Program Verification: Lecture 25 (part II)

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Verification of Concurrent Imperative Programs

An imperative programming language $\mathcal{L}$ can be either deterministic (single-threaded) or concurrent (multi-threaded). IMPL is a deterministic imperative language. Instead, Java is a concurrent imperative language.

The Rewriting Logic Semantics Project approach can define the semantics of any imperative language $\mathcal{L}$, either deterministic or concurrent, as a rewrite theory $\mathcal{R}_{\mathcal{L}}$.

Given a deterministic or concurrent imperative language $\mathcal{L}$, $\mathcal{R}_{\mathcal{L}}$ specified as a Maude system module automatically gives as a parser and an interpreter for $\mathcal{L}$. But there is more:

Verification of Concurrent Imperative Programs (II)

1. We can prove reachability properties of programs in $\mathcal{L}$ deductively by instantiating the Reachability Logic prover to $\mathcal{R}_{\mathcal{L}}$.

2. We can prove LTL properties of programs in $\mathcal{L}$ automatically by model checking them by instantiating the Maude LTL Model Checker with $\mathcal{R}_{\mathcal{L}}$.

We can illustrate this language-generic model checking method by defining the rewriting logic semantics of a simple concurrent imperative language called `PARALLEL`. The same approach can be used to model check programs in a deterministic language like IMPL.

The Rewriting Semantics of **PARALLEL**

```
*** A simple parallel language and its rewriting logic semantics.
*** Extends an even simpler language presented in ``The Maude LTL
*** Model Checker'' by Eker, Meseguer, and Sridaranarayanan,
*** in Proc. WRLA'02, ENTCS Vol. 71, Elsevier, 2002.

fmod MEMORY is inc INT .  inc QID .
  sorts Memory Bool? Int? .
  subsorts Bool < Bool? . subsorts Int < Int? .
  op null : -> Int? .
  op none : -> Memory .
  op __ : Memory Memory -> Memory [assoc comm id: none] .
  op [_,_] : Qid Int? -> Memory .
  op _in_ : Qid Memory -> Bool? .
  var Q : Qid .   var M : Memory .  var N? : Int? .
  eq null + N? = null .
  eq null * N? = null .
  eq Q in [Q,N?] M = true .
endfm
```

```
***(Equality test comparing the contents of a named memory location to
an Int? value.)

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'? .
  ceq eval(Q = N?, M) = N? == null if Q in M =/= true .
endfm

***(Syntax for arithmetic expressions, and their evaluation semantics.  To
avoid evaluation of expressions by themselves, which would happen even without
a memory for integer subexpressions if we keep the usual syntax, the operators
+ and * are specified as constructors with syntax +' and *' )
```

```
fmod EXPRESSION is
  inc MEMORY .
  sort Expression .
  subsorts Qid Int? < Expression .
  op _+'_ : Expression Expression -> Expression [ctor] .
  op _*'_ : Expression Expression -> Expression [ctor] .
  op eval : Expression Memory -> Int? .

  var Q : Qid .
  var M : Memory .
  vars N N' : Int .
  var N? : Int? .
  vars E E' : Expression .

  eq eval(N?, M) = N? .
  eq eval(Q, [Q, N?] M) = N? .
  ceq eval(Q,M) = null if Q in M =/= true .
  eq eval(E +' E', M) = eval(E,M) + eval(E',M) .
  eq eval(E *' E', M) = eval(E,M) * eval(E',M) .
endfm
```

```
***(Syntax for a trival sequential language.  We allow abstracting
out program fragments as elements of sorts LoopingUserStatement and
UserStatement.  Elements of sort LoopingUserStatement abstract out
potentially nonterminating program fragments, whereas elements of sort
UserStatement but not of sort LoopingUserStatement abstract out terminating
program fragments.)

fmod SEQUENTIAL is
  inc TESTS .
  inc EXPRESSION .

  sorts UserStatement LoopingUserStatement Program .
  subsort LoopingUserStatement < UserStatement < Program .
  op skip : -> Program .
  op _;_ : Program Program -> Program [prec 61 assoc id: skip] .
  op _:=_ : Qid Expression -> Program .
  op if_then_fi : Test Program -> Program .
  op while_do_od : Test Program -> Program .
  op repeat_forever : Program -> Program .
endfm
```

The Rewriting Semantics of `PARALLEL` (II)

Using the above functional modules, we can then define our simple parallel language in a system module `PARALLEL`. The global state is a triple consisting of:

1. a "soup" (set) of processes;

2. the shared memory; and

3. a process identifier recording the last process that touched the memory or, in any event, performed some computation.

Processes themselves are pairs having a process identifier and a program.

The Rewriting Semantics of **PARALLEL** (III)

```
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 .
```

```
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

One of the earliest correct solutions to the mutual exclusion problem was given by Dekker with his algorithm. 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 rightaway. In case of a tie (both variables set to 1) the tie is broken using a variable `turn` that takes values in $\{1, 2\}$.

Dekker's Mutex Algorithm (II)

The code of process 1 in `PARALLEL` 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 ;
  rem1
forever .
```

Dekker's Mutex Algorithm (III)

The code of process 2 is entirely symmetric:

```
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 ;
  rem2
forever .
```

Dekker's Mutex Algorithm (IV)

We can then define the two processes for Dekker's algorithm and the desired initial state in the following module extending `PARALLEL`. Note that we assume that `crit` does terminate, whereas `rem` may not.

```
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
```

Model Checking 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
```

Model Checking Dekker's Algorithm (II)

The mutual exclusion 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
```

Model Checking Dekker's Algorithm (III)

But the strong 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}
   ...
```

Model Checking Dekker's Algorithm (V)

However, 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 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
```

Specifying Java and JVM

`PARALLEL` is a toy language. Can the rewriting logic approach scale up to real concurrent languages? The answer is "yes." For example, to Java and the JVM.

Java was defined at UIUC by Feng Chen, using a CPS semantics as above, with 600 equations and 15 rewrite rules. Azadeh Farzan developed a more direct specification for the JVM, not based on continuations, with around 300 equations and 40 rewrite rules.

Both the Java and the JVM specifications include multithreading, inheritance, polymorphism, object references, and dynamic object allocation. Native methods and most Java libraries are not supported at present.

JavaFAN Project

Based on Maude rewriting logic specifications of Java and JVM, the JavaFAN (Java Formal ANalyzer), a tool in which Java and JVM code can be executed and analyzed, was developed.

Performance of JavaFAN

Tests	JVM	Java	Other
Remote Agent (s)	0.3	0.1	2 (Stanford)
2-stage Pipeline	17m	—	100m+ (Stanford)
DinPhil (4)	0.64	1.2	—
DinPhil (6)	33.3	81.7	—
DinPhil (8)	13.7m	98m	—
DinPhil (9)	803.2m	—	—
Deadlock-free DinPhil (5)	3.2m	19.2	∞ (JPF)
Deadlock-free DinPhil (7)	686.4m	27m	∞ (JPF)
Thread Game (100) (s)	17.1	6.6	—
Thread Game (1000) (s)	10.1m	5.1m	—

Performance of JavaFAN: Some discussion

There are essentially two reasons for JavaFAN to compare favorably with more conventional Java analysis tools: (1) the high performance of Maude for execution, search, and model checking; and (2) optimized equational and rule definitions.

The second reason is the use of performance-enhancing specification techniques at the Maude level, including:

- expressing as equations E the semantics of all deterministic computations, and as rules R only concurrent computations.
- favoring unconditional equations and rules over less efficient conditional versions.
- using a continuation passing style in semantic equations.

Other Language Case Studies

Similar positive experience in using rewriting logic and Maude to give semantics definitions of concurrent programming languages and getting interpreters and program analysis tools for free for those languages is reported in several papers, including the surveys by Meseguer and Roşu in: (i) Proc. IJCAR'04, Springer LNCS 3097; and (ii) Proc. SOS'05, Elsevier ENTCS.

In particular, semantic definitions have already been given in Maude for substantial subsets of the following languages: ABEL, bc, Beta, CCS, CIAO, CML, Creol, ELOTOS, Haskell, Lisp, LLVM, MSR, Pi-Calculus, Pict, PLAN, Python, Ruby, SIMPLE, Verilog, and Samalltalk. And full definitions have been given in K-Maude to C and Scheme.