**BCT 2313: FORMAL SOFTWARE SPECIFICATION METHODS**

**NAME: STANLEY NGUGI REG NUM: SCT212-0065/2017**

1. **Justify with four areas of commercial use the following statement “Formal specification methods very significant applications in software engineering ( 2 Marks )**

* Eliminating Design Errors in Your Algorithm Using **Simulink Design Verifier** in Matlab. Formal Specification Methods help engineers to verify that your embedded system software models and code behave correctly
* **Polyspace Bug Finder** identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Through Formal Specification methods, defects can be highlighted as soon as they are detected, facilitating fix bugs early in the development process.
* Formal Methods have been used to test the correctness of Kernel in operating systems such as in SEL4, a third-generation microkernel of L4 which comprises of 8700 lines of C code and 600 lines of assembler
* Formal methods have been used to test the technical correctness and security of open source code such as in NuXmv, Framed-C and Klee.

1. **Is the following a tautology? Justify your answer with a truth table table [ (P ⇒ Q) ∧ (Q ⇒ R) ] ⇒ (P ⇒ R) (3 Marks)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **R** | **P 🡪Q** | **Q🡪R** | **^** | **P🡪R** | **^🡪P🡪R** |
| **T** | **T** | **T** | T | T | T | T | T |
| **T** | **T** | **F** | T | F | F | F | T |
| **T** | **F** | **T** | F | T | F | T | T |
| **T** | **F** | **F** | F | T | F | F | T |
| **F** | **T** | **T** | T | T | T | T | T |
| **F** | **T** | **F** | T | F | F | T | T |
| **F** | **F** | **T** | T | T | T | T | T |
| **F** | **F** | **F** | T | T | T | T | T |

The final column returns all true proving that the expression **is** a tautology

1. **With aid of a working example, explain the properties of safety and liveness as applied in temporal logic (2 Marks)**

* Safety property assures that something bad will not happen. For instance, finite-length error trace.
* Liveness property assueress that something good will happen. For example, infinite-length error trace

1. **Compute the weakest precondition for the following statement and post conditions**
2. **X=2\* y-3 {x>25} (1 Marks)**

Post condition is { x> 25}

Substitute 2\*y-3 for x

2\*y-3 > 25

2y > 28

y> 14

Thus y> 14 is the weakest precondition

wp(x=2\* y-3,x>25) = y > 14

1. **Compute the weakest precondition fo the following if statement:**

*if (a == b)*

*b = 2 \* a + 1;*

*else*

*b = 2 \* a;*

*{b > 1}*  **(2 Marks)**

**If**(a == b)

b = 2 \* a + 1;

**else**

b = 2 \* a;

{b> 1}

wp(IF, b > 1) = (a == b ᴧwp(b = 2\*a+1 , b > 1)) ᴠ (a != b ᴧ wp(b = 2 \* a , b > 1))

= ( a == b ᴧ 2a > 0 ) ᴠ (a != b ᴧ 2a > 1)

= ( a == b ᴧ a > 0 ) ᴠ (a != b ᴧ a > 0.5)

= (a> 0) ᴠ (a > 0.5)

= a > 0

1. **Describe briefly the following specification paradigms (2 Marks)**
   1. **History Based Specification**

This specification involves characterizing a system with reference to the set of maximal behaviors of the system over time, that is, the historical perfomance of the system.

The properties of interest are specified by temporal logic assertions about system objects and such assertions include operators referring to past, current and future states. Assertions are interpreted over time structures, in which time can be linear or branching while time structures can be discrete, dense or continuous.

* 1. **Transition Based Specifications**

This specification involves characterizing the system as if it were a state machine. This includes the transition of the machine across the various states accepted by the machine which is governed by the set of transition functions of the machine. The transition function for a system object gives, for each input state and triggering event, the corresponding output state.

1. **Using First Order Predicate Calculus, represent the following (2 Marks)**
   1. *All football players are strong*

∀ X( basketball\_player (X) 🡪 strong ( X ) )

* 1. *Nobody likes githeri*

¬Xlikes ( X,githeri )

* 1. *Some people like Omena*

∃ X ( person(X) ∧ likes ( X, omena ) )

* 1. *If it does not rain on Monday, Jane will attend Lectures*

¬ weather( rain, Monday ) 🡪 attend(Jane, lectures)

1. **Prove by induction that**

**for all positive integer values of n**. (3 Marks)

**Base case:** when n = 1 , the left side is r2 = (1)2 = 1 and the right side is

1/6(2)(3) = 1 so both sides are equal and (1) is true for n = 1;

**Induction step:** let *k* be a positive integer and is given and suppose (1) is true for n = k.

Then

= + (k+1)2

= + (k+1)2

=  + k2 + 2k + 1

= (2k2 + k + 2k + 1 ) + k2 + 2k + 1

= (2k2 + 3k + 1)

1. **Write a loop to set sum = 1 + 2 + … + n and prove that it is correct. (3 Marks)**

**Loop**

*int total\_sum = 0;*

*int counter=1;*

*while( counter != 11){*

*total\_sum = total\_sum + 1;*

*counter ++;*

*}*

**Proof:**

**Part 2**

1. **Describe the Hoares Rules and how they are applied to total and partial specification 10 Marks**

**Hoare Logic**

1. **Assignment rule**

The most central aspect of imperative language is reduced to simple syntactic formula substitution.

Where *x*is any variable, *e*is any expression, *p* is any statement.

*Examples*: {38 = 38} x:= 38 {x = 38}

{y = 2} x:= 2 {y=x}

1. **A forward assignment rule**

This is the original semantics of assignment due to Floyd.

{p} x := e {∃v. x = e[v/x] ∧ p[v/x]}

Here *v*  is a fresh variable, that is, *v*  doesn’t equal  *x* or occur in *p* or *e*.

*Example:*

{x=1} x:=x+1 {∃v. X = X+1[v/X] ∧ X=1[v/x]}

*Simplifying the postcondition*

{x=1} x:=x+1 {∃v. x = x+1[v/x] ∧ x=1[v/x]}

{X=1} x:=X+1 {∃v. X = v + 1 ∧ v = 1}

{x=1} x:=x+1 {∃v. x = 1 + 1 ∧ v = 1}

{x=1} x:=x+1 {x = 1 + 1 ∧∃v. v = 1}

{x=1} x:=x+1 {x = 2 ∧ T}

{x=1} x:=x+1 {x = 2}

Forward rule is equivalent to the standard one but harder to use.

1. **Strengthening precondition and weakening postcondition**

*Strengthening precedent (SP)*

(sp)

*Weakening consequent (WC)*

(wc)

*Example:*

{x = n} x := x + 1 {x = n + 1}

N is a logical variable.

Proof:

x = n ⇒ x + 1 = n + 1 (predicate logic)

{x + 1 = n + 1} x := x + 1 {x = n + 1} (Assignment rule)

{x = n} x := x + 1 {x = n + 1} (strengthening precedent 1,2)

1. **The consequent rule**

The rules Strengthening precedent(sp) and weakening consequent are sometimes merged to the *consequence rule.*

This rule is not syntax directed.

1. **The sequential composition rule**

Syntax: c1; · · ·; cn

Semantics: the commands : c1; · · ·; cnare executed in that order.

Example: r := x ;x := y ; y := r

the values of x and y are swapped using r as a temporary variable

1. **The skip rule**

{p}skip{p}

1. **Conditional rule**

1. **While rule.**

The *partial correctness of* ***while:***

i is called the loop invariant.

It says that

- if executing c once preserves the truth of i, then executing c any number of times also preserves the truth of i

- after a while command has terminated, the test must be false

The while rule for total correctness

The while commands are the only commands in our simple language that can cause non-termination. They are thus the only kind of command with a non-trivial termination rule.

The idea behind the while rule for total correctness is:

-to prove while b do c terminates

-show that some non-negative metric (e.g. a loop counter) decreases on each iteration of c

-this decreasing metric is called a variant.

where x0 fv(c) ∪fv(e) ∪fv(i) ∪fv(b).

Application to total and partial specification

The assignment rule, forward assignment rule, skip rule, conditional rule, consequent rule, sequential composition rule , Strengthening precondition and weakening postconditionhold partial and total correctness. They are useful in slitting a proof into independent bits.

The while rule is the major rule that distinguishes the logic for total correctness from partial correctness.

1. **Prove the correctness of the following Hoare triple showing the various rules applied . 10 Marks**

*{ x = x0 }*

*[]*

*x = x - 3;*

*if (x < 0) {*

*x = 1;*

*} else {*

*if (true) {*

*x = x + 1;*

*} else {*

*x = 10;*

*}*

*}*

*{ (x0 < 3 -> x = 1) & (x0 >= 3 -> x < x0) }*

{ x = x0} x = x – 3 ; if x<0 then x = 1 else if true x = x+1 else x = 10 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

Step 1: sequential composition rule :

Lets first prove ;

{ x = x0} x = x – 3 ; if x<0 then x = 1 else if true x = x+1 else x = 10 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

1. {(x0 < 3 -> 1 = 1) (x0 >= 3 -> 1 < x0) } x=1{(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0)

(uses assignment rule)

1. {{(x0 < 3 -> x+1 = 1) (x0 >= 3 -> x+1 < x0) }} x = x+1 {{(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }}

( uses assignment rule)

1. {(x0 < 3 -> 10 = 1) (x0 >= 3 -> 10 < x0) } x = 10 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

(uses assignment rule)

1. x = x0 x < 0 =>{(x0 < 3 -> 1 = 1) (x0 >= 3 -> 1 < x0) }
2. x = x0 ¬(x < 10) =>{(x0 < 3 -> x+1 = 1) (x0 >= 3 -> x+1 < x0)}
3. x = x0 ¬(¬(x < 10) ) =>{(x0 < 3 -> 10 = 1) (x0 >= 3 -> 10 < x0) }
4. { x = x0 x < 0 } x = 1 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

( strengthening precedent rule, 1,4 )

1. { x = x0 ¬(x < 10) } x = x+1 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0)}

( strengthening precedent rule, 2, 5)

1. { x = x0 ¬(¬(x < 10) ) } x = 10 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

( strengthening precedent rule, 3, 6)

thus

{ x = x0} x = x – 3 ; if x<0 then x = 1 else if true x = x+1 else x = 10 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

Is true

Proof for { x = x0} x = x – 3 ;{(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }

1. {(x0 < 3 -> x-1 = 1) (x0 >= 3 -> x-1< x0) } x = x – 3 ; {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) }
2. x =x0 =>{(x0 < 3 -> x-1 = 1) (x0 >= 3 -> x-1< x0) } (Predicate logic)
3. {x = x0 } x = x-3 {(x0 < 3 -> x = 1) (x0 >= 3 -> x < x0) } (strengthening precedent ,1,2)
4. **Prove the partial correctness of the following program that divides a by b, q is the quotient and a is the remainder (5 marks)**

*{a = \_a & b > 0 & a >= 0 }*

*q = 0;*

*while (a >= b) {*

*a = a - b;*

*q = q + 1;*

*}*

*}\]*

*{ q \* b + a= \_a & a >= 0 & a < b }*

Loop invariant:

q \* b + a = \_a & a >= 0

Proof:

Assignment

{ a = \_a & b > 0 & a > 0}

[ q := 0]

While (a >=b) {

a = a –b ;

q = q + 1;

}

}\]

{ q \* b + a = \_a & a > = 0 & a < b}

Using loop rule with the above invariant gives new proof obligations A, B and C.

A( invariant initially valid):

a = \_a & b > 0 & a > = 0 - >[q := 0] (q \* b + a =\_a & a >= 0)

Apply update

a = \_a & b > 0 & b >= 0 -> 0 \* b + a = \_a & a >= 0

simplify

a = \_a & b > 0 & a >= 0 -> a = \_a & a >= 0

This is trivially valid.

B (invariant preserved):

{q \* b + a = \_a & a >= 0 & (a >=b) = true}

a = a –b;

q = q + 1;

{ q \* b + a = \_a & a >= 0 }

two assignment, then exit

q\* b + a = \_a & a > = 0 & a >= b ->

(q + 1) \* b + (a – b) = \_a & (a – b) >= 0

Rewrite consequents

q \* b + a = \_a & a > 0 & a > = b->

q \* b + a = \_a & a >= b

This is trivially valid.

C (use invariant)

{ q \* b + a = \_a & a >= 0 & (a > =b) = false }

{ q \* b + a = \_a & a >= 0 & a < b }

Hoare triple contains no program (and no update).

Exit rule

q \* b + a = \_ a & a >= 0 & ! a >= b ->

q \* b + a = \_a & a > = 0 & a < b