {[I, if eval(T, M) then P else skip fi ; R] | S, M} .

  rl {[I, while T do P od ; R] | S, M} =>
     {[I, if eval(T, M) then (P ; while T do P od) else skip fi ; R]
         | S, M} .

  rl {[I, repeat P forever ; R] | S, M} =>
        {[I, P ; repeat P forever ; R] | S, M} .
endm
```

The Slightly Modified Rules for **PARALLEL** for LTL Model Checking

```
mod PARALLEL is
  inc SEQUENTIAL .
  inc TESTS .

  sorts Pid Process Soup MachineState .
  subsort Process < Soup .
  subsort Int < Pid .
  op [_,_] : Pid Program -> Process .
  op empty : -> Soup .
  op _|_ : Soup Soup -> Soup [prec 61 assoc comm id: empty] .
  op {_,_,_} : Soup Memory Pid -> MachineState .
  *** Pid of last process acting on the state has been added as 3rd component

  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 . var E : Expression .

  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 := E) ; R] | S, [Q, X?] M, J} =>
        {[I, R] | S, [Q,eval(E,[Q, X?] M)] M, I} .

  crl {[I, (Q := E) ; R] | S, M, J} =>
        {[I, R] | S, [Q,eval(E,M)] M, I} if Q in M =/= true .

  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
```

Dekker's Mutex Algorithm

Dekker's mutex algorithm was explained in Lecture 19. As there, for ease of notation it is specified with the desired initial state in the following module extending (the slightly modified version of) PARALLEL:

```
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
```

LTL Model Checking of Dekker's Algorithm

We need to define three state predicates parameterized by the process id: `enterCrit`, when the process is about to enter its critical section, `in-rem`, when the process is executing its remaining code fragment, 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
```

LTL Model Checking of Dekker's Algorithm (II)

The mutual exclusion safety property is satisfied:

```
reduce in CHECK : modelCheck(initial,[]~ (enterCrit(1) /\ enterCrit(2))) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1714 in 50ms cpu (50ms real) (34280 rewrites/second)
result Bool: true
```

LTL Model Checking of Dekker's Algorithm (III)

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

```
reduce in CHECK : modelCheck(initial,[]<> exec(1) -> []<> enterCrit(1)) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 16 system states.
rewrites: 159 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 ;
    crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [2,repeat 'c2 := 1 ; while
    'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 :=
    1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,0] ['c2,0] [
    'turn,1],0},unlabeled}
   ...
```

LTL Model Checking of Dekker's Algorithm (V)

However, the more subtle weaker fairness 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 holds; of course, the same holds for p2.

```
reduce in CHECK : modelCheck(initial,[]<> exec(1) /\
                                []<> exec(2) -> []<> ~ in-rem(1)
    -> []<> enterCrit(1)) .
ModelChecker: Property automaton has 5 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 2219 in 60ms cpu (70ms real) (36983 rewrites/second)
result Bool: true
```