Term Rewriting: Lecture 16

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Verification of Concurrent Imperative Programs

Executable rewrite theories are concurrent declarative programs, which can be implemented as distributed algorithms in Maude using Maude's built-in sockets. We have already seen how program properties such as invariants and LTL formulas can be verified either deductively (e.g., with the InvA tool), or by search or LTL model checking.

But how can we reason formally about programs in a concurrent imperative language say, $\mathcal{L}$? Since such languages are not mathematical objects, but baroque engineering artifacts, to mathematicaly verify the properties of a program P in $\mathcal{L}$, we first need a mathematical model of $\mathcal{L}$, such as a formal semantics for $\mathcal{L}$. How can this be done?

Verification of Concurrent Imperative Programs (II)

Since rewriting logic is a computational logic for concurrencym we should of course specify the semantics of a concurrent imperative language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$. This can be done in an executable way as a Maude system module, thus getting an interpreter for $\mathcal{L}$.

Then, the correctness of imperative programs in $\mathcal{L}$ can be reduced to proving inductive properties satisfied by the canonical reachability model $\mathcal{C}_{\mathcal{R}_{\mathcal{L}}}$. If such properties are specified in temporal logic, then we can use methods such as model checking or deductive proof.

We can illustrate this idea by definig the rewriting logic semantics of a simple parallel language called `PARALLEL`.

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 (IV)

Even the weaker liveness property that if both p1 and p2 execute infinitely often then both enter their critical sections infinitely often fails, due to possible looping in the rem part:

```
reduce in CHECK : modelCheck(initial,[]<> exec(1) /\ []<> exec(2) -> []<> enterC
    /\ []<> enterCrit(2)) .
ModelChecker: Property automaton has 7 states.
ModelCheckerSymbol: Examined 236 system states.
rewrites: 1972 in 50ms cpu (50ms real) (39440 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-r
    -> []<> 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
```

The Thread Game

A simple, yet interesting, program that we can also implement in PARALLEL is a "game," suggested by J Moore, between two forever-looping processes accessing a shared variable 'c that initially holds the value 1.

Each process loop reads twice the value of 'c in two different local variables, and then writes the sum of those two local variables back into 'c. There is no synchronization at all between the processes.

Two interesting questions are: (1) which values can 'c hold, depending on the different strategies in this game? and (2) which values can 'c hold if only one of the processes is actually running?

The Thread Game (II)

The code for these processes and the relevant initial states can be defined as follows,

```
mod THREAD-GAME is
  inc PARALLEL .
  ops p1 p2 : -> Program .
  ops init init1 init2 : -> MachineState .

  eq p1 =
       repeat
         'a1 := 'c ;
         'b1 := 'c ;
         'c := 'a1 +' 'b1
       forever .
```

```
  eq p2 =
        repeat
          'a2 := 'c ;
          'b2 := 'c ;
          'c := 'a2 +' 'b2
        forever .

  eq init = { [1, p1] | [2, p2], ['c, 1], 0 } .
  eq init1 = { [1, p1], ['c, 1], 0 } .
  eq init2 = { [2, p2], ['c, 1], 0 } .
endm
```

The Thread Game (II)

We can use the `search` command in Maude to gain some experimental evidence about the first question,

```
Maude> search [1] init =>* { S:Soup, ['c, 1] M:Memory, J:Pid } .
Solution 1 (state 0)
states: 1  rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> none
J:Pid --> 0

Maude> search [1] init =>* { S:Soup, ['c, 2] M:Memory, J:Pid } .
Solution 1 (state 13)
states: 14  rewrites: 38 in 10ms cpu (10ms real) (3800 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,1] ['b1,1]
```

```
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 3] M:Memory, J:Pid } .
Solution 1 (state 69)
states: 70  rewrites: 326 in 0ms cpu (0ms real) (~ rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,1] ['b1,1] ['a2,1] ['b2,2]
J:Pid --> 2

Maude> search [1] init =>* { S:Soup, ['c, 4] M:Memory, J:Pid } .
search [1] in THREAD-GAME : init =>* {S:Soup,M:Memory ['c,4],J:Pid} .
states: 62  rewrites: 282 in 10ms cpu (10ms real) (28200 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,2] ['b1,2]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 5] M:Memory, J:Pid } .
Solution 1 (state 275)
states: 276  rewrites: 1437 in 30ms cpu (30ms real) (47900 rewrites/second)
```

```
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,2] ['b1,3] ['a2,1] ['b2,2]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 6] M:Memory, J:Pid } .
Solution 1 (state 243)
states: 244  rewrites: 1278 in 20ms cpu (20ms real) (63900 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,2] ['b1,2] ['a2,2] ['b2,4]
J:Pid --> 2

Maude> search [1] init =>* { S:Soup, ['c, 7] M:Memory, J:Pid } .
Solution 1 (state 912)
states: 913  rewrites: 4998 in 100ms cpu (100ms real) (49980 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,4] ['b1,3] ['a2,1] ['b2,2]
J:Pid --> 1
```

```
Maude> search [1] init =>* { S:Soup, ['c, 8] M:Memory, J:Pid } .
search [1] in THREAD-GAME : init =>* {S:Soup,M:Memory ['c,8],J:Pid} .
states: 236  rewrites: 1234 in 30ms cpu (30ms real) (41133 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,4] ['b1,4]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 9] M:Memory, J:Pid } .
Solution 1 (state 883)
states: 884  rewrites: 4846 in 90ms cpu (90ms real) (53844 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,3] ['b1,3] ['a2,3] ['b2,6]
J:Pid --> 2

Maude> search [1] init =>* { S:Soup, ['c, 10] M:Memory, J:Pid } .
Solution 1 (state 829)
states: 830  rewrites: 4511 in 90ms cpu (90ms real) (50122 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
```

```
M:Memory --> ['a1,4] ['b1,6] ['a2,2] ['b2,4]
J:Pid --> 1

...

Maude> search [1] init =>* { S:Soup, ['c, 99] M:Memory, J:Pid } .
Solution 1 (state 68974)
states: 68975  rewrites: 408394 in 8960ms cpu (9020ms real) (45579
    rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever] | [2,
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,48] ['b1,51] ['a2,3] ['b2,48]
J:Pid --> 1
```

The Thread Game (III)

We can likewise use the `rewite` command in Maude to gain some experimental evidence about the second question,

```
Maude> rewrite [20] in THREAD-GAME : init1 .
rewrite [20] in THREAD-GAME : init1 .
--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 := 'c ;
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['c,1],1}

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c
    := ('a1 +' 'b1) forever],['c,1] ['a1,eval('c, ['c,1])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +'
    'b1) forever],(['a1,1] ['c,1]) ['b1,eval('c, ['a1,1] ['c,1])],1}
```

```
--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(['a1,1]
    ['b1,1]) ['c,eval('a1 +' 'b1, (['a1,1] ['b1,1]) ['c,1])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 := 'c ;
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,1] ['c,2] ['b1,1],1}

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c
    := ('a1 +' 'b1) forever],(['c,2] ['b1,1]) ['a1,eval('c, (['c,2] ['b1,1]) [
    'a1,1])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +'
    'b1) forever],(['a1,2] ['c,2]) ['b1,eval('c, (['a1,2] ['c,2]) ['b1,1])],1}

--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(['a1,2]
    ['b1,2]) ['c,eval('a1 +' 'b1, (['a1,2] ['b1,2]) ['c,2])],1}
```

```
--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 := 'c ;
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,2] ['c,4] ['b1,2],1}

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c
    := ('a1 +' 'b1) forever],(['c,4] ['b1,2]) ['a1,eval('c, (['c,4] ['b1,2]) [
    'a1,2])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +'
    'b1) forever],(['a1,4] ['c,4]) ['b1,eval('c, (['a1,4] ['c,4]) ['b1,2])],1}

--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(['a1,4]
    ['b1,4]) ['c,eval('a1 +' 'b1, (['a1,4] ['b1,4]) ['c,4])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 := 'c ;
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,4] ['c,8] ['b1,4],1}
```

```
--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c
    := ('a1 +' 'b1) forever],(['c,8] ['b1,4]) ['a1,eval('c, (['c,8] ['b1,4]) [
    'a1,4])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +'
    'b1) forever],(['a1,8] ['c,8]) ['b1,eval('c, (['a1,8] ['c,8]) ['b1,4])],1}

--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(['a1,8]
    ['b1,8]) ['c,eval('a1 +' 'b1, (['a1,8] ['b1,8]) ['c,8])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 := 'c ;
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,8] ['c,16] ['b1,8],1}

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c
    := ('a1 +' 'b1) forever],(['c,16] ['b1,8]) ['a1,eval('c, (['c,16] ['b1,8])
    ['a1,8])],1}
```

```
--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +'
    'b1) forever],(['a1,16] ['c,16]) ['b1,eval('c, (['a1,16] ['c,16]) ['b1,
    8])],1}

--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(['a1,
    16] ['b1,16]) ['c,eval('a1 +' 'b1, (['a1,16] ['b1,16]) ['c,16])],1}

result MachineState: {[1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)
    forever],['a1,16] ['c,32] ['b1,16],1}
Maude>
```

The Thread Game (IV)

The above experimental evidence suggests the following two conjectures:

1. when both processes are running, then for any $n \geq 1$ there is an execution such that `'c` eventually holds n

2. when only one process is running, then `'c` will initially hold 1, and then for each $n \geq 0$ if it holds 2^n, it will continue holding that value until it eventually holds 2^{n+1}.

Can you prove it? (**Note**: Any precise mathematical proof will do; do not even need to use temporal logic).

A Semantic Framework for Programming Languages

PARALLEL is a toy language. Can the rewriting logic approach scale up to real concurrent languages? The answer is "yes." We can define the semantics of a concurrent programming language L by a rewrite theory $\mathcal{R}_L = (\Sigma_L, E_L, R_L)$, where:

- Σ_L specifies L's syntax and the auxiliary operators needed in semantic definitions (memory, environment, etc.)
- the equations E_L specify the semantics of all the deterministic features of L and of the auxiliary semantic operations.
- the rewrite rules R_L specify the semantics of all the concurrent features of L.

Execution and Formal Analysis of Concurrent Programs

Once a definition of a language is given in Maude, we get an interpreter for free and we also get:

1. a semi-decision procedure to find failures of safety properties in a (possibly infinite-state) concurrent program using Maude's `search` command;

2. an LTL model checker for finite-state programs or program abstractions;

3. a theorem prover (Maude's ITP) that can be used to semi-automatically prove programs correct.

Specifying Java and JVM

Java has been recently defined at UIUC by Feng Chen, using a CPS semantics as above, with 600 equations and 15 rewrite rules. Azadeh Farzan has 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, we are developing JavaFAN (Java Formal ANalyzer), a tool in which Java and JVM code can be executed and analyzed. The following figure shows the architecture of JavaFAN.

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) TCS, 373, 213-237, 2007; and (ii) Proc. FTC'11, LNCS Vol. 6914, 1–37, 2011.

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