

CS476 Takehome Final, Due at 4pm on Wednesday 12/13

Note: Should hand by the above deadline a hardcopy in latex, or similar formatting language, to Ms. Andrea Whitesell, my administrative assistant, who is in office SC 2106. Should likewise email the corresponding working code by the same deadline to meseguer@cs.uiuc.edu. The Maude specifications and all auxiliary files needed for Exercises 3–6 can be retrieved from the course web page.

1. Solve **Ex.** 9.1 in Lecture 9.
2. Solve **Ex.** 10.3 in Lecture 10. **Note:** The formula φ in this problem is supposed to be a universally quantified equation. The exercise then asks you to also consider the case when φ is a (universally quantified) conditional equation. A clearer version of **Ex.** 10.3 is now available in a revised version of Lecture 10 posted on the course web page.
3. Consider the following specifications involving sorting as described in Lecture 12 which you can download from the course web page:

```
fmod INT-LIST is protecting INT .
sorts List .
op nil : -> List [ctor] .
op _:_ : Int List -> List [ctor] .
endfm
```

```
fmod FRAME-SORTING-REQUIREMENTS is protecting INT-LIST .
sort Multiset .
subsort Int < Multiset .
op sorted : List -> Bool .
op null : -> Multiset .
op __ : Multiset Multiset -> Multiset [assoc comm id: null] .
op mset : List -> Multiset .
var L : List .
vars N M : Int .
eq sorted(nil) = true .
eq sorted(N : nil) = true .
ceq sorted(N : M : L) = sorted(M : L) if (N <= M) = true .
ceq sorted(N : M : L) = false if N <= M = false .
eq mset(nil) = null .
eq mset(N : L) = N mset(L) .
endfm
```

```
fmod INSERT-SORT is
protecting INT-LIST .
op ins : Int List -> List .
op sort : List -> List .
vars N M : Int .
var L : List .
eq ins(N, nil) = N : nil .
ceq ins(N, M : L) = N : M : L if N <= M = true .
ceq ins(N, M : L) = M : ins(N, L) if N <= M = false .
```

```

eq sort(nil) = nil .
eq sort(N : L) = ins(N, sort(L)) .
endfm

```

```

fmod INSERT-SORT-VERIFICATION is
protecting INSERT-SORT .
protecting FRAME-SORTING-REQUIREMENTS .
endfm

```

Use the version of the Maude ITP available on the course web page (*not* the one for the Java+ITP tool, but the version you used for a similar exercise in Homework 5) to prove the following goal about the above sorting specification (the goal itself you can also download from the course web page):

```

select ITP-TOOL .
loop init-itp .

```

```

(goal ins1 : INSERT-SORT-VERIFICATION
  |- A{L:List}((sorted(sort(L:List))) = (true)) .)

```

Please, include *both* the proof as a list of commands to the ITP, and a printout of your interaction with the ITP in your hardcopy.

4. Consider a system modelling the behavior of two mathematicians, *A* and *B*, who cycle through an infinite sequence of “think” and “eat” states. There is only one chair at the table, and the right to sit on it is granted by the parity of a counter *n*: *A* can sit down if *n* is odd, while *B* can do so when *n* is even. Upon leaving the table, each mathematician modifies the value of *n* in his/her own fashion: *A* sets it to $3 * n + 1$, and *B* to $n/2$. The specification in Maude of the protocol is as follows (you can retrieve it from the course web page):

```

fmod NATURAL is
  sort Natural .
  op zero : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  ops rem2 quo2 : Natural -> Natural .
  op '+' : Natural Natural -> Natural .
  op '*' : Natural Natural -> Natural .
  vars N M : Natural .
  eq N +' zero = N .
  eq N +' s(M) = s(N +' M) .
  eq N *' zero = zero .
  eq N *' s(M) = N +' (N *' M) .
  eq rem2(s(s(N))) = rem2(N) .
  eq rem2(s(zero)) = s(zero) .
  eq rem2(zero) = zero .
  eq quo2(s(s(N))) = s(quo2(N)) .
  eq quo2(s(zero)) = zero .
  eq quo2(zero) = zero .
endfm

```

```

mod DAMS is
  protecting NATURAL .
  sorts Mode System .
  ops think eat : -> Mode [ctor] .
  op st : Mode Mode Natural -> System [ctor] .
  vars L1 L2 : Mode .
  vars M N : Natural .
  crl st(think, L2, N) => st(eat, L2, N) if rem2(N) = s(zero) .

```

```

rl  st(eat, L2, N) => st(think, L2, (s(s(s(zero))) *' N) +' s(zero)) .
crl st(L1, think, N) => st(L1, eat, N) if rem2(N) = zero .
crl st(L1, eat, N) => st(L1, think, quo2(N)) if rem2(N) = zero .
endm

```

The reason why the natural numbers above are not the predefined ones in Maude is that tools like the Church-Rosser Checker, the Coherence Checker, and the MTT tool do not allow at present use of predefined Maude modules. Furthermore, to use MTT you typically want to:

- flatten the module by including the equations of all submodules
- remove all rules, keep only the equations and make it an fmod
- use prefix syntax for all operators (`plus(N,M)` instead of `N + M`, and so on).

This is because some of the tools that MTT invokes, like AProVe and CiME do not handle well mixfix syntax. The first component of the constructor `st` corresponds to A , the second to B , and the third to the counter n . We are interested in checking, for all computations starting in any state s in which both mathematicians are thinking, that

$$\text{DAMS}, s \models \varphi_i, \quad i = 1, 2,$$

or, equivalently,

$$\text{DAMS} \models \text{initial} \rightarrow \varphi_i, \quad i = 1, 2,$$

where `initial` is a predicate characterizing those initial states s , and each φ_i formalizes in LTL:

- it is always the case that one of the mathematicians is thinking (mutual exclusion);
- every eating phase of A is eventually followed by an eating phase of B ;

In an extension of DAMS, define state predicates that allow you to specify those properties formally as LTL formulas. For this you will need to also import the `SATISFACTION` module in the file `model-checker.maude`. But suppose that you want to use the Church-Rosser Checker to check that the equations defining those state predicates are confluent, or the coherence checker to check that the rules in DAMS are ground coherent with those equations. Since `SATISFACTION` imports `BOOL` which is predefined, you cannot use the above tools. Therefore, for purposes of using the above tools, use instead the following variant of `SATISFACTION`,

```

(fmod SATISFACTION is
  sorts State Prop NewBool .
  ops tt ff : -> NewBool [ctor] .
  op _|=_ : State Prop -> NewBool [frozen] .
endfm)

```

and change your state predicate definitions in a version where `true` becomes `tt`, and `false` becomes `ff`. That is, you will need to have two versions of your specification: one with the ordinary `SATISFACTION` module, and another with the above variant, only for purposes of using tools such as MTT, the Church-Rosser Checker, and the Coherence Checker.

Note that infinity creeps into this problem in two different ways. On the one hand, since n is a natural number, the states reachable starting in an arbitrary state are potentially infinite. On the other hand, the number of initial states for which we want to prove the result is also infinite. Therefore it is impossible to verify these properties directly by model checking (there are infinitely many reachable states, and infinitely many initial states).

It is however possible to verify the above properties by defining an abstraction such that:

- its equations are ground confluent, sort-decreasing and terminating;
- the rules (including perhaps new rules) are ground coherent with respect to the equations;
- the state predicates are preserved;
- the set of reachable states is now finite;

(e) the original module is deadlock-free.

Indeed, because of (d), even though we originally had an infinite number of initial states, in the abstract system they will be represented by a finite number of initial states, so it is enough to model check the properties for each of those initial states to conclude that they hold for all the (infinitely many) initial states in the original system. You should do that using the Maude model checker.

You should also check properties (a)–(c) above using the SCC, MTT, Church-Rosser Checker and Coherence Checker tools, and give a printout of each tool’s output.

Note that by Theorem 2 in page 16 of Lecture 26 (a similar argument was used for abstractions for invariants in pg. 14, Lecture 23), checking (c) above in the context of (a) and (b) boils down to checking that the sort `Bool` (when using the tools you will need the sort `NewBool`) is protected, which can be done automatically using the SCC, MTT, Church-Rosser Checker tools, as explained in those lectures.

Property (b) is the trickiest, and the one where you should do some careful thinking. You should try to check ground coherence of your abstraction giving the command “check ground coherence” for your abstraction module to the ground coherence tool. It is quite possible that the tool will give you some critical pairs it could not check. You now have to:

- think carefully about what rules, if any, need to be added to your abstraction to make it ground coherent;
- convince yourself and convince me (the instructor) that the rules now are ground coherent.

to convince me you need not give a mechanical proof, since it may be the case that the coherence checker cannot establish ground coherence even after you add some extra rules. What you need to do is to give a sound mathematical argument of why your rules are ground coherent; without that, any model checking results will be bogus.

Regarding property (e) above, it is not necessary for you to use the theory transformation explained in Lecture 26, pages 65 and following. It is enough to convince yourself and me that the original module is deadlock-free.

Please note the following:

- you should use the MTT and SCC versions available on the course web page just as before;
- to use the Church-Rosser Checker and Coherence Checker Tool (and only for this) you need to use the latest versions of: (i) Maude (Maude 2.3 of Nov. 20 2006, including the latest `prelude.maude` and `model-checker.maude` files), (ii) Full Maude (Nov. 6 2006) (in file `full-maude.maude`), and (iii) the Church-Rosser Checker and Coherence Checker Tool (Nov. 6 2006) (all in one tool in file `crchc.maude`). All this software is available in the course web page with this exam;
- note that to use the Church-Rosser Checker and Coherence Checker Tool your modules should be enclosed in parentheses and there should be no predefined modules;
- you should first load `full-maude.maude`, then `crchc.maude`, and then your modules, and give the appropriate commands to the Church-Rosser Checker and Coherence Checker Tool.

5. The ThreadGame program below:

```
class ThreadGame
{
    public static void main(String[] args)
    {
        (new Process(1)).start() ;
        (new Process(1)).start() ;
    }
}

class Process extends Thread
{
    static int c;
```

```

public Process(int i)
    {
        c = i ;
    }

public void run()
{
    while (true) {
        c = c + c ;
    }
}
}

```

is a simple multithreaded Java program which shows the possible data races between two threads accessing a common variable (you saw a variant of this in Lecture 27). Each thread reads the value of the static variable `c` twice and writes the sum of the two values back to `c`. The question is what values can `c` possibly hold during the infinite execution of the program. Theoretically, it has been proved that all natural numbers can be achieved. Here we want to use JavaFAN to verify this conclusion for a specific natural number using the search command. More specifically, for the natural number 999, please write a search command to find a possible execution during which the value of `c` becomes 999 at some moment. **Hint:** the deadlock search function provided by JavaFAN gives a good starting point and the property translation shows the way to extract the value of a static field from the `JavaState`. Remember how in the last lecture on JavaFAN it was explained how you can see the Maude command corresponding to invoking a JavaFAN search command for deadlocks. You can then modify this Maude command to obtain the search command you need.

6. The Pipeline Java program below

```

class Main {
static public void main (String argv[]) {
Connector c1, c2, c3;
c1 = new Connector();
c2 = new Connector();
c3 = new Connector();
(new Stage(1, c1, c2)).start();
(new Stage(2, c2, c3)).start();
(new Listener(c3)).start();
for (int i=1; i<2; i=i+1) c1.add(i);
c1.stop();
}
}

class Connector {
public int queue = -1;
public synchronized int take(){
int value;
while ( queue < 0 )
try {wait();} catch (InterruptedException ex) {}
value = queue; queue = -1; return value;
}
public synchronized void add(int o) { queue = o; notifyAll(); }
public synchronized void stop(){ queue = 0; notifyAll(); }
}

class Stage extends Thread {
int id; Connector c1, c2;

```

```

int stop = -1;
public Stage(int i, Connector a1, Connector a2)
{ id = i; c1 = a1; c2 = a2; }
public void run() {
int tmp = -1;
while (tmp != 0)
if ((tmp=c1.take()) != 0){
c2.add(tmp+1);
}
stop = 0;
c2.stop();
}
}

```

```

class Listener extends Thread {
Connector c;
int stop = -1;
public Listener(Connector con) { this.c = con; }
public void run(){
int tmp = -1;
while (tmp != 0)
if ((tmp=c.take()) != 0);
stop = 0;
System.out.println("Listener stop.");
}
}

```

simulates a pipeline architecture. A desired property for this program is the propagation of termination: if the first stage stops, the final listener should stop eventually. Please use JavaFAN to verify the correctness of this program w.r.t. the propagation of termination property. **Hint:** a special field, stop, is added to the program to indicate the stoppage of the stages.

Note. The JavaRL website is at <http://fsl.cs.uiuc.edu/index.php/RewritingLogicSemanticsOfJava>. The tool should be available for download from there. If you have difficulty downloading it or other reasonable questions regarding the use JavaRL, you can send email to Feng Chen (fengchen@cs.uiuc.edu).