Victoria University of Wellington
School of Engineering and Computer Science

SWEN326: Safety-Critical Systems

Assignment 4

Due: Friday 7th June @ 23:59

In this assignment you will work on specifying and verifying a range of relatively small programs in Whiley. This includes writing loop invariants and specifying functions from scratch. Accompanying source code for this assignment can be found on the SWEN326 Lecture Schedule. **You are advised to test and verify your programs using the Whiley tool.** — see the Appendix for help on running Whiley.

1 Implementing Specifications (15 marks)

(a) [1 mark] Provide any implementation for the following function which meets the given specification.

```
function aByte() -> (int r)
ensures 0 <= r && r <= 255:
   ...
```

(b) [1 mark] Provide any implementation for the following function which meets the given specification.

```
function g() -> (int x, int y)
ensures x < y:
   //
   ...
```

(c) [1 mark] Provide any implementation for the following function which meets the given specification.

```
function diff() -> (int x, int y)
ensures x - y == 0:
   ...
```

(d) [1 mark] Provide any implementation for the following function which meets the given specification.

```
function n() -> (int r)
ensures 2*r >= r:
   ...
```

(e) [1 mark] Provide any implementation for the following function which meets the given specification.

```
function f(int[] items) -> (int r)
requires |items| > 0
ensures some { i in 0..|items| | items[i] == r }:
   ...
```

(f) [5 marks] Provide an appropriate implementation for the `giveChange()` function below which calculates change where one or more five dollar notes are used to purchase an item:

```
type Money is {
   int dollars, // number of dollar coins
   int fiftyCents // number of fifty cent pieces
}

function giveChange(int fiveDollarNotes, Money cost) -> (Money r)
// Money provided for payment must be greater than cost of item
requires fiveDollarNotes * 500 >= cost.dollars * 100 + cost.fiftyCents * 50
// Amount of change returned must give total after cost
ensures (r.dollars + cost.dollars) * 100 +
      (r.fiftyCents + cost.fiftyCents) * 50 == fiveDollarNotes * 500:
   //
   return ...
```

(g) [5 marks] Provide an appropriate implementation for the `tick()` function below which increments the time by one minute:

```
type Time is { int hours, int minutes }
// There are exactly 24 hours in a day, and 60 minutes in an hour
where hours >= 0 && hours < 24 && minutes >= 0 && minutes < 60

function tick(Time t) -> (Time r):
   ...
```

Be sure to check that your implementation verifies!

2 Writing Functions and Specifications (30 Marks)

(a) [1 marks] Provide appropriate **`requires`** and **`ensures`** clauses for the following `swap` function, which accepts two integers and returns them with their order swapped:

```
function swap(int x, int y) -> (int a, int b):
   return y, x
```

(b) [2 marks] A *natural* number is an integer which is greater-than-or-equal to zero. Provide appropriate **`requires`** and **`ensures`** clauses for the following function which adds three natural numbers together to produce a natural number:

```
function sum3(int x, int y, int z) -> (int r)
// No parameter can be negative
requires ...
// Return value cannot be negative
ensures ...:
   //
   return x + y + z
```

(c) [3 marks] Complete the following specification for an array of natural numbers:

```
type nats is (int[] items) where ...
```

You should test your specification by writing some small test functions which take variables of type `nats` as parameters or returns, and making sure they verify as expected.

(a) [7 marks] Write a function `mid()` that takes three integer parameters, of different values, and returns the middle value. You function should include an appropriate specification and you should verify that it meets the specification:

```
function mid(int x,int y, int z) -> (int r)
requires ....
ensures ...:
   ...
```

(d) [7 marks] The Gregorian calendar is the most widely used organisation of dates. A well-known saying for remembering the number of days in each month is the following:

> *"Thirty days hath September, April, June and November. All the rest have thirty-one, except February which has twenty-eight ..."*

You may ignore the issue of leap years. A simple function for returning a date can be defined as follows:

```
final int Jan = 1
final int Feb = 2
final int Mar = 3
final int Apr = 4
final int May = 5
final int Jun = 6
final int Jul = 7
final int Aug = 8
final int Sep = 9
final int Oct = 10
final int Nov = 11
final int Dec = 12

function getDate() -> (int day, int month, int year)
ensures ...:
   return 17, Sep, 2015
```

Provide an **`ensures`** clause for this function which ensures the returned date is valid (ignoring leap years).

(e) [10 marks] We now consider a cut down version of the infamous "water jugs" puzzle with two jugs: a small jug (containing three litres) and a large jug (containing five litres). The following function implements pouring water from the small jug into the large jug:

```
function pourSmall2Large(int smallJug, int largeJug) ->
                        (int smallJugAfter, int largeJugAfter)
// The small jug holds between 0 and 3 litres (before)
requires ...
// The large jug holds between 0 and 5 litres (before)
requires ...
// The small jug holds between 0 and 3 litres (after)
ensures ...
// The large jug holds between 0 and 5 litres (after)
ensures ...
// The amount in both jugs is unchanged by this function
ensures ...
// Afterwards, either the small jug is empty or the large jug is full
ensures ...:
    if smallJug + largeJug <= 5:
        // indicates we're emptying the small jug
        largeJug = largeJug + smallJug
        smallJug = 0
    else:
        // indicates we're filling up the large jug
        smallJug = smallJug - (5 - largeJug)
        largeJug = 5
    return smallJug, largeJug
```

Complete the missing **`requires`** and **`ensures`** clauses based on the given English descriptions.

3 Writing Loop Invariants (15 marks)

(a) [5 marks] Provide an *appropriate* loop invariant for the following function, such that it can be verified to meet its specification.

```
function isPalindrome(int[] chars) -> (bool r)
ensures r <==> all { k in 0..|chars| | chars[k] == chars[|chars|-(k+1)] }:
    //
    int i = 0
    int j = |chars|
    //
    while i < j:
        j = j - 1
        if chars[i] != chars[j]:
            return false
        i = i + 1
    //
    return true
```

(b) [5 marks] Provide an *appropriate* loop invariant for the following function, such that it can be verified to meet its specification.

```
function append(int[] items, int item) -> (int[] rs)
ensures all { k in 0..|items| | rs[k] == items[k] }
ensures rs[|items|] == item:
   int[] nitems = [0; |items| + 1]
   int i = 0
   //
   while i < |items|:
      nitems[i] = items[i]
      i = i + 1
   //
   nitems[i] = item
   return nitems
```

(c) [5 marks] Provide an *appropriate* loop invariant for the following function, such that it can be verified to meet its specification.

```
function lastIndexOf(int[] items, int item) -> (int r)
ensures r >= 0 ==> items[r] == item
ensures r >= 0 ==> all { i in r+1 .. |items| | items[i] != item }
ensures r < 0 ==> all { i in 0 .. |items| | items[i] != item }:
   //
   int i = |items|
   while i > 0:
      i = i - 1
      if items[i] == item:
         return i
   //
   return -1
```

4 Writing Verified Programs (20 marks)

In this part of the assignment, you are asked to specify, implement and verify some small functions. In each case, you need to think carefully what what is an appropriate specification for the function in question.

(a) [10 marks] Provide an *appropriate* specification and implementation for the following function:

```
function reverse(int[] xs) -> (int[] ys)
```

This function should reverse the array provided as the parameter. For example, `reverse([1,2,3])=[3,2,1]`.

(b) [10 marks] Provide an *appropriate* specification and implementation for the following function:

```
function insert(int[] items, int pos, int item) -> (int[] ys)
```

This function inserts an item at a given position in an array. For example, `insert([1,2,3],1,4)=[1,4,2,3]` and `insert([1,2,3],3,4)=[1,2,3,4]`, etc.

5 Case Study: Steam Boiler (20 marks)

This assignment is concerned with design validation of the *steam boiler problem*, with which you should already be familiar. The purpose of the assignment is to gain experience developing formal models of the steam boiler system. The assignment is broken into two parts: the *alloy* model; and, the *whiley* model. Both models are based around the following simplified description of the steam boiler problem.

> "A steam boiler *state* consists of the current *water level* and the state of each *pump* (i.e. either *open* or *closed*). In the initial state, the boiler is empty and all pumps are off. At each transition either a *pump opens*, a *pump closes*, *water is pumped in*, or *steam escapes*."

The above description is considerably simplified compared with the original. Unfortunately, this is necessary in order to develop formal models which are manageable for this assignment. The second part is to develop a Whiley model of the simplified steam boiler system described above. An accompanying file, `SteamBoiler.whiley` is also provided. The *type* for a state is given as follows:

```
type nat is (int n) where n >= 0

type State is { nat level, bool[] pumps }
```

Step 1. Complete the following function which returns the *initial state*: **[5 marks]**

```
function InitialState(nat n) -> (State after)
// Eactly n pumps created
ensures ...
// All pumps initially off
ensures ...
// Boiler initially empty
ensures ...:
   return {level: 0, pumps: [false; n]}
```

Step 2. Complete the following *properties* capturing static behaviour of pumps between the *before* and *after* states: **[5 marks]**

```
// All pumps are unchanged between before/after state
property pumpsUnchanged(State before, State after)
// Number of pumps preserved
where ...
// All pumps remain unchanged
where ...

// All pumps except ith are unchanged between before/after state
property pumpsUnchangedExcept(State before, State after, int i)
// Number of pumps preserved
where ...
// All pumps remain unchanged except ith
where ...
```

Step 3. Complete the following functions for the transitions *pump open* and *pump closed*: **[5 marks]**

```
// Pump open transition occurs when a pump was closed before and is
// open after. All other pumps and the water level remain unchanged.
function PumpOpen(State before, int i) -> (State after)
// Valid pump required
requires ...
// Pump must have been closed
requires ...
// Pump must now be open
ensures ...
// All other pumps unchanged
ensures ...:
   ...
// Pump closed transition occurs when a pump was open before and is
// closed after. All other pumps and the water level remain unchanged.
function PumpClosed(State before, int i) -> (State after)
// Valid pump required
requires ...
// Pump must have been open
requires ...
// Pump must now be closed
ensures ...
// All other pumps unchanged
ensures ...:
   ...
```

Step 4. Complete the following functions for the transitions *water in* and *steam out*: **[5 marks]**

```
// Water in transition occurs when water is pumped into boiler. All
// pumps remain unchanged, whilst the water level increases.
function WaterIn(State before) -> (State after)
// Requires some pump to be open
requires ...
// All pump states unchanged
ensures ...
// Water level increases
ensures ...:
   ...
// Steam out transition occurs when steam leaves boiler. All
// pumps remain unchanged, whilst the water level decreases.
function SteamOut(State before) -> (State after)
// Need some water to remove
requires ...
// All pump states unchanged
ensures ...
// Water level decreases
ensures ...:
   ...
```

Submission

Your program code should be submitted electronically via the *online submission system*. Your code files should be clearly marked and given the "`.whiley`" extension. The set of requires files is:

```
append.whiley
byte.whiley
change.whiley
clock.whiley
compare.whiley
date.whiley
diff.whiley
insert.whiley
lastIndexOf.whiley
mid.whiley
natarray.whiley
natb.whiley
palindrome.whiley
reverse.whiley
some.whiley
SteamBoiler.whiley
sum3.whiley
swap.whiley
waterjugs.whiley
SteamBoiler.whiley
```

Appendix — Running Whiley

There are several different ways to run Whiley:

- **Cloud IDE.** Currently Whiley is running in the cloud at "`http://whileylabs.com`". Whilst this is the easiest way to run Whiley, *you should note that it may be unresponsive under high load (i.e. when lots of students are trying to access it).*

- **Local IDE**. You can run the Whiley IDE quite easily on your local machine by cloning the GitHub repository "`https://github.com/Whiley/WhileyWeb`". To do this, run the following:

```
% git clone git@github.com:Whiley/WhileyWeb.git
Cloning into 'WhileyWeb'...
remote: Counting objects: 220, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 220 (delta 0), reused 4 (delta 0), pack-reused 213
Receiving objects: 100% (220/220), 5.98 MiB | 163.00 KiB/s, done.
Resolving deltas: 100% (76/76), done.
Checking connectivity... done.
```

  You can run the web IDE from the command line using ant (if you have that installed) as follows:

```
% ant run
Buildfile: /Users/djp/projects/WhileyWeb/build.xml

compile:

run:
    [java] Port 80 in use by another application.
    [java] WhileyLabs running on port 8080.
```

  Then, just point your web browser to `http://localhost:8080` to access the IDE. If you don't have Ant installed, you can import this into Eclipse and (if necessary) convert it to a "Maven project".

- **ECS Machines.** On the ECS machines you can run the Whiley IDE using the following command from the terminal:

```
% /vol/courses/swen326/whiley/server.cgi
```

  Again, to access the local server point your web-browser to: `http://localhost:8080`.

- **Development Kit.** You can also download and install the latest *Whiley Development Kit (WDK)* from here: `https://github.com/Whiley/WhileyDevelopmentKit/`. This does not include the IDE and, instead, you will need to compile and run Whiley programs from the command-line.