TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2009

MID-TERM TEST

COMP 202 / SWEN 202
Formal Methods of Computer Science / Formal Foundations of Software Engineering
WITH ANSWERS

Time Allowed: 90 minutes

Instructions: There are **five** (5) questions.

Answer **all** the questions.

The exam will be marked out of **eighty** (80).

Calculators ARE NOT ALLOWED.

Non-electronic foreign language dictionaries are allowed.

No other reference material is allowed.

Question 1. [20 marks]

The following Alloy provides a model of family relations.

```
abstract sig Person {
  mother: lone Woman,
  father: lone Man,
}
sig Man, Woman extends Person {}
```

(a) [8 marks]

Consider the following instance:

$$\texttt{Man} = \{M0, M1\} \qquad \texttt{Woman} = \{W0, W1, W2\}$$

$$\texttt{father} = \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} \qquad \texttt{mother} = \begin{array}{|c|c|}\hline M1 & W0 \\ \hline W1 & W0 \\ \hline \end{array}$$

(i) [2 marks] Draw a visualisation (i.e. a graph representation as Alloy would give) of this instance.

(ii) [2 marks] Compute `mother.~mother`

$$\texttt{mother.\~{}mother} = \begin{array}{|c|c|}\hline M1 & W0 \\ \hline W1 & W0 \\ \hline \end{array} \cdot \tilde{} \begin{array}{|c|c|}\hline M1 & W0 \\ \hline W1 & W0 \\ \hline \end{array} = \begin{array}{|c|c|}\hline M1 & W0 \\ \hline W1 & W0 \\ \hline \end{array} \cdot \begin{array}{|c|c|}\hline W0 & M1 \\ \hline W0 & W1 \\ \hline \end{array} = \begin{array}{|c|c|}\hline M1 & M1 \\ \hline M1 & W1 \\ \hline W1 & M1 \\ \hline W1 & W1 \\ \hline \end{array}$$

(iii) [2 marks] Compute `Man <: (father + mother)`.

$$\begin{aligned} \texttt{(Man <: (father + mother)} &= \begin{array}{|c|}\hline M0 \\ \hline M1 \\ \hline \end{array} \texttt{<:} \left(\begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} + \begin{array}{|c|c|}\hline M1 & W0 \\ \hline W1 & W0 \\ \hline \end{array}\right) \\ &= \begin{array}{|c|}\hline M0 \\ \hline M1 \\ \hline \end{array} \texttt{<:} \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline M1 & W0 \\ \hline W1 & W0 \\ \hline \end{array} \\ &= \begin{array}{|c|c|}\hline M1 & M0 \\ \hline M1 & W0 \\ \hline \end{array} \end{aligned}$$

(iv) [2 marks] Compute `^father`

$$
\begin{aligned}
& \texttt{\^{}father} \\
= \; & \texttt{father + father.father + father.father.father + ...} \\
= \; & \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} + \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} \cdot \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} + \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} \cdot \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} \cdot \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} + \ldots \\
= \; & \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline \end{array} + \begin{array}{|c|c|}\hline W2 & M0 \\ \hline \end{array} + \texttt{none} + \ldots \\
= \; & \begin{array}{|c|c|}\hline M1 & M0 \\ \hline W2 & M1 \\ \hline W2 & M0 \\ \hline \end{array}
\end{aligned}
$$

(b) [12 marks]

(i) [2 marks] Provide a run command that allows you to show instances with at least one person (that is, in which set `Person` is not empty).

```
run { some Person }
```

(ii) [3 marks] Write an Alloy function that takes a person as argument and returns all his or her descendants; i.e. children, grandchildren, great-grandchildren, etc.

```
fun descendant[p: Person]: set Person {
  ^(mother+father).p
}
```

(iii) [2 marks] Write a fact that ensures that no person is a descendant of itself.

```
fact { no p: Person | p in descendant[p] }
```

(iv) [2 marks] Write an Alloy predicate that takes two persons as arguments and is true if the two have at least one child in common.

```
pred met[p,q: Person] {
  some c: Person | c->p + c->q in father + mother
}
```

(v) [3 marks] Provide an Alloy command to check that, forall persons that have a child in common, one of them must be a man and one of them a woman. Is this assertion true for the model given above?

```
assert a {
  all p,q: Person | met[p,q] implies
       one Man & (p + q) and one Woman & (p + q)
}
check a
```

Question 2. [20 marks]

Consider the following Alloy model of a file system:

```
sig Directory {}

sig FileSystem {
  root: Directory,
  dirs: set (Directory-root),
  parent: dirs -> one (dirs + root)
}

pred init[fs: FileSystem] {
  no fs.dirs
}
```

(a) [2 marks]

Adding a fact to the above specification requiring the root directory to be an ancestor (i.e. a parent or a parent of a parent etc.) of itself would make the model inconsistent. Explain, with reference to this example, what it means for a model to be inconsistent.

The `parent` relation above associates directories excluding the root directory to their parents. Thus the root directory cannot have a parent directory and so it cannot have ancestors. Requiring the root directory to be one of its ancestors contradicts the fact that the root directory cannot have ancestors, making the model inconsistent.
An inconsistent specification contains contradictory facts and thus cannot be satisfied. Run commands won't find any instances for an inconstistent model, neither will there be any counter-examples for assertions.

(b) [4 marks]

Provide an invariant (a predicate called `inv`) that is true for file systems in which the root directory is an ancestor (i.e. a parent or a parent of a parent etc.) of all non-root directories in the file system.

```
pred inv[fs: FileSystem] {
  all d: fs.dirs | fs.root in d.^(fs.parent)
}
```

(c) [2 marks]

The given predicate `init` describes the initial states of the file system. What can you say about the `parent` relation of initial file systems?

If relation `dirs` is empty, relation `parent` must be empty too. Thus, relation `parent` is empty for all initial filesystems.

(d) [3 marks]

Write an Alloy command to check that initial file systems satisfy the invariant provided in **(b)**. Is this assertion true for the model given above?

```
assert InitEstablishesInv {
  all fs: FileSystem | init[fs] implies inv[fs]
}
check InitEstablishesInv for 10
```

The assertion is implied by the model thus true.

(e) [3 marks]

What does it mean for an operation to preserve an invariant?

An operation preserves an invariant if all post-states of the operation satisfy the invariant provided the pre-state of the operation satisfies the invariant.

(f) [6 marks]

Provide an operation that models adding a new directory to a given directory of a file system. Make sure your operation preserves the invariant provided in **(b)**.

```
pred add[fs, fs': FileSystem, new, where: Directory] {
  where in fs.dirs+fs.root
  new !in fs.dirs+fs.root
  fs.root = fs'.root
  fs'.dirs = fs.dirs + new
  fs'.parent = fs.parent + new->where
}
```

Question 3. [14 marks]

Consider the following JML annotated Java program:

```
public class Person
{
  //@ invariant height >= 0;
  private /*@ spec_public @*/ int height;

  //@ ensures height == \old(height) + distance;
  public void grow(int distance) {
    height += distance;
  }

  public /*@ pure @*/ int getHeight() {
    return height;
  }

  public static void main(String[] args) {
    Person p = new Person();
    p.grow(30);
    System.out.println(p.getHeight());
  }
}
```

(a) [1 mark]

The program compiles using `jmlc` without problems. What will happen if you execute it using `jmlrac`?

It prints 30.

(b) [3 marks]

Provide a new `main` method that shows that the `grow` method does not preserve class `Person`'s invariant. What will happen if you execute your new `main` method using `jmlrac`?

```
public static void main(String[] args) {
  Person p = new Person();
  p.grow(-30);
  System.out.println(p.getHeight());
}
```

Running with `jmlrac` will produce a `JMLInvariantError` caused by method `grow`.

(c) [4 marks]

Provide a JML annotation that ensures that the `grow` method preserves the invariant. What will happen if you execute your main method from part **(b)** with this modified program using `jmlrac`?

Add `requires` clause for `grow` method:

```
//@ requires distance >= 0;
//@ ensures ...
public void grow(int distance)
```

Running with `jmlrac` will produce a `JMLPreconditionError` for method `grow` called in `main`.

(d) [3 marks]

When does a *postcondition* for a method need to hold? Under which circumstances is the postcondition not required to hold?

A postcondition needs to hold when a method finishes, provided the precondition and class invariant were satisfied when the method was called.
If the precondition or class invariant were broken when the method was called, the method might not establish the postcondition on termination, or might not terminate at all.

(e) [3 marks]

What is a *pure* method? Why can only pure methods be used in JML specifications?

Pure methods do not assign to non-local variables during their execution, thus don't have any side-effects.
A JML specification gets executed by tools like `jmlrac` but must not influence the execution of a program (apart from signaling specification violations). A pure methods does not change any fields so guarantees to not influence the program execution.

Question 4. [6 marks]

Do the following methods correctly implement their specification? Give a brief explanation why you think they do or do not.

(a) [2 marks]

```
//@ requires true;
//@ ensures \result == 0 || \result == 1;
int foo() {
  return 0;
}
```

Yes, since 0 is returned and 0 or 1 is expected.

(b) [2 marks]

```
//@ requires x != y;
//@ ensures \result == x || \result == y;
//@ ensures \result > x || \result > y;
int bar(int x, int y) {
  if (x >= y) return x;
  return y;
}
```

Yes, since the method always returns either `x` or `y` and, if `x != y` (see precondition), the method returns the greater of the two.

(c) [2 marks]

```
//@ requires false;
//@ ensures true;
int baz() {
  throw new IllegalArgumentException();
}
```

Yes, since the precondition cannot be satisfied.

Question 5. [20 marks]

Consider a Java method:

```
boolean equivalent(int[] a, int[] b) {
  int i = 0;
  while (i < a.length && a[i] == b[i]) {
    i++;
  }
  return i == a.length;
}
```

This method takes as input two integer arrays of the same length and determines whether the given arrays are equivalent, that is, contain the same integers in exactly the same order.

(a) [4 marks]

Give a JML specification (precondition and postcondition) for this method.

```
//@ requires a != null && b != null && a.length == b.length;
//@ ensures \result <==> (\forall int k; 0 <= k && k < a.length; a[k] == b[k]);
```

(b) [6 marks]

Provide a loop invariant that may be used in the verification of the above implementation of the method.

```
//@ loop_invariant 0 <= i && i <= a.length;
//@ loop_invariant (\forall int k; 0 <= k && k < i; a[k] == b[k]);
```

(c) [10 marks]

Give an argument (informal proof), using your loop invariant from part **(b)**, to show that the method correctly implements its specification; i.e. show that:

(i) [2 marks] The loop invariant holds on entry to the loop.

```
//@ requires a != null && b != null && a.length == b.length;
public static boolean diff(int[] a, int[] b)
{
  // the precondition should hold at this point:
  //@ assert a != null && b != null && a.length == b.length;

  int i = 0;

  //@ assert i == 0;
  //@ assert a != null && b != null && a.length == b.length;
  // these imply:
  //@ assert 0 <= i && i <= a.length;
  // We also know:
  //@ assert (\forall int k; 0 <= k && k < 0; a[k] == b[k]);
  // and with i == 0 thus
  //@ assert (\forall int k; 0 <= k && k < i; a[k] == b[k]);
```

(ii) [3 marks] The loop invariant is preserved by the loop body.

```
// provided the loop invariant holds:
//@ assert 0 <= i && i <= a.length;
//@ assert (\forall int k; 0 <= k && k < i; a[k] == b[k]);
// We also know that loop condition is true
// (otherwise loop body wouldn't be executed)
//@ assert i < a.length && a[i] == b[i];

// loop body consists only of one statement:
i++;

//@ assert 0 <= i && i <= a.length;
//@ assert (\forall int k; 0 <= k && k < (i-1); a[k] == b[k]);
//@ assert a[i-1] == b[i-1];
// these two assertions imply:
//@ assert (\forall int k; 0 <= k && k < i; a[k] == b[k]);
```

(iii) [5 marks] The postcondition of the method holds when the loop exits with the loop invariant true.

```
  while (i < a.length && a[i] == b[i]) {
    i++;
  }
// the loop invariant holds when the loop exists:
//@ assert 0 <= i && i <= a.length;
//@ assert (\forall int k; 0 <= k && k < i; a[k] == b[k]);
// We also know that the loop condition is false
// (otherwise loop wouldn't have exited)
//@ assert ! (i < a.length && a[i] == b[i]);
// rewritten:
//@ assert i >= a.length || a[i] != b[i];
// together with i <= a.length:
//@ assert i == a.length || a[i] != b[i];
// Thus:
//@ assert ( i == a.length &&
             (\forall int k; 0 <= k && k < a.length; a[k] == b[k]))
           ||
           (0 <= i && i < a.length && a[i] != b[i])

return i == a.length;
```

****