Surname: . . . . **First Name:** . . . . **Student ID:** . . . .

Te Herenga Waka—Victoria University of Wellington

EXAMINATIONS – 2022

TRIMESTER 1

SWEN 326

SAFETY-CRITICAL SYSTEMS

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Instructions: Answer all questions

Question	Topic	Marks
1.	Risk, Hazards and Failure	30
2.	Testing	30
3.	Static Analysis	30
4.	Design Validation	30
	Total	120

1. Risk, Hazards and Failure **(30 marks)**

(a) Consider the following description of a system for controlling a *model rocket*:

> "The rocket is operated by a *software controller*. If the *navigation sensor* detects that the trajectory is too low, the engine is *turned off* and the parachute is *deployed*. Under no circumstance should the rocket be allowed to travel at speed towards the ground."

i. **(2 marks)** Following the terminology of IEC61508, identify the *Equipment Under Control* for the rocket system.

The rocket

ii. **(2 marks)** Identify an important *hazard* for the rocket system.

The rocket crashing into the ground

iii. **(2 marks)** Briefly, discuss what is required to estimate the *risk* posed by the above hazard.

We need to know the likelihood of a dangerous trajectory happening.

iv. **(2 marks)** Following the terminology of IEC61508, identify which part of the system is relied upon to ensure *functional safety*.

The navigation sensor

v. **(2 marks)** Briefly, discuss how the above hazard is *mitigated* in the system.

It is mitigated by stopping the rocket if things appear to be going wrong, and deploying a shoot to ensure a safe landing

(Question 1 continued on next page)

(Question 1 continued)

vi. **(4 marks)** Under IEC61508 there is always *residual risk*. Briefly, discuss what this means with respect to the rocket system.

This means that, despite whatever risk mitigation systems are in place, there is always a chance they will still fail. The residual risk is the leftover risk after applying the mitigation. For example, in the rocket system, there is the potential for a problem with the shoot (e.g. it doesn't open properly).

(b) **(4 marks)** Briefly, discuss why the possibility of a *multiple component failure* can be particularly problematic.

- Multiple component failures are significantly less likely than a single component failure.
- Multiple component failures can be hard to identify. For example, if we have two sensors watching a pump, and both fail at the same time we might be unable to tell if/when the pump fails.
- To handling multiple component failures, additional redundancy in the design is required which leads to more cost. Furthermore, we still need to bound the number of components which could fail at the same time.

(c) **(4 marks)** Rule 3 from the *"Power-of-Ten"* coding guidelines prohibits the use of *"dynamic memory allocation after initialization"*. Briefly, discuss the motivation behind this rule.

- Manual memory allocation through malloc / free is prone to memory leaks and other bugs (e.g. free after free, or failing to check whether malloc was successful, etc)
- Garbage collection, on the other hand, has unpredictable performance which can lead to performance spikes.
- Dynamic memory allocation has the potential to exhaust available memory and it is difficult to assess whether or not this could happen. Hence, allocating all memory at initialisation means that once initialisation is passed, this form of error is no longer possible.

(Question 1 continued on next page)

(Question 1 continued)

(d) **(4 marks)** The SWEN326 Java Coding Standard requires that *"JavaDoc errors must be enabled"*. Briefly, discuss the pros/cons of this requirement.

- The JavaDoc checker (e.g. in Eclipse) helps to ensure that JavaDoc comments are provided and, furthermore, are consistent (e.g. that parameters and returns are commented).
- However, the JavaDoc checker cannot ensure that JavaDoc comments are accurate or well written.
- Enabling JavaDoc checking also imposes a cost on development time that, in some cases, many not be considered cost effective.

(e) **(4 marks)** The SWEN326 Java Coding Standard used for the steam boiler assignment requires that "Eclipse's non-null analysis is enabled". Briefly, discuss the pros/cons of this requirement.

- `NullPointerExceptions` are a significant cause of failure in Java programs. Enabling Eclipse's non-null analysis can dramatically reduce the likelihood of such an error occurring in practice.
- Enabling non-null analysis also has the benefit that it ensures good documentation for methods regarding whether their parameters can be **`null`** or not.
- Unfortunately, Eclipse's non-null analysis does often require the source code to be changed in ways that allow the analysis to pass, which can be frustrating and confusing. Likewise, writing annotations can also be time consuming.

SPARE PAGE FOR EXTRA ANSWERS

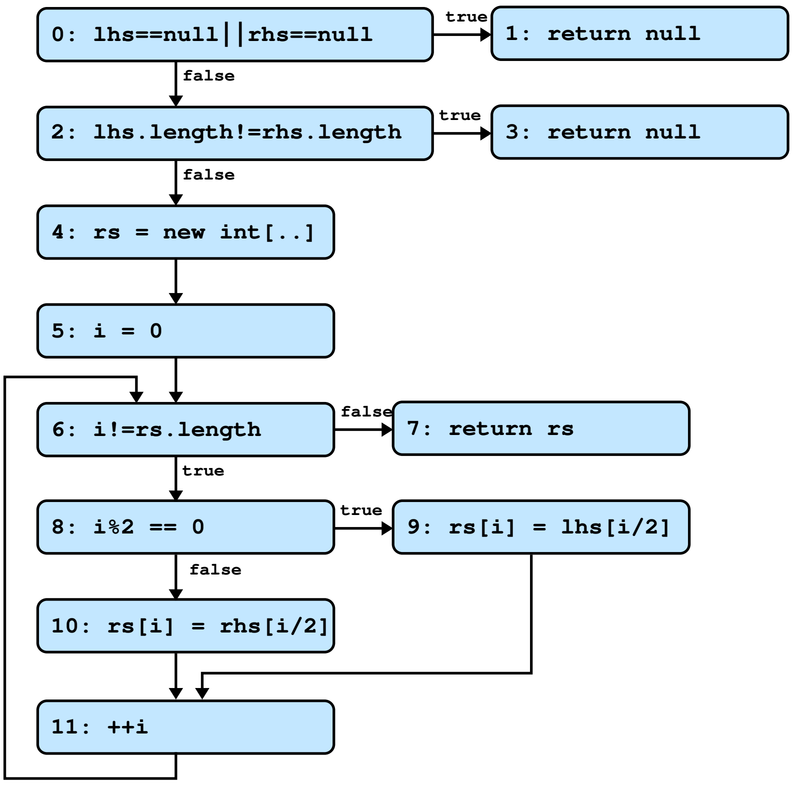
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

2. Testing **(30 marks)**

Consider the following Java method which compiles without error:

```java
public int[] zip(int[] lhs, int[] rhs) {
   if(lhs == null || rhs == null) {
      return null;
   } else if(lhs.length != rhs.length) {
      return null;
   } else {
      int[] rs = new int[lhs.length + rhs.length];
      // Merge like a zip!
      for(int i=0;i!=rs.length;++i) {
         if(i%2 == 0) { rs[i] = lhs[i / 2]; }
         else { rs[i] = rhs[i / 2]; }
      }
      // Done
      return rs;
} }
```

(a) **(10 marks)** Draw the *control-flow graph* for the `zip()` method.



(Question 2 continued on next page)

(Question 2 continued)

(b) **(6 marks)** Write three JUnit tests which achieve 100% *branch coverage* of `zip()`.

```
    @Test public void test_1() {
       int[] lhs = {1,2,3};
       int[] r = zip(lhs,null);
       assert r == null;

    }
```

```
    @Test public void test_2() {
       int[] lhs = {1,2,3};
       int[] r = zip(lhs,new int[0]);
       assert r == null;

    }
```

```
    @Test public void test_3() {
       int[] lhs = {1,2,3};
       int[] rhs = {4,5,6};
       int[] r = zip(lhs,rhs);
       assert Arrays.equals(r, new int[]{1,4,2,5,3,6});
    }
```

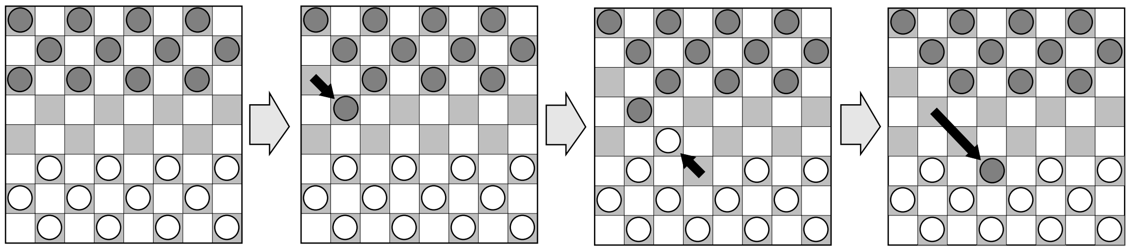
(c) **(3 marks)** Briefly, discuss whether additional tests are required to achieve 100% *Modified Condition/Decision Coverage* (MC/DC) for the `zip()` method.

Yes, additional tests are required to achieve full MC/DC coverage of `zip()`. MC/DC requires each condition to have both outcomes tested. With the tests given above, the first decision in `zip()` has the right condition tested (`rhs==null`) but not the left (`lhs==null`).

(Question 2 continued on next page)

(Question 2 continued)

(d) In the game of *drafts* there are two players. The first player uses *black pieces* and the second player uses *white pieces*. The *board* consists of 64 squares arranged in an 8x8 grid. Each square holds at most one piece. Players take turns *moving* their pieces one square at a time along the diagonals of the board. Players can *capture* their opponent's pieces by jumping over them into an empty square. The following illustrates the first three moves of a game:



Suppose we have a program which implements the rules of drafts.

i. **(5 marks)** Briefly, discuss what a *test oracle* for this program might look like.

A test oracle for this program could be another program which correctly implements the game (e.g. a *model solution*). The key is that it must provide a means to decide whether a sequence of moves in the game is *valid* or *invalid*. Without this we cannot generate tests for the program automatically.

ii. **(6 marks)** Suppose we test our program by generating random games consisting of *at most two moves* by each player. Briefly, discuss the pros / cons of this approach:

- Restricting the number of moves to just two dramatically reduces the search space and, hence, provides a quick and easy means of testing the program.
- It is also likely that it will obtain good code coverage (i.e. that it is *effective* at testing the program).
- However, with only two moves being considered many aspects of the game will remain untested. For example, whether or not a player has one the game, or a stalemate was reached. Similarly, the situation where one player takes the piece of another cannot be tested.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

3. Static Analysis **(30 marks)**

(a) This question is concerned with *static analysis*.

i. **(3 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. Find-Bugs) are often used to look for likely bugs in program code and can provide a guarantee that such bugs are absent.

ii. **(3 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).

(b) This question is concerned with *non-null* analysis.

i. **(2 marks)** Briefly, discuss what the `@Nullable` annotation means.

This means the annotated variable can hold the **`null`** value.

ii. **(2 marks)** Briefly, discuss what "*non-null by default*" means for a non-null analysis.

Non-null by default instructions the non-null checker to assume that an unannotated variable is non-null. This typically reduces the number of annotations that must be written, since most references are non-null.

iii. **(2 marks)** Briefly, discuss what a *false negative* means with respect to a non-null analysis.

A false-negative is a problem in the code (i.e. something which can cause a `NonNullException`) that the non-null checker doesn't notice. For example, the handling of `@NonNull` fields is *unsound* because of limitations in the language's design.

(Question 3 continued on next page)

(Question 3 continued)

iv. **(9 marks)** For each *parameter*, *return* and *field* in the following program, insert `@NonNull` or `@Nullable` annotations (as appropriate) by writing in the box. You should not assume non-null by default.

```java
class Point {

  public final int x, y;

  public Point(int x, int y) { this.x = x; this.y = y; }
}

interface Shape { boolean contains(Point p); }

class Rectangle implements Shape {

  private final Point tl;

  private final Point br;

  public Rectangle(int x, int y, int width, int height) {
    tl = new Point(x,y);
    br = new Point(x+width-1,y+height-1);
  }

  public boolean contains(Point p) {
    return (tl.x <= p.x) && (p.x <= br.x) &&
           (tl.y <= p.y) && (p.y <= br.y);
  }
}

class Union implements Shape {

  private final Shape l;

  private final Shape r;

  public Union(Shape lhs, Shape rhs) { l = lhs; r = rhs; }

  public boolean contains(Point p) {
    if(p == null) { return false; }
    else if(l==null && r==null) { return false; }
    else if(l==null && r!=null) { return r.contains(p); }
    else if(l!=null && r==null) { return l.contains(p); }
    return l.contains(p) || r.contains(p);
  }
}
```

(Question 3 continued on next page)

(Question 3 continued)

v. **(3 marks)** The `assert` statement can help with foreign code that has not been annotated (e.g. the standard library). Using an example to illustrate, explain how this works.

The `assert` statement allows us to override what variables the non-null checker assumes can hold **`null`**. For example, when working with the standard library many `toString()` methods return non-null objects, but unfortunately the library is not annotated accordingly. An example would be something like this:

```
class toString { int[] items;
  @NonNull String toString() {
    String s = Arrays.toString(items);
    assert s != null;
    return s;
} }
```

Consider the following program written in Java:

```
1  class BoundedList {
2    private Object @NonNull [] items;
3    private int length;
4
5    public BoundedList(int size) { items = new Object[size]; }
6
7    public void add(@NonNull Object x) {
8       if(length < items.length) { items[length++] = x; }
9    }
10   public @NonNull Object get(int i) {
11     if (i < length) { return items[i]; }
12     throw new ArrayIndexOutOfBoundsException();
13 } }
```

vi. **(2 marks)** The above program fails non-null checking. Briefly, identify what the problem is.

The problem is that elements in the `items` array can be **`null`**, therefore on line 11 we cannot return `items[i]` as `@NonNull Object`.

vii. **(4 marks)** Briefly, discuss why the above program presents an inherent challenge for a non-null analysis.

The program presents a challenge because the `items` array really can hold **`null`** values. For example, elements in `items` above `length` may be **`null`**. In essence, the class maintains a subtle invariant the all elements upto `length` are non-null, but this cannot be described in the type system.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

4. Design Validation **(30 marks)**

(a) For each of the following, provide one set of parameter values which meet the precondition and one set which does not.

i. **(2 marks)**

```
function set(int[] items, int i, int val) -> (int[] r)
requires 0 <= i && i < |items|:
   //...
```

Meets preconditon: `items=[0],i=0,val=0`

Does not meet precondition: `items=[],i=0,val=0`

ii. **(2 marks)**

```
function indexOf(int[] items, int val) -> (int r)
requires some { i in 0..|items| | items[i] == val }:
   //...
```

Meets preconditon: `items=[0],val=0`

Does not meet precondition: `items=[],val=0`

iii. **(2 marks)**

```
function reverse(int[] bs) -> (int[] r)
requires all { i in 0..|bs| | bs[i] >= 0 && bs[i] <= 255 }:
   //...
```

Meets preconditon: `bs=[0]`

Does not meet precondition: `bs=[-1]`

iv. **(2 marks)**

```
function find(int[] xs, int val) -> (int r)
requires |xs| > 1 ==> all { i in 1..|xs| | xs[i-1] < xs[i] }:
   //...
```

Meets preconditon: `xs=[0,1,3],val=0`

Does not meet precondition: `xs=[3,2,1],val=0`

(Question 4 continued on next page)

(Question 4 continued)

(b) Consider the following implementation for the function `fill()`.

```
function fill(int[] items, int item, int index) -> (int[] r):
   //
   if index == |items|:
      return items
   else:
      items[index] = item
      return fill(items, item, index+1)
```

i. **(2 marks)** Describe in your own words what the function `fill()` does.

The fill function sets all elements of the array `items` from `index` upto the end of the array to `item`

ii. **(5 marks)** Provide an appropriate specification for function `fill()`.

```
requires 0 <= index && index <= |items|
ensures |items| == |r|
ensures all { i in index .. |r| | r[i] == item }
ensures all { i in 0 .. index | r[i] == items[i] }
```

(Question 4 continued on next page)

(Question 4 continued)

(c) Consider the following implementation for the function `cut()`:

```
function cut(int[] xs) -> (int[] rs)
ensures |rs| == |xs|
ensures all { k in 0..|xs| | xs[k] >= 0 ==> rs[k] == xs[k] }
ensures all { k in 0..|xs| | xs[k] < 0 ==> rs[k] == 0 }:
   //
   int i = 0
   int[] ys = xs
   //
   while i < |xs|:
      if xs[i] < 0:
         ys[i] = 0
      else:
         ys[i] = xs[i]
      i = i + 1
   //
   return ys
```

i. **(2 marks)** Briefly, describe in your own words what function `cut()` does.

The cut function takes an array `xs` and returns an updated array where all negative elements are replaced with `0`, and all other elements are left as is.

ii. **(4 marks)** Provide an appropriate *loop invariant* for function `cut()`.

```
   while i < |xs|
   where i >= 0 && |xs| == |ys|
   where all {j in 0..i | xs[j] < 0 ==> ys[j] == 0 }
   where all {j in 0..i | xs[j] >= 0 ==> ys[j] == xs[j] }:
```

(Question 4 continued on next page)

(Question 4 continued)

(d) Consider the following implementation for the function `cmp()`:

```
function cmp(int[] left, int[] right, int i) -> (int r):
   if i == |left|:
      return 0
   else if left[i] < right[i]:
      return -1
   else if left[i] > right[i]:
      return 1
   else:
      return cmp(left,right,i+1)
```

i. **(2 marks)** Briefly, describe in your own words what function `cmp()` does.

The `cmp()` function finds the first position `i` in which both arrays differ and indicates whether `left[i]` is below `right[i]` at that point. If no such position exists, then it returns `0` to indicate they are equal.

ii. **(7 marks)** Provide an appropriate specification for function `cmp()`.

```
function cmp(int[] left, int[] right, int i) -> (int r)
requires |left| == |right|

requires i >= 0 && i <= |left|

requires all { j in 0..i | left[j] == right[j] }

ensures r >= -1 && r <= 1

ensures r == 0 ==> all { j in i..|left| | left[j] == right[j] }

ensures r < 0 ==> some { j in i..|left| | left[j] < right[j]
                  && all { k in 0..j | left[k] == right[k] } }
```

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.