Program Verification: Lecture 26

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign
Verification of Concurrent Imperative Programs

In the case of deterministic programs, we first studied the verification of declarative deterministic programs such as Maude functional modules. Then, in a sense, we reduced to this case the verification of imperative programs.

Indeed, we can specify the semantics of a deterministic imperative language $\mathcal{L}$ as an equational theory $\mathcal{E}(\mathcal{L})$ (in fact, a Maude functional module).

Then, reasoning about the correctness of imperative programs in $\mathcal{L}$ reduces (perhaps through decomposition by means of a Hoare logic) to proving inductive properties satisfied by the initial model $T_{\mathcal{E}(\mathcal{L})}$. 
What should the analogous situation be in the case of concurrent imperative programs? We should of course specify the semantics of a concurrent imperative language $\mathcal{L}$ as a rewrite theory $\mathcal{R}(\mathcal{L})$ (in fact, a Maude system module).

Then, the correctness of imperative programs in $\mathcal{L}$ can be reduced to proving inductive properties satisfied by the initial model $(T_{\Sigma_L/E_L}, \rightarrow_{\mathcal{R}_L})$. If such properties are specified in temporal logic (resp. reachability logic) then we can use methods such as model checking (resp. deductive proof).

We can illustrate this general method by defining the rewriting logic semantics of a simple threaded language called THREADED-IMP extending IMP.
The Rewriting Semantics of THREADED-IMP

THREADED-IMP extends the semantics of IMP with threads that can communicate with each other through shared variables.

mod THREADED-IMP-SEMANTICS is pr IMP-EVAL + IMP-SYNTAX + IMP-NUMBERS + TERM-STMT

sort LState Thread ThreadSet .
subsort Thread < ThreadSet .
op none : -> ThreadSet .
op {_|_} : Stmt Natural -> Thread [ctor] .
op _|_|_ : ThreadSet Memory Natural -> LState [ctor prec 62] .

Each thread has the form \( \{S \mid J\} \), where \( S \) is a sequential IMP program and \( J \) a number (identifier). The global state is a tripe consisting of an AC set of threads, a shared memory, and the identifier of the last executed thread (used for model checking purposes). The semantics is defined by:
The Rewriting Semantics of THREADED-IMP (II)

var NE : NatExp . var S S' : Stmt . var I : Id . var TS : ThreadSet .
var M : Memory . var N J K : Natural . var BR : BoolRedex .
var B : Boolean . var BE : BoolExp .

rl \{T:TermStmt \ ; S' \mid J\} TS | M | K => \{S' \mid J\} TS | M | J .
rl \{skip \ ; S' \mid J\} TS | M | K => \{S' \mid J\} TS | M | J .
rl \{I := NE \ ; S' \mid J\} TS | [I,N] M | K => \{S' \mid J\} TS | [I,eval([I,N] M,NE)] M | J .
rl \{if BR then S fi \ ; S' \mid J\} TS | M | K =>
    \{if eval(M,BR) then S fi \ ; S' \mid J\} TS | M | J .
rl \{if true then S fi \ ; S' \mid J\} TS | M | K => \{S \ ; S' \mid J\} TS | M | J .
rl \{if false then S fi \ ; S' \mid J\} TS | M | K => \{S' \mid J\} TS | M | J .
rl \{while BE do S od \ ; S' \mid J\} TS | M | K =>
    \{if BE then S \ ; while BE do S od fi \ ; S' \mid J\} TS | M | J .
rl \{repeat S forever \ ; S' \mid J\} TS | M | K =>
    \{S \ ; repeat S forever \ ; S' \mid J\} TS | M | J .
endm

Call the two processes $p_0$ and $p_1$. Each process repeatedly goes through two phases: (i) one in which they try to enter the critical section to execute some terminating code $\text{crit}$, and another in which, after exiting the critical section they (ii) execute some terminating remaining code $\text{rem}$. To indicate that it wants to enter the critical section $p_0$ (resp. $p_1$) sets flag $a$ (resp. $b$) to 1, and then set the shared variable $t$ to 0 (resp. 1). After executing its critical code $p_0$ (resp. $p_1$) sets its flag $a$ (resp. $b$) to 0. Here is the code:
mod PETERSON is pr THREADED-IMP-SEMANTICS .
ops crit rest : -> TermStmt [ctor] .
ops p0 p1 : -> Thread .
op init-mem : -> Memory .
eq init-mem = [a,0] [b,0] [t,0] .
eq p0 = {repeat
  a := 1 ;
  t := 0 ;
  while b = 1 && t = 0 do skip od ;
  crit ; a := 0 ; rest
  forever ; skip | 0} .
eq p1 = {repeat
  b := 1 ;
  t := 1 ;
  while a = 1 && t = 1 do skip od ;
  crit ; b := 0 ; rest forever ; skip | 1} .
endm
The most important property guaranteed by Peterson’s algorithm is mutual exclusion. This property is violated by any state in which both processes are in their critical section.

We can verify this property by giving the search command:

```
search in PETERSON :
p0 p1 | init-mem | 0 =>*
   {crit ; S0:Stmt | 0} {crit ; S1:Stmt | 1} | M:Memory | J:Natural .
```

No solution.
states: 258 rewrites: 1241 in 7ms cpu (7ms real) (174714 rewrites/second)
We should also check the non-starvation of $p_0$ and $p_1$: if each process executes infinitely often, then it will enter its critical section infinitely often. We need to define suitable state predicates:

```plaintext
mod CHECK is inc PETERSON . inc MODEL-CHECKER .
  subsort LState < State .
  ops exec crit : Natural -> Prop .
  eq ({S | J} {S' | L} | M | J) |= exec(J) = true .
  eq ({S | J} {S' | L} | M | K) |= exec(J) = false [owise] .
  eq ({crit ; S | J} {S' | L} | M | K) |= crit(J) = true .
  eq ({S | J} {S' | L} | M | K) |= crit(J) = false [owise] .
endm
```
Unfortunately, starvation can happen:

```
reduce in CHECK : modelCheck(p0 p1 | init-mem | 0, []<> exec(0) -> 
[]<> crit(0)) .
rewrites: 432 in 2ms cpu (2ms real) (170212 rewrites/second)
result ModelCheckResult: counterexample({{repeat a := 1 ; t := 0 ; while b = 1 
   && t = 0 do skip od ; crit ; a := 0 ; rest forever ; skip | 0} {repeat b 
      := 1 ; t := 1 ; while a = 1 && t = 1 do skip od ; crit ; b := 0 ; rest 
      forever ; skip | 1} | [a,0] [b,0] [t,0] | 0,unlabeled} ...}
```

Analyzing the counterexample, the problem happens because p0 can busy-wait forever without p1 getting any chance to make progress.
Model Checking Peterson's Algorithm (IV)

Computations in which one of the threads does not get to execute are totally unfair. We can exclude them by assuming both processes execute infinitely often. Under this assumption, we get the desired non-starvation of p0 and p1:

\[
\text{reduce in CHECK : modelCheck(p0 p1 | init-mem | 0, \[\]\text{<> exec(0)} \land \[\]\text{<> exec(1)} \to \[\]\text{<> crit(0)})}. \\
\text{rewrites: 1991 in 6ms cpu (7ms real) (326019 rewrites/second)} \\
\text{result Bool: (true).False}
\]

\[
\text{reduce in CHECK : modelCheck(p0 p1 | init-mem | 0, \[\]\text{<> exec(0)} \land \[\]\text{<> exec(1)} \to \[\]\text{<> crit(1)})}. \\
\text{rewrites: 1995 in 13ms cpu (13ms real) (151262 rewrites/second)} \\
\text{result Bool: (true).False}
\]
THREADED-IMP 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 $R_L = (\Sigma_L, E_L, R_L)$, where:

- $\Sigma_L$ 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 $L$. 
Once a definition of a language is given in Maude, we get an \textit{interpreter for free} and we also get:

1. a \textit{semi-decision procedure} to find failures of safety properties in a (possibly infinite-state) concurrent program using Maude’s \texttt{search} command;

2. an \textit{LTL model checker} for finite-state programs or program abstractions;

3. a \textit{theorem prover} (Maude’s RL Tool) that can be used to semi-automatically prove programs correct.
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.
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.
## Performance of JavaFAN

<table>
<thead>
<tr>
<th>Tests</th>
<th>JVM</th>
<th>Java</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Agent (s)</td>
<td>0.3</td>
<td>0.1</td>
<td>2 (Stanford)</td>
</tr>
<tr>
<td>2-stage Pipeline</td>
<td>17m</td>
<td>—</td>
<td>100m+ (Stanford)</td>
</tr>
<tr>
<td>DinPhil (4)</td>
<td>0.64</td>
<td>1.2</td>
<td>—</td>
</tr>
<tr>
<td>DinPhil (6)</td>
<td>33.3</td>
<td>81.7</td>
<td>—</td>
</tr>
<tr>
<td>DinPhil (8)</td>
<td>13.7m</td>
<td>98m</td>
<td>—</td>
</tr>
<tr>
<td>DinPhil (9)</td>
<td>803.2m</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Deadlock-free DinPhil (5)</td>
<td>3.2m</td>
<td>19.2</td>
<td>∞ (JPF)</td>
</tr>
<tr>
<td>Deadlock-free DinPhil (7)</td>
<td>686.4m</td>
<td>27m</td>
<td>∞ (JPF)</td>
</tr>
<tr>
<td>Thread Game (100) (s)</td>
<td>17.1</td>
<td>6.6</td>
<td>—</td>
</tr>
<tr>
<td>Thread Game (1000) (s)</td>
<td>10.1m</td>
<td>5.1m</td>
<td>—</td>
</tr>
</tbody>
</table>
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.
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 Roșu 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 Smalltalk. And full definitions have been given in K-Maude to C and Scheme.