CSED490O Automated Software Verification (Spring 2020)

Homework #6 (due Sunday, June 28th)

Problem 1 (Exercise 3.12 in *Principle of Model Checking*)**.** Let P be an linear-time property and $[\![P]\!] = [\![P_{safe}]\!] \cap [\![P_{live}]\!]$ where P_{safe} is a safety property and P_{live} is a liveness property. Show that:

1. $closure([\![P]\!]) \subseteq [\![P_{safe}]\!]$
2. $[\![P_{live}]\!] \subseteq [\![P]\!] \cup ((2^{AP})^{\omega} \setminus closure([\![P]\!]))$

Problem 2 (Exercise 5.8 in *Principle of Model Checking*)**.** Consider the operators $\mathcal{W}$ (weak-until) and $\mathcal{R}$ (release), where $\varphi \,\mathcal{W}\, \psi = (\varphi \,\mathcal{U}\, \psi) \vee \Box\varphi$ and $\varphi \,\mathcal{R}\, \psi = \neg(\neg\varphi \,\mathcal{U}\, \neg\psi)$. Prove the following:

1. $\varphi \,\mathcal{R}\, \psi \;\equiv\; \psi \wedge (\varphi \vee \bigcirc(\varphi \,\mathcal{R}\, \psi))$
2. $\varphi \,\mathcal{R}\, \psi \;\equiv\; (\neg\varphi \wedge \psi) \,\mathcal{W}\, (\varphi \wedge \psi)$

Problem 3 (Exercise 7.2 in *Principle of Model Checking*)**.** Let $M = (S, I, \rightarrow, AP, \mathcal{L})$ be a Kripke structure. Prove the following statements.

1. The union of simulations between M and M is a simulation between M and M.
2. The union of bisimulations between M and M is a bisimulation between M and M.

Problem 4. Let $M = (S, I, \rightarrow, AP, \mathcal{L})$ be a Kripke structure, $R \subseteq S \times S$ be an equivalence relation, and $\alpha : S \rightarrow \widehat{S}$ be an abstraction function. Prove the following statements.

1. M/R and $\pi_R(M)$ are bisimilar (Corollary 44).
2. $\alpha(M)$ and $M/{\equiv_\alpha}$ are bisimilar (Corollary 45).

Problem 5. In this problem, you will implement the big-step and small-step SOS of IMP in Maude. The syntax of IMP programs is declared in the module `IMP-PROGRAM`,[1] and some IMP program examples are provided in the module `IMP-EXAMPLE-CODE`.

- Define rewrite rules for the big-step SOS such that executing an IMP program P results in a state σ iff $\mathcal{R} \vdash$ `exec <` P`, empty >` $\longrightarrow$ `<` σ `>` holds. For example:

```
Maude> rew exec < var 'a ; var 'b ; 'a := # 1 ; 'b := # 2 ; 'a := 'a + 'b, empty > .
rewrites: 15 in 0ms cpu (0ms real) (~ rewrites/second)
result Result: < ('a |-> # 3) ; 'b |-> # 2 >
```

- Define rewrite rules for the small-step SOS such that one-step execution from $\langle p, \sigma \rangle$ to $\langle p', \sigma' \rangle$ exists iff $\mathcal{R} \vdash$ `step <` p`,` σ `>` $\longrightarrow$ `<` p'`,` σ' `>` holds. For example:

```
Maude> rew step (step < var 'a ; var 'b ; 'a := # 1 ; 'b := # 2 ; 'a := 'a + 'b, empty >) .
rewrites: 6 in 0ms cpu (0ms real) (6000000 rewrites/second)
result Config: < skip ; 'b := # 2 ; 'a := 'a + 'b,('a |-> # 1) ; 'b |-> # 0 >
```

[1] For parsing purposes, variables are written as quoted identifiers, and values are written with the prefix `#`.

Problem 6. Dekker's algorithm is one of the earliest solutions to the mutual exclusion problem. There are two threads th_1 and th_2. Thread 1 sets variable `c1` to 0 to indicate that it wishes to enter its critical section. Thread 2 does the same with variable `c2`. If one thread, after setting its variable to 0, finds that the variable of its competitor is 1, then it enters its critical section. In case of a tie, the tie is broken using variable `turn`. The IMP programs for Dekker's algorithm are declared as follows, where the code fragment of the critical section is abstracted as the constant `crit`, and the code fragment of the remaining part is abstracted as `rem`.

```
eq th1 =
    while # true do
      'c1 := # 0 ;
      while 'c2 = # 0 do
        if 'turn = # 2 then
          'c1 := # 1 ;
          while 'turn = # 2 do skip end ;
          'c1 := # 0
        fi
      end ;
      crit ;
      'turn := # 2 ;
      'c1 := # 1 ;
      rem
    end .
```

```
eq th2 =
    while # true do
      'c2 := # 0 ;
      while 'c1 = # 0 do
        if 'turn = # 1 then
          'c2 := # 1 ;
          while 'turn = # 1 do skip end ;
          'c2 := # 0
        fi
      end ;
      crit ;
      'turn := # 1 ;
      'c2 := # 1 ;
      rem
    end .
```

The K semantics of IMP for multiple threads with shared variables is declared in the module `IMP-RULES-THREADS`. A configuration with n threads has the form

$$thread(k(\ldots)\ id(i_1))\ \ldots\ thread(k(\ldots)\ id(i_n))\ env(\ldots)\ latest(i_j)$$

where $i_1, \ldots, i_n$ denote the thread identifiers, and *latest* denotes the most recently executed thread. We assume that `crit` is *terminating*, but `rem` may not be. This is achieved by declaring `crit` as a constant of sort `UserStatement`, and `rem` as one of sort `LoopingUserStatement`, where `LoopingUserStatement` is a subsort of `UserStatement`. The following are the rewrite rules for variable assignment, if, variable declaration, user, and looping user statements.

```
rl [assign]: thread(k(X := V ~> K) id(ID)) env((X |-> V') ; ENV) latest(ID')
          => thread(k(          K) id(ID)) env((X |-> V)  ; ENV) latest(ID) .

rl [cond]: thread(k(if # B then S else S' fi ~> K) id(ID)) latest(ID')
        => thread(k(if B then S else S' fi   ~> K) id(ID)) latest(ID) .

rl [alloc]: thread(k({var X} ~> K) id(ID)) env(ENV) latest(ID')
         => thread(k(           K) id(ID)) env((X |-> # 0) ; ENV) latest(ID) .

rl [user]: thread(k(U ~> K) id(ID)) latest(ID')
        => thread(k(     K) id(ID)) latest(ID) .

rl [user-looping]: thread(k(L ~> K) id(ID)) latest(ID')
                => thread(k(L ~> K) id(ID)) latest(ID) .
```

The initial state for Dekker's algorithm contains two threads th_1 and th_2, and the values of the variables $\{c1 \mapsto 0, c2 \mapsto 0, turn \mapsto 1\}$. For example, the following `frew` command shows one possible execution of Dekker's algorithm, where Thread 1 enters the `crit` section.

```
Maude> frew [20] thread(id(1) k(th1)) thread(id(2) k(th2)) latest(0)
                 env(('c1 |-> # 0) ; ('c2 |-> # 0) ; 'turn |-> # 1) .
rewrites: 60 in 0ms cpu (0ms real) (60000000 rewrites/second)
result KConfig: env(('c1 |-> # 0) ; ('c2 |-> # 1) ; 'turn |-> # 1) thread(k(crit ~> 'turn := # 2 ;
    'c1 := # 1 ; rem ~> while # true do 'c1 := # 0 ; while 'c2 = # 0 do if 'turn = # 2 then 'c1 :=
    # 1 ; while 'turn = # 2 do skip end ; 'c1 := # 0 else skip fi end ; crit ; 'turn := # 2 ; 'c1
    := # 1 ; rem end) id(1)) thread(k(while 'turn = # 1 do skip end ~> 'c2 := # 0 ~> while 'c1 = #
    0 do if 'turn = # 1 then 'c2 := # 1 ; while 'turn = # 1 do skip end ; 'c2 := # 0 else skip fi
    end ~> crit ; 'turn := # 1 ; 'c2 := # 1 ; rem ~> while # true do 'c2 := # 0 ; while 'c1 = # 0
    do if 'turn = # 1 then 'c2 := # 1 ; while 'turn = # 1 do skip end ; 'c2 := # 0 else skip fi
    end ; crit ; 'turn := # 1 ; 'c2 := # 1 ; rem end) id(2)) latest(1)
```

For model checking Dekker's algorithm, four state propositions—parameterized by the thread id—are defined: *in-crit*(i) holds iff Thread i is in the `crit` section; *in-rem*(i) holds iff Thread i is in the `rem` section; *exec*(i) holds iff Thread i has just been executed (i.e., the *latest* id is i); and *enabled*(i) holds iff Thread i can be executed. The following equations define *in-crit*, *in-rem*, and *exec*, where the entire configuration is enclosed by the operator `{_} : KConfig -> State`:

```
ops in-crit in-rem exec : Nat -> Prop .

eq {thread(id(ID) k(crit ~> K)) CF} |= in-crit(ID) = true .
eq {CF} |= in-crit(ID) = false [owise] .

eq {thread(id(ID) k(rem  ~> K)) CF} |= in-rem(ID)  = true .
eq {CF} |= in-rem(ID) = false [owise] .

eq {latest(ID) CF} |= exec(ID') = ID == ID' .
```

1. Define equations for the proposition *enabled*(i), which holds in a configuration where a rewrite rule for Thread i can be applied. For example:

   ```
   Maude> red {thread(id(1) k(th1)) thread(id(2) k(th2)) latest(0)
               env(('c1 |-> # 0) ; ('c2 |-> # 0) ; 'turn |-> # 1)} |= enabled(1) .
   result Bool: true

   Maude> red {thread(id(1) k(skip)) thread(id(2) k(th2)) latest(0)
               env(('c1 |-> # 0) ; ('c2 |-> # 0) ; 'turn |-> # 1)} |= enabled(1) .
   result Bool: false
   ```

2. Use LTL model checking in Maude to verify the mutual exclusion property: *two threads cannot enter the critical section at the same time.*

3. The liveness property $\Box\Diamond$*in-crit*(1) (*Thread* 1 *can enter the critical section infinitely many times*) does not hold, due to unrealistic counterexamples in which Thread 1 is not executed at all. What fairness assumptions are needed to verify $\Box\Diamond$*in-crit*(1)? Use LTL model checking to verify the liveness property $\Box\Diamond$*in-crit*(1) under your fairness assumptions.

Problem 7. In this problem, you will implement the K semantics of IMP#, an extension of IMP with: (i) variable increment; (ii) input and output; (iii) arrays; (iv) halt; (v) functions; and (vi) local and global variables. (Unlike IMP++, we do not consider multiple threads in this problem.) For example, the following IMP# program implements binary search.

```
var array[100];
var x
;;
function main() {
    var size;
    var i;
    size := read;
    for i = 0 to size - 1 do
        array[i] := read
    end;
    x := read;
    print(bsearch(0, size - 1));
    return 0
}
```

```
function bsearch(left, right) {
    var mid;
    if right >= left then
        mid := left + (right - left) / 2 ;
        if (array [mid] = x) then
            return mid
        else if (array [mid] > x) then
            return bsearch(left, mid - 1)
        else
            return bsearch(mid + 1, right)
        fi fi
    fi ;
    return -1
}
```

The syntax of IMP# is defined as follows, declared in the file `imp#-syntax.maude` by extending the syntax of IMP (in the file `imp-syntax.maude`), where i denotes integers, b denotes Boolean values, x denotes variable identifiers, f denotes function identifiers, el denotes comma-separated lists of expressions, and xl denotes comma-separated lists of variable identifiers:

expression	e	$::= i \mid b \mid x \mid e + e \mid e * e \mid e \,/\, e \mid -e \mid e < e \mid e \leq e$
		$\mid e = e \mid e \;\&\&\; e \mid \,!\, e \mid {+}{+}x \mid x[e] \mid f(el) \mid \text{read}$
statement	s	$::= \text{skip} \mid x := e \mid s\,;s \mid \text{if } e \text{ then } s \text{ else } s \text{ fi} \mid \text{while } e \text{ do } s \text{ end}$
		$\mid \text{halt} \mid \text{print}(e) \mid x[e] := e \mid \text{return } e \mid \text{call } e$
declaration	vd	$::= \text{var } x \mid \text{var } x[i] \mid vd\,;vd$
code-block	$code$	$::= vd\,;s \mid s$
function	fd	$::= \text{function } f(xl)\ \{code\} \mid fd\ fd$
program	p	$::= vd\ ;;\ fd$

In addition, the following expressions and statements in IMP# are declared as syntactic sugar:

$$i_1 - i_2 \equiv i_1 + (-i_2) \qquad i_1 > i_2 \equiv i_2 < i_1 \qquad i_1 \geq i_2 \equiv i_2 \leq i_1 \qquad b_1 \,||\, b_2 \equiv \,!(!b_1 \,\&\&\, !b_2)$$

$$\text{if } e \text{ then } s \text{ fi} \equiv \text{if } e \text{ then } s \text{ else skip fi}$$

$$\text{for } x = e_1 \text{ to } e_2 \text{ do } s \text{ end} \equiv x := e_1\,;\text{while } x \leq e_2 \text{ do } s\,; x := x + 1 \text{ end}$$

The heating and cooling rules for IMP# are declared in the file `imp#-strictness.maude`, which extends `imp-strictness.maude` for IMP. In particular, the new rules for IMP# are as follows:

$$\begin{array}{rclcrcl} \mathit{print}\ e & \rightleftharpoons & e \curvearrowright \mathit{print}\ \square & \qquad & \mathit{return}\ e & \rightleftharpoons & e \curvearrowright \mathit{return}\ \square \qquad \mathit{call}\ e \rightleftharpoons e \curvearrowright \mathit{call}\ \square \\ x[e] & \rightleftharpoons & e \curvearrowright x[\square] & & f(el, e, el') & \rightleftharpoons & e \curvearrowright f(el, \square, el') \\ x[e] := e' & \rightleftharpoons & e \curvearrowright x[\square] := e' & & x[v] := e & \rightleftharpoons & e \curvearrowright x[v] := \square \end{array}$$

The K semantics of IMP# is considerably modified to support (recursive) functions and local variable scopes. In particular, the value of a variable is stored in the "memory" and each variable corresponds to a memory location. A K configuration for IMP# has the form:

$$k(\ldots)\ \mathit{env}(\rho)\ \mathit{genv}(\rho')\ \mathit{store}(\sigma)\ \mathit{nextLoc}(l)\ \mathit{stack}(\tau)\ \mathit{funcs}(\mathit{fd})\ \mathit{in}(\ldots)\ \mathit{out}(\ldots)$$

where: (i) *env* denotes a local variable environment; (ii) *genv* denotes a global variable environment; (iii) *store* denotes a memory; (iv) *nextLoc* denotes the next location of the memory to be allocated; (v) *stack* denotes a call stack; (vi) *funcs* denotes function declarations; and (vii) *in* denotes an input stream, and (viii) *out* denotes an output stream.

A variable environment ρ is a mapping from variable identifiers to memory locations, given by as semicolon-separated set of assignments. The function $\rho[x \leftarrow l]$ returns a new environment obtained by updating the location of variable x in ρ to l, defined by the following equations:

$$\begin{aligned}(x \mapsto l' \,;\rho)[x \leftarrow l] &= x \mapsto l \,;\rho \\ (x' \mapsto l' \,;\rho)[x \leftarrow l] &= x' \mapsto l' \,;(\rho[x \leftarrow l]) \qquad \text{if } x \neq x' \\ \emptyset[x \leftarrow l] &= x \mapsto l\end{aligned}$$

Similarly, a memory σ is a mapping from locations to values, and the function $\sigma[l \leftarrow v]$ returns a new memory obtained by updating the value of location l in σ to v, A call stack τ is given by a ::-separated list of stack items, where each stack item is a triple $\langle \rho, K, l \rangle$.

The following K rules define the semantics for the IMP++ subset of IMP#. Only the first four rules on the left have changed from the previous semantics of IMP++. Specifically, variable declarations no longer initialize the values of the variables to 0; var x allocates only a memory location for x. To assign a value to x, variable assignment $x := v$ must be executed in IMP#.

$$k(\underset{v}{x} \curvearrowright K)\ \mathit{env}(x \mapsto l \,;\rho)\ \mathit{store}(l \mapsto v \,;\sigma)$$

$$k(\underset{\cdot}{\text{var } x} \curvearrowright K)\ \mathit{env}(\underset{\rho[x \leftarrow l]}{\rho})\ \mathit{nextLoc}(\underset{l +_{Int} 1}{l})$$

$$k(\underset{\cdot}{x := v} \curvearrowright K)\ \mathit{env}(x \mapsto l \,;\rho)\ \mathit{store}(\underset{\sigma[l \leftarrow v]}{\sigma})$$

$$k(\underset{i +_{Int} 1}{++x} \curvearrowright K)\ \mathit{env}(x \mapsto l \,;\rho)\ \mathit{store}(l \mapsto \underset{i +_{Int} 1}{i} \,;\sigma)$$

$$k(\underset{\text{if } e \text{ then } (s \,;\text{while } e \text{ do } s \text{ end}) \text{ else skip fi}}{\text{while } e \text{ do } s \text{ end}} \curvearrowright K)$$

$$\text{if } \mathit{true} \text{ then } s_1 \text{ else } s_2 \text{ fi} \longrightarrow s_1$$

$$\text{if } \mathit{false} \text{ then } s_1 \text{ else } s_2 \text{ fi} \longrightarrow s_2$$

$$k(\underset{v}{\mathit{read}} \curvearrowright K)\ \mathit{in}((\underset{\cdot}{v},\ VL))$$

$$k(\underset{\cdot}{\mathit{print}(v)} \curvearrowright K)\ \mathit{out}((VL,\ \underset{v}{\cdot}))$$

$$\begin{aligned}
i_1 + i_2 &\longrightarrow i_1 +_{Int} i_2 \\
i_1 * i_2 &\longrightarrow i_1 *_{Int} i_2 \\
i_1 / i_2 &\longrightarrow i_1 /_{Int} i_2 \quad \textbf{if } i_2 \neq 0 \\
-\, i &\longrightarrow -_{Int}\, 1 \\
i_1 < i_2 &\longrightarrow i_1 <_{Int} i_2 \\
i_1 \leq i_2 &\longrightarrow i_1 \leq_{Int} i_2 \\
i_1 = i_2 &\longrightarrow i_1 =_{Int} i_2 \\
\mathit{true}\ \&\&\ e &\longrightarrow e \\
\mathit{false}\ \&\&\ e &\longrightarrow \mathit{false} \\
!\, b &\longrightarrow \mathit{not}(b) \\
\text{skip} &\longrightarrow \cdot \\
s_1 ; s_2 &\longrightarrow s_1 \curvearrowright s_2 \\
vd \,; s &\longrightarrow vd \curvearrowright s \\
vd_1 \,; vd_2 &\longrightarrow vd_1 \curvearrowright vd_2 \\
k(\mathit{halt} \curvearrowright K) &\longrightarrow k(\cdot)
\end{aligned}$$

The following K rules define the semantics for arrays in IMP#. Notice that these rules for arrays are similar to the rules for variables (the first three rules above). Similarly, array declarations do not initialize its values, but allocate only memory locations using *nextLoc*.

$$k(\underline{x[i]} \curvearrowright K)\ env(x \mapsto l\,;\rho)\ store(l' \mapsto v\,;\sigma), \qquad \text{where } l' = l +_{Int} i$$
$$\underline{v}$$

$$k(\underline{\text{var } x[i]} \curvearrowright K)\ env(\underline{\quad \rho \quad})\ nextLoc(\underline{\quad l \quad})$$
$$\underline{\cdot} \qquad\qquad \underline{\rho[x \leftarrow l]} \qquad\qquad \underline{l +_{Int} i}$$

$$k(\underline{x[i] := v} \curvearrowright K)\ env(x \mapsto l\,;\rho)\ store(\underline{\qquad\qquad \sigma \qquad\qquad})$$
$$\underline{\cdot} \qquad\qquad\qquad\qquad\qquad\qquad \underline{\sigma[(l +_{Int} i) \leftarrow v]}$$

The following K rules define the semantics for functions in IMP#. When a function $f(vl)$ is called, a new "stack frame" is created; i.e., the current environment ρ, computation K, and next location l are pushed into the call *stack*. The new local environment for f will include the newly allocated locations for the arguments vl and the global environment ρ'. For this purpose, the K label $|\ xl\ |\ vl\ |$ allocates memory locations for the argument variables xl and assigns the argument values vl to them. When a return statement is executed, the previous environment ρ', computation K', and next memory location l' for the callee are restored.

$$k(\underline{\quad f(vl) \curvearrowright K \quad}\ )\ env(\underline{\rho})\ genv(\rho')\ stack(\underline{\quad \cdot \quad} :: \tau)\ funcs((f(xl)\{code\})\ fd)\ nextLoc(l)$$
$$\underline{|\ xl\ |\ vl\ | \curvearrowright code} \qquad\qquad \underline{\rho'} \qquad\qquad\qquad\qquad \underline{\langle \rho, K, l \rangle}$$

$$k(|\ x, xl\ |\ v, vl\ | \curvearrowright K) \longrightarrow k(\text{var } x \curvearrowright x := v \curvearrowright |\ xl\ |\ vl\ | \curvearrowright K)$$
$$k(|\ nil\ |\ nil\ | \curvearrowright K) \longrightarrow k(K)$$

$$k(\underline{\text{return } v \curvearrowright K})\ env(\underline{\rho})\ stack(\underline{\langle \rho', K', l' \rangle} :: \tau)\ nextLoc(l')$$
$$\underline{v \curvearrowright K'} \qquad\qquad \underline{\rho'} \qquad\qquad\qquad\qquad \underline{\cdot}$$

$$k(call\ v \curvearrowright K) \longrightarrow k(K)$$

Finally, the following K rules deal with IMP# programs. Given an IMP# program $vd\,;;fd$, the global variable declarations in vd are first processed. The remaining function declarations in fd are added into *funcs*, and the `main` function is called.

$$vd\,;;fd \longrightarrow vd \curvearrowright fd$$

$$k(\underline{\quad fd \quad})\ env(\rho)\ genv(\underline{\cdot})\ funcs(\underline{\cdot})$$
$$\underline{\text{call main}()} \qquad\qquad\qquad\qquad \underline{\rho'} \qquad\qquad \underline{fd}$$

Implement K rules for IMP# in the module `IMP#-RULES`. The heating and cooling rules and the K rules in blue above are already defined. You only need to implement the remaining rules.