Program Verification: Lecture 20

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Semantics of Concurrent Imperative Languages

An imperative programming language $\mathcal{L}$ can be either deterministic (single-threaded) or concurrent (multi-threaded). For example, 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:

Semantics of Concurrent Imperative Programs (II)

We can prove that an invariant Q, i.e., a "good" or "safe" set of states, holds for a program P in $\mathcal{L}$ by entering the specification $\mathcal{R}_{\mathcal{L}}$ in Maude and then giving the command:

`search init-P =>*` u `s.t.` φ `.`

where $u \mid \varphi$ is a constrained constructor pattern s. t. $Q = [\![u \mid \varphi]\!]$.

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 prove invariants of programs in any other languages.

The Rewriting Semantics of **PARALLEL**

```
*** A simple parallel language and its rewriting logic semantics.
*** Memory model with locations named by Qids holding Ints.

fmod MEMORY is
  protecting INT .
  protecting QID .

  sorts Memory Bool? .
  subsorts Bool < Bool? .
  op none : -> Memory .
  op __ : Memory Memory -> Memory [assoc comm id: none] .
  op [_,_] : Qid Int -> Memory .
  op _in_ : Qid Memory -> Bool? . *** cell allocated for Q?

  var Q : Qid .          var M : Memory .          var N : Int .

  eq Q in [Q,N] M = true .
endfm
```

```
***(Test comparing the contents of a named memory location to
an integer.  By default, value of non-allocated Qid is 0.)

fmod TESTS is
  inc MEMORY .

  sort Test .
  op _=_ : Qid Int -> Test .
  op _>'_ : Qid Int -> Test .
  op  _&_ : Test Test -> Test [assoc] .
  op eval : Test Memory -> Bool .
  var Q : Qid .                          var M : Memory .
  var N N' K : Int .                     vars T T' : Test .

  eq eval(Q = N, [Q, N'] M) = N == N' .
  ceq eval(Q = N, M) = N == 0 if Q in M =/= true .
  eq eval(Q >' N, [Q, K] M) = K > N .
  ceq eval(Q >' N, M) = 0 > N if Q in M =/= true .
  eq eval(T & T',M) = eval(T,M) and eval(T',M) .
endfm
```

```
***(Syntax for arithmetic expressions, and their evaluation semantics.
To avoid evaluation of expressions by themselves, 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 _-'_ : Expression Expression -> Expression [ctor] .
  op eval : Expression Memory -> Int .
var Q : Qid .    var M : Memory .   vars N N' : Int .  vars E E' : Expression .

  eq eval(N, M) = N .
  eq eval(Q, [Q, N] M) = N .
  ceq eval(Q,M) = 0 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) .
  eq eval(E -' E', M) = eval(E,M) - eval(E',M) .
endfm
```

```
***(
Syntax for a trival sequential programming language.  We allow abstracting
out program fragments as elements of sorts LoopingUserStatement and
UserStatement.  LoopingUserStatements abstract out potentially
nonterminating program fragments. UserStatements which are not
LoopingUserStatements 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 pair consisting of:

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

2. the shared memory.

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 -> 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
```

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 ;
    crit1 ;
    '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 ;
  crit2 ;
  'turn := 1 ;
  'c2 := 0 ;
  rem2
```

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 `crit1` and `crit1` terminate, whereas `rem1` `rem2` may not.

```
mod DEKKER is
inc PARALLEL .
subsort Int < Pid .
ops crit1 crit2 : -> UserStatement .
ops rem1 rem2 : -> LoopingUserStatement .
ops p1 p2 : -> Program .
op initialMem : -> Memory .
op initial : -> MachineState .

var M : Memory .
vars P R : Program .
var S : Soup .          var I : Pid .
```

```
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 ;
      crit1 ;
      'turn := 2 ;
      'c1 := 0 ;
      rem1
   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 ;
    crit2 ;
    'turn := 1 ;
    'c2 := 0 ;
    rem2
forever .

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

Verifying Mutual Exclusion for Dekker's Algorithm

Mutual exclusion for Dekker's algorithm of course means that `p1` and `p2` can never both be in their critical sections at the same time.

We can define the failure of our `mutex` predicate by a simple pattern and search for it as follows:

```
search initial =>* {S | [1,crit1 ; R] | [2,crit2 ; P],M} .

No solution.
```

Verifying Deadlock Freedom for Dekker's Algorithm

Deadlock freedom for Dekker's algorithm means the obvious: the algorithm should go on forever without ever getting stuck.

We can pove this property by using the `=>!` option in `search`:

```
search initial =>! MS:MachineState .

No solution.
```

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.

Since the Maude rewriting logic specifications of Java and the JVM could be used to verify programs we compared the performance of our specifications with two verification tools, one at Stanford and another at NASA (JPF).

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) Theor. Comp. Sci. (373) 213–237 (2007); (iii) (with Serbanuta) Info. & Comp. (207) 305–340 (2009); (iii) Info. & Comp. (231) 338–69 (2013).

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 Smalltalk. And full definitions have been given in K-Maude to C and Scheme.