Victoria University of Wellington
School of Engineering and Computer Science

SWEN224: Formal Foundations of Programming

Exam Prep Questions

1 Static Analysis

These questions are *illustrative* of the kinds of questions you could expect on static analysis in the exam.

1. **(5 marks)** Briefly, discuss what is meant by the term *static analysis*.

   Static analysis is a generic term referring to the use of tools at *compile-time* to automatically determine or check properties about programs. For example, static analysis tools (e.g. FindBugs) are often used to look for likely bugs in program code and can provide a guarantee that such bugs are absent.

2. **(5 marks)** Briefly, discuss an example of a *static analysis* that is in everyday use.

   The FindBugs tool is widely used for identifying potential bugs in program code. The tool employs simple "bug patterns" to look for situations where a bug is likely. However, the bug patterns can still report situations which look problematic but, in fact, are OK. The tool supports a wide-range of different bug patterns. For example, one bug pattern is to look for an `equals()` method where the parameter is not of type `Object`. This suggests a possible bug as the programmer was likely intending to override the default `equals()` method.

3. **(5 marks)** Briefly, discuss the difference between *compile-time* and *run-time*.

   Compile time occurs before the program is run, whilst run-time refers to the point when a programming is running. The difference affects the scope of any results reported by a tool. For example, a static analysis tool running at compile time will report results that are true for any possible run of the program. In contrast, a tool operating at runtime will report results which hold only for that particular run of the program (i.e. for the particular inputs that started the run).

4. **(5 marks)** Briefly, discuss how the use of static analysis can help to find errors in software.

   Static analysis tools, such as FindBugs, can help find errors in software because they can analyse the program to determine aspects of its runtime behaviour. For example, that a given variable may or may not hold the **`null`** value. Since static analysis tools operate at compile-time, they can report results which hold true for any possible run of the program. However, this also means they are susceptible to reporting *false-positives*.

5. **(5 marks)** Briefly, discuss what *conservatism* means in the context of static analysis.

   Conservatism refers to the fact that no static analysis tool can precisely estimate the runtime behaviour of a program. Thus, tools either *over-approximate* or *under-approximate* the results they report. Tools which over-approximate are said to be conservative as they are susceptible to reporting *false-positives*. On the other hand, tools which under approximate are said to be unsound as they are susceptible to reporting *false-negatives*.

6. **(6 marks)** For each *parameter*, *return* and *field* in the following program, insert `@NonNull` or `@Nullable` annotations (where appropriate) by writing in the box.

```java
public class ArraySet {
    private Object[] items;
    private int count; // counts number of elements currently used.

    public ArraySet(int n) {
        this.items = new Object[n];
        this.count = 0;
    }

    public void add(Object item) {
        items[count] = item;
        count = count + 1;
    }

    public boolean contains(Object o) {
        for(int i=0;i!=items.length;++i) {
            if(items[i].equals(o)) {
                return true;
            }
        }
        return false;
    }
}
```

```java
public class ArraySet {
    private @NonNull Object @NonNull [] items;
    private int count; // counts number of elements currently used.

    public ArraySet(int n) {
        this.items = new Object[n];
        this.count = 0;
    }

    public void add(@NonNull Object item) {
        items[count] = item;
        count = count + 1;
    }

    public boolean contains(Object o) {
        for(int i=0;i!=items.length;++i) {
            if(items[i].equals(o)) {
                return true;
            }
        }
        return false;
    }
}
```

2 Specification & Verification

These questions are *illustrative* of the kinds of questions you could expect on specification and verification in the exam.

1. **(5 marks)** In your own words, discuss the benefits of providing specifications for code.

   Specifications are useful because they clarify the expected input/output behaviour of a function or of a program. In particular, this can help to attribute *blame* when a fault occurs. Thus, if the requirements on inputs values are not met and a fault arose, it is the callers reponisbility. Or, if the input values given were valid, but an invalid output value was returned then it is the implementors reponsibility

2. **(5 marks)** Specifications are often described as contracts between the *client* and *supplier*. Briefly, discuss what is meant by this and how the two roles interact.

   The client refers to some part of a program which calls another part provided by the supplier. The client and supplier code could be part of the same program, or the supplier could be part of an external library being called by the client. The two roles interact through the given specifications. The client must ensure that the precondition of a function is met when it is called. The supplier must ensure that the postcondition of a function is met, assuming that the precondition was met by the client. Finally, the client can assume the post-condition holds after a function is called.

3. **(2 marks)** State in English what the following specification says:

```
function add(int x, int y) -> (int z)
requires 0 <= x && x <= 5
requires 0 <= y && y <= 5
ensures 0 <= z && z <= 10:
    //
    return x + y
```

   That the values of `x` and `y` must be between `0` and `5` (inclusive), whilst the value returned will be between `0` and `10` (inclusive).

4. **(2 marks)** State in English what the following specification says:

```
function zeroOut(int[] items, int start) -> (int [] r)
requires start >= 0 && start < |items|
ensures all { k in start..|r| | r[k] == 0 }:
    //...
```

   That all elements including and above a given point in the array (`start`) are set to zero.

5. **(3 marks)** Briefly, discuss whether or not the following implementation of `zeroOut()` meet its specification.

```
function zeroOut(int[] items, int start) -> (int [] r)
requires start >= 0 && start < |items|
ensures all { k in start..|r| | r[k] == 0 }:
    //...
    int i = 0
    while i < |items|:
        items[i] = 0
        i = i + 1
    //
    return items
```

Yes, this implementation does meet its specification though perhaps it is not what was intended. In particular, by ensuring that all elements in the array are set to zero, it is meeting the postcondition that all above a given point are set to zero.

6. **(3 marks)** Provide an appropriate *loop invariant* for the `zeroOut()` function above.

```
function zeroOut(int[] items, int start) -> (int [] r)
requires start >= 0 && start < |items|
ensures all { k in start..|r| | r[k] == 0 }:
    //...
    int i = 0
    while i < |items|
    where i >= 0
    where all { k in 0..i | items[k] == 0 }:
        items[i] = 0
        i = i + 1
    //
    return items
```

7. **(5 marks)** Loop invariants differ from pre- and post-conditions as they do not form part of a function's specification. Briefly, discuss what the purpose of a loop invariant is.

Preconditions and postconditions are required to form a functions' specification. However, a loop invariant is required only to help the verifier check that a function meets its specification. Thus, loop invariants differ from pre- and post-conditions because they are not fundamentally required but, rather, exist only as an artifact of the way that current tools perform verification.

8. **(5 marks)** Briefly, discuss why the following implementation *does not* meet its specification. You should provide *parameter* and *return* values to illustrate.

```
 1  function isSorted(int[] arr) -> (bool r)
 2  requires |arr| > 0
 3  ensures r ==> all { i in 1 .. |arr| | arr[i-1] <= arr[i] }
 4  ensures !r ==> some { i in 1 .. |arr| | arr[i-1] > arr[i] }:
 5      //
 6      int i = 1
 7      int last = arr[0]
 8      //
 9      while i < |arr|:
10          //
11          if last > arr[i]:
12              return false
13          i = i + 1
14          last = arr[i]
15      //
16      return true
```

This function does not meet its specification because it increments `i` before updating the `last` variable. Furthermore, any input for which the loop iterates twice or more will result in a runtime crash. For example, the parameter value `arr=[3,4]` would cause an out-of-bounds error on line 14.

9. **(8 marks)** Provide an updated implementation of `isSorted()` which does meet its specification. You should additionally include an appropriate *loop invariant* to ensure that it will verify.

```
function isSorted(int[] arr) -> (bool r)
requires |arr| > 0
ensures r ==> all { i in 1 .. |arr| | arr[i-1] <= arr[i] }
ensures !r ==> some { i in 1 .. |arr| | arr[i-1] > arr[i] }:
    //
    int i = 1
    int last = arr[0]
    //
    while i < |arr|
    where i >= 0
    where all { k in 1 .. i | arr[k-1] <= arr[k] }:
        //
        if last > arr[i]:
            return false
        last = arr[i]
        i = i + 1
    //
    return true
```

10. **(6 marks)** Next to each numbered comment in the program below, give appropriate logical conditions which are true at that point in the program.

```
function diff(int x, int y) -> (int z)
requires x >= 0 && x <= 255 && y >= 0 && y <= 255
ensures z >= 0
ensures z == (x-y) || z == (y-x):
    //
    int r
    //
    //(1)
    //
    if x > y:
        //
        //(2)
        //
        r = x - y
        //
        //(3)
        //
    else:
        //
        //(4)
        //
        r = y - x
        //
        //(5)
        //
    //
    return r
```

```
function diff(int x, int y) -> (int z)
requires x >= 0 && x <= 255 && y >= 0 && y <= 255
ensures z >= 0
ensures z == (x-y) || z == (y-x):
    //
    int r
    //
    //(1) 0 ≤ x ≤ 255 ∧ 0 ≤ y ≤ 255
    //
    if x > y:
        //
        //(2) x > y ∧ (1)
        //
        r = x - y
        //
        //(3) r == x - y ∧ (2)
        //
    else:
        //
        //(4) x ≤ y ∧ (1)
        //
        r = y - x
        //
        //(5) r == y - x ∧ (4)
        //
    //
    return r
```

11. **(5 marks)** Briefly, discuss why the precondition given for the following function is not the *weakest precondition*.

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

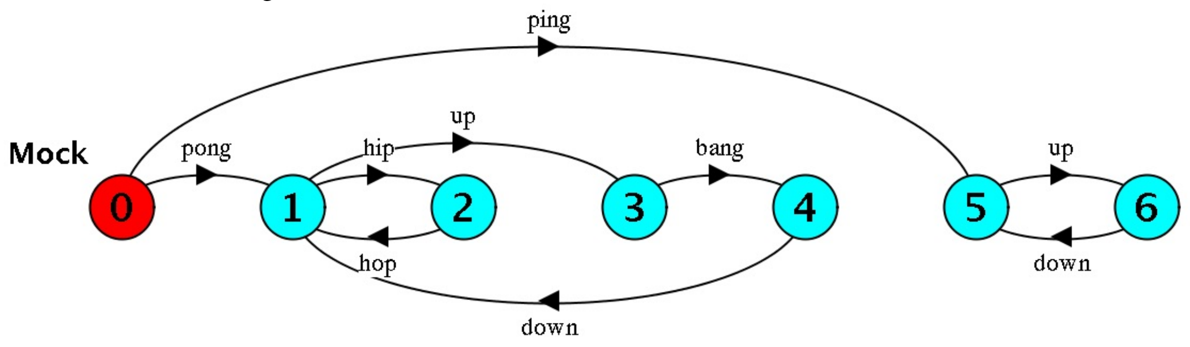
The weakest precondition is the least restrictive precondition necessary to ensure the postcondition. In this case, however, a preconditoon of `true` is sufficient to show the postcondition and, hence, the given precondition is not weakest.

12. **(5 marks)** The Whiley verification system ensures that functions are *partially correct* but it does not ensure they are *totally correct*. Briefly, discuss what this means.

Totally correct requires that the function is guaranteed to meet its specification and also guaranteed to terminate. Partially correct, however, means only that the function is guaranteed to meets its specification if it terminates, but not that it is guaranteed to terminate.

3 Checking Requirements:

These questions are *illustrative* of the kinds of questions you could expect on requirements checking in the exam. Consider the following automaton.



Will **Mock**, above, satisfy the following statements (Yes/No)

1. **(2 marks)** `progress T = {hip}`
2. **(2 marks)** `progress T = {hop,down}`.
3. **(2 marks)** `property Sh = (pong->hip->down->Sh).`
4. **(2 marks)** `property St = (up->down->St)..`

4 Modeling

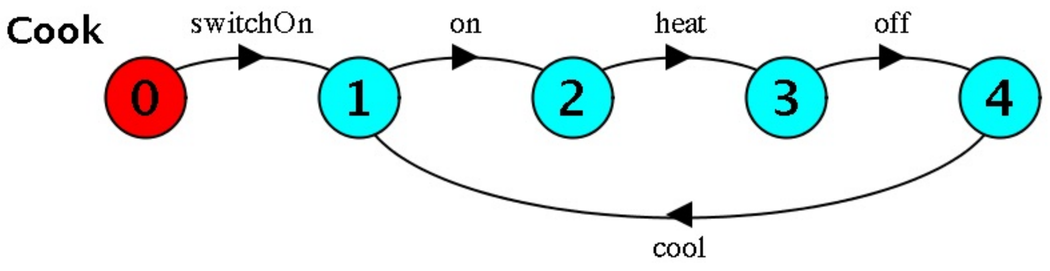
Cook Modeling a Simple cooker

When a cooker, **Cook** is switched on, **switchOn** you can assume that the cooker is cold and the element will be turned **on** and the cooker will **heat** up. Once the cooker is hot it will turn **off** and **cool** before turning itself **on** again.

Requirement. Once the cooker **Cook** is switched on it can not be switched off and forever cycles around being cold then hot then cold, ...

Only use events: **switchOn**, **on**, **heat**, **off** and **cool**.

1. **(2 marks)** Draw the automaton of the **Cook** automaton.



2. **(2 marks)** Specify the **Cook** process using the LTSA language.

$Cook = (switchOn- > Coff),$
$Coff = (on- > Con),$
$Con = (heat- > Hon),$
$Hon = (off- > Hoff),$
$Hoff = (cool- > Coff).$

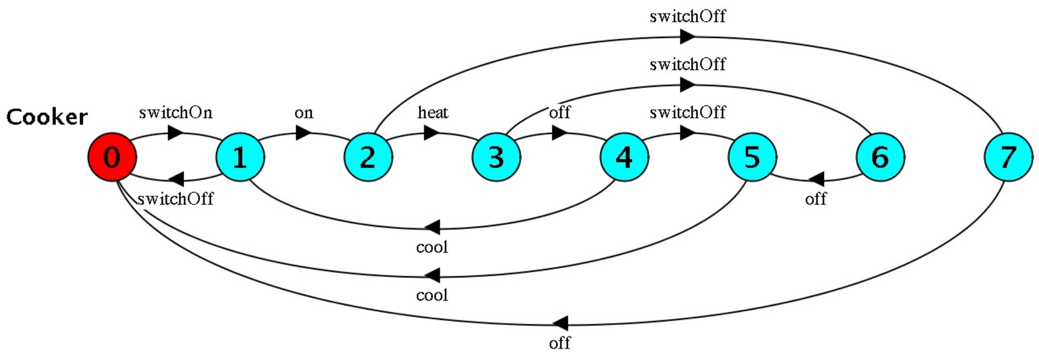
3. **(2 marks)** Briefly, discuss whether the **requirement** given above is a *safety* or *liveness* requirement.

Cooker a simple cooker with an off switch.

Build a **Cooker** process by extending the **Cook** process in the previous question, by adding the ability to switch off the cooker. After the user switches off the cooker, the **switchOff** event, the cooker must turn itself off if it is currently on. When off the cooker may still need to cool down before it returns to it initial state.

Only use events: **switchOn**, **switchOff**, **on**, **heat**, **off** and **cool**.

1. **(3 marks)** Draw the automaton of the **Cooker** automaton.



2. **(3 marks)** Specify the **Cooker** process using the LTSA language.

$Cooker = (switchOn- > Coff),$

$Coff = (on- > Con|switchOff- > Cooker),$

$Con = (heat- > Hon|switchOff- > off- > Cooker),$

$Hon = (off- > Hoff|switchOff- > off- > Hot),$

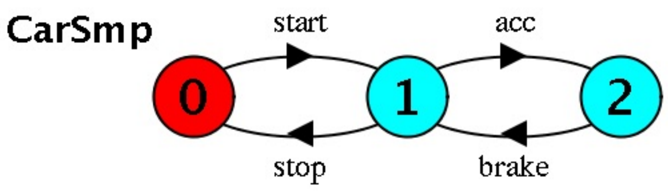
$Hoff = (cool- > Coff|switchOff- > Hot),$

$Hot = (cool- > Cooker).$

Car a simple specification

A **CarSmp** is initially at rest and not running. It can **start** after which it is running. when running it can **stop**. In addition the car can **acc**elerate and then **break**. Before it has started it cannot accelerate.

1. **(3 marks)** Draw the automaton of the **CarSmp** specified above using only the events **start**, **stop**, **acc** and **break**.



2. **(3 marks)** Specify the **CarSmp** process using the LTSA language.

$CarSmp = (start- > C),$

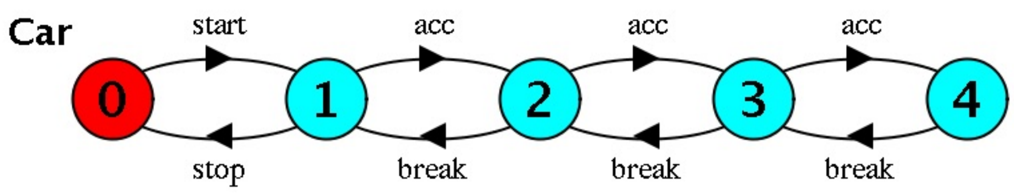
$C = (acc- > Go|stop- > Carmp),$

$Go = (brake- > C)$

Car again

Car is like the previous **CarSmp** model except that is can travel at speeds 0,1,2,3,....N and to accelerate from rest to speed i it must **acc**elerate i times, and to stop from speed i it must **brake** i times.

1. **(2 marks)** Draw the automaton of the **CarSmp** specified above using only the events **start**, **stop**, **acc** and **brake** when **N=3**



2. **(2 marks)** Specify the **Car** process using the LTSA language.

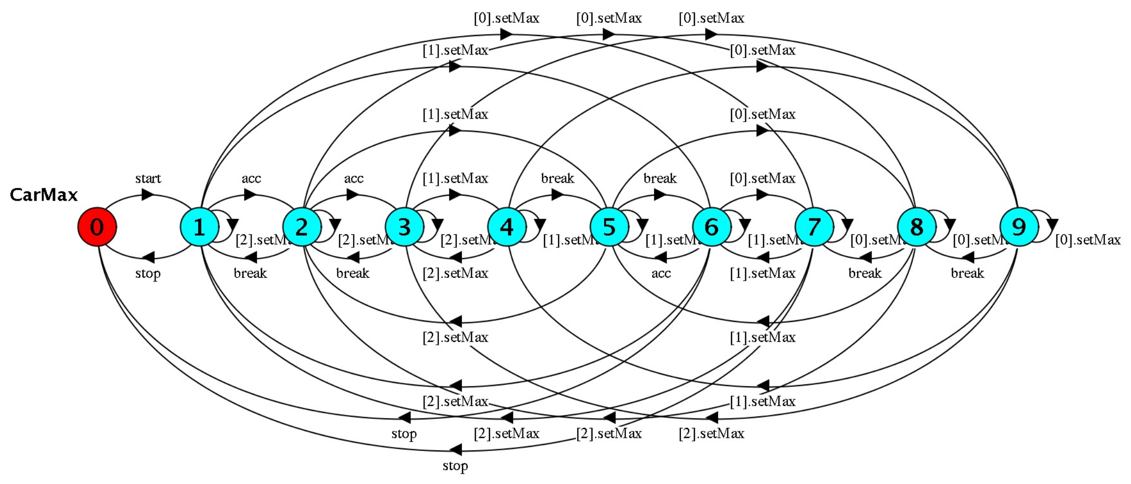
$Car = (start- > C[0]),$
$C[i : 0..N] = (when(i < N)acc- > C[i + 1]$
$|when(i > 0)break- > C[i - 1]$
$|when(i == 0)stop- > Car).$

CarMax and indexed process.

The **CarMax** process is the **Car** process from the previous question with the ability to set the maximum speed of the car to some input parameter.

Only use the events **start**, **stop**, **acc**, **brake** and **setMax**

1. **(5 marks)** Draw the automaton of the **CarMax** specified above when the constant **N = 2**.



2. **(5 marks)** Specify the **CarMax** process using the LTSA language.

$CarM = (start- > C[0][N]),$
$C[i : 0..N][j : 0..N] = (when(i < N\&i < j)acc- > C[i + 1][j]$
$|when(i > 0)break- > C[i - 1][j]$
$|when(i == 0)stop- > Car$
$|when(i =!j)[x : 0..N].setMax- > C[i][x]).$