

# CSE 503

Software Engineering

Winter 2021

## Partial test oracles

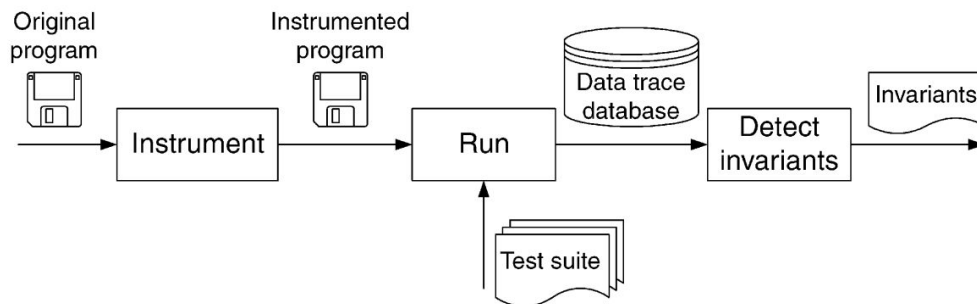
February 12, 2021

## Recap: Preconditions, postconditions, invariants

```
1 double avgAbs(double[] nums) {
2   int n = nums.length;
3   double sum = 0;
4
5   int i = 0;
6   while (i != n) {
7     if(nums[i]>0) {
8       sum = sum + nums[i];
9     } else {
10      sum = sum - nums[i];
11    }
12    i = i + 1;
13  }
14
15  return sum / n;
16 }
```

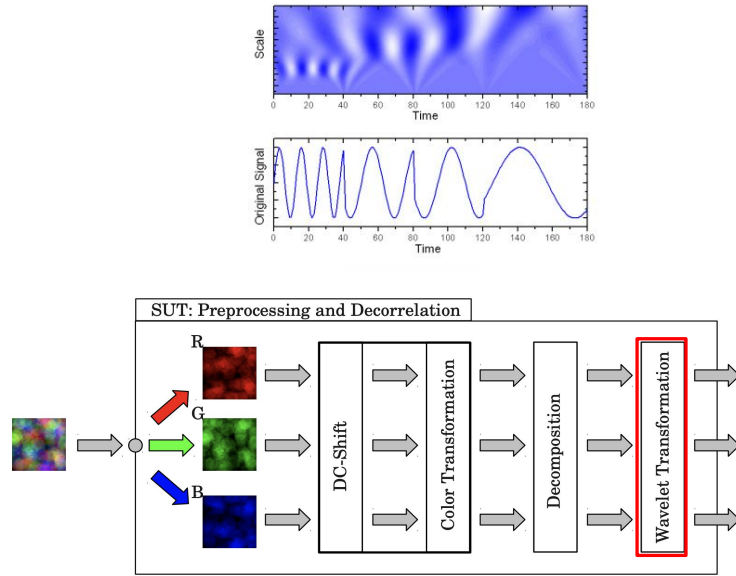
Diagram showing the code with callouts for "Entry point" at line 2 and "Exit point" at line 15.

## Recap: Detection of likely invariants (Daikon)

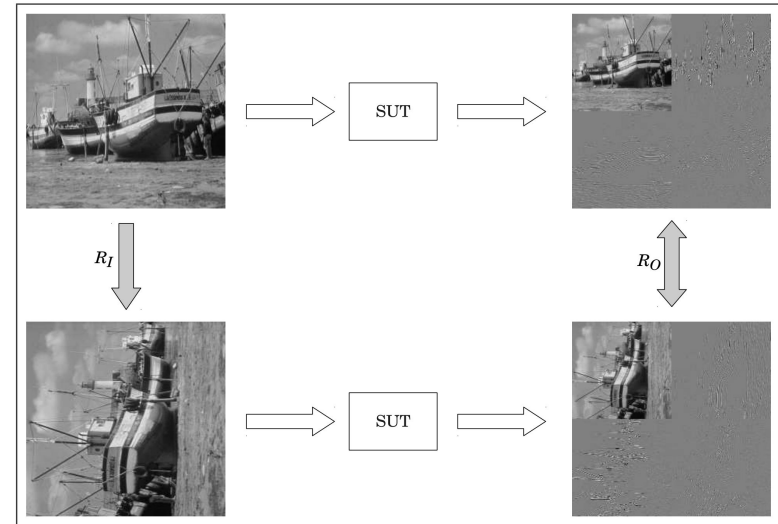


## Metamorphic testing

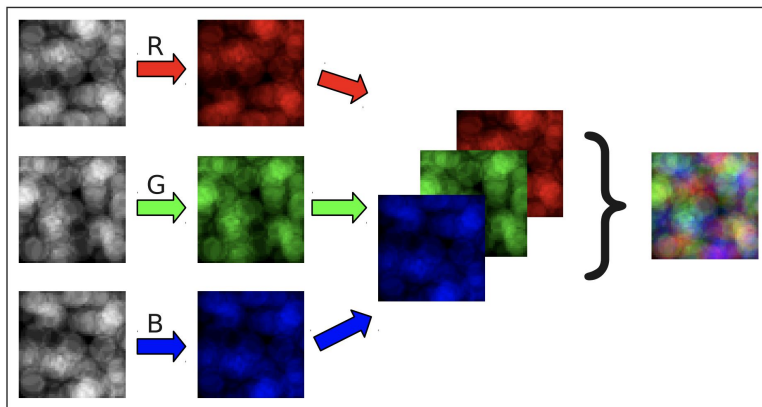
## Metamorphic testing: an example system



## Metamorphic testing: SUT (Wavelet transformation)



## Metamorphic testing: Input generation



## Metamorphic testing: Relations

- R<sub>I</sub>**  $R_I$ : Add an offset to the color values.
- $R_O$ : Only the DC component must be affected.



## Metamorphic testing: Relations

- R1**  $R_I$ : Add an offset to the color values.  
 $R_O$ : Only the DC component must be affected.
- R2**  $R_I$ : Multiply the color values by a coefficient.  
 $R_O$ : Every pixel has to be affected.



## Metamorphic testing: Relations

- R1**  $R_I$ : Add an offset to the color values.  
 $R_O$ : Only the DC component must be affected.
- R2**  $R_I$ : Multiply the color values by a coefficient.  
 $R_O$ : Every pixel has to be affected.
- R3**  $R_I$ : Transpose the pixel array of the input image.  
 $R_O$ : The resulting components have to be transposed.



## Metamorphic testing: Relations

- R1**  $R_I$ : Add an offset to the color values.  
 $R_O$ : Only the DC component must be affected.
- R2**  $R_I$ : Multiply the color values by a coefficient.  
 $R_O$ : Every pixel has to be affected.
- R3**  $R_I$ : Transpose the pixel array of the input image.  
 $R_O$ : The resulting components have to be transposed.
- R4**  $R_I$ : Enlarge the input image with zero-padding.  
 $R_O$ : The resulting components have to be shifted.



## Metamorphic testing: Relations

- R1**  $R_I$ : Add an offset to the color values.  
 $R_O$ : Only the DC component must be affected.
- R2**  $R_I$ : Multiply the color values by a coefficient.  
 $R_O$ : Every pixel has to be affected.
- R3**  $R_I$ : Transpose the pixel array of the input image.  
 $R_O$ : The resulting components have to be transposed.
- R4**  $R_I$ : Enlarge the input image with zero-padding.  
 $R_O$ : The resulting components have to be shifted.
- R5**  $R_I$ : Invert the color values of the input image.  
 $R_O$ : The color values of the resulting components have to be inverted.



## Metamorphic testing: Relations

**R1**  $R_I$ : Add an offset to the color values.

$R_O$ : Only the DC component must be affected.

**R2**  $R_I$ : Multiply the color values by a coefficient.

$R_O$ : Every pixel has to be affected.

**R3**  $R_I$ : Transpose the pixel array of the input image.

$R_O$ : The resulting components have to be transposed.

**R4**  $R_I$ : Enlarge the input image with zero-padding.

$R_O$ : The resulting components have to be shifted.

**R5**  $R_I$ : Invert the color values of the input image.

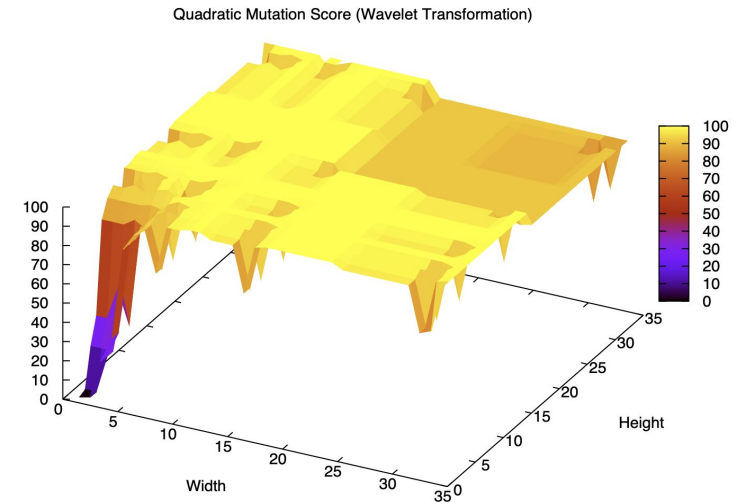
$R_O$ : The color values of the resulting components have to be inverted.

Commutative

Time-invariant

It turns out that composition makes strong oracles.

## Metamorphic testing: Effectiveness



## Your turn



1. Hash out two metamorphic relations (ideally for your course project).
2. Discuss effectiveness and composability.
3. Present your results to the class.

**A primer on solver-aided reasoning**  
(507 covers this topic in detail)

**Z3**

## What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.

```
(echo "Running Z3...")
```

## What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - **Declare variables** and functions.

```
(echo "Running Z3...")  
(declare-const a Int)
```

## What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - **Declare variables** and functions.
  - **Define constraints**.

```
(echo "Running Z3...")  
(declare-const a Int)  
(assert (> a 0))
```

## What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - **Declare variables** and functions.
  - **Define constraints**.
  - **Check satisfiability** and **obtain a model**.
  - ...

```
(echo "Running Z3...")  
(declare-const a Int)  
(assert (> a 0))  
(check-sat)  
(get-model)
```

## What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - **Declare variables** and functions.
  - **Define constraints**.
  - **Check satisfiability** and **obtain a model**.
  - ...

```
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

Which question does this code answer?

## What is Z3?

- An SMT (Satisfiability Modulo Theories) solver.
- Uses a standard language (SMT-LIB).
  - Print to the screen.
  - **Declare variables** and functions.
  - **Define constraints**.
  - **Check satisfiability** and **obtain a model**.
  - ...

```
(echo "Running Z3...")
(declare-const a Int)
(assert (> a 0))
(check-sat)
(get-model)
```

This code is asking the question:  
Does an integer greater than 0 exist?

## A first example

```
1 int simpleMath(int a, int b) {
2   assert(b>0);
3   if(a + b < a) {
4     return 1;
5   }
6   return 0;
7 }
```

Does this method ever return 1?

## A first example

```
1 int simpleMath(int a, int b) {
2   assert(b>0);
3   if(a + b < a) {
4     return 1;
5   }
6   return 0;
7 }
```

```
(declare-const a Int)
(declare-const b Int)
(assert (> b 0))
(assert (< (+ a b) a))
(check-sat)
```

Does this method ever return 1? Let's ask Z3...



## A more complex example

```

1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b < c) return 1;
5   if (a + b > c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }

```

Does this method ever return 3?

## A more complex example

```

1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b < c) return 1;
5   if (a + b > c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }

```

All of the following must be true (why?):

- $!(c == 0)$
- $!(c == 4)$
- $!(a + b < c)$
- $!(a + b > c)$
- $a * b == c$

Does this method ever return 3?

## A more complex example

```

1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b < c) return 1;
5   if (a + b > c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }

```

All of the following must be true (why?):

- $!(c == 0)$
- $!(c == 4)$
- $!(a + b < c)$
- $!(a + b > c)$
- $a * b == c$

```

(declare-const a Int)
(declare-const b Int)
(declare-const c Int)

(assert (not (= c 0)))
(assert (not (= c 4)))
(assert (not (< (+ a b) c)))
(assert (not (> (+ a b) c)))
(assert (= (* a b) c))

(check-sat)

```

## A first example: limited precision

```

1 int simpleMath(int a, int b) {
2   assert(b>0);
3   if(a + b < a) {
4     return 1;
5   }
6   return 0;
7 }

```

Z3 supports Bitvectors of arbitrary size.  
Let's model Java ints (32 bits) and ask the same question...

## A first example: limited precision

```
1 int simpleMath(int a, int b) {
2   assert(b>0);
3   if(a + b < a) {
4     return 1;
5   }
6   return 0;
7 }
```

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(assert (bvsgt b #x00000000))
(assert (bvslt (bvadd a b) a))
(check-sat)
(get-model)
```

Z3 supports Bitvectors of arbitrary size.  
Let's model Java ints (32 bits) and ask the same question...

## A more complex example: limited precision

```
1 int getNumber(int a, int b, int c) {
2   if (c==0) return 0;
3   if (c==4) return 0;
4   if (a + b < c) return 1;
5   if (a + b > c) return 2;
6   if (a * b == c) return 3;
7   return 4;
8 }
```

```
(define-sort JInt () (_ BitVec 32))
(declare-const a JInt)
(declare-const b JInt)
(declare-const c JInt)
(assert (not (= c #x00000000)))
(assert (not (= c #x00000004)))
(assert (not (bvslt (bvadd a b) c)))
(assert (not (bvsgt (bvadd a b) c)))
(assert (= (bvmul a b) c))
(check-sat)
(get-model)
```

## Reasoning about program equivalence

```
1 int add1(int a, int b) {
2   return a + b;
3 }
4
5 int add2(int a, int b) {
6   return a * b;
7 }
```

Are these two methods semantically equivalent?

## Reasoning about program equivalence

```
1 int add1(int a, int b) {
2   return a + b;
3 }
4
5 int add2(int a, int b) {
6   return a * b;
7 }
```

```
(declare-const a Int)
(declare-const b Int)
(declare-const add1 Int)
(declare-const add2 Int)
(assert (= add1 (+ a b)))
(assert (= add2 (* a b)))
(assert (= add1 add2))
(check-sat)
(get-model)
```

Are these two methods semantically equivalent?



## Reasoning about program equivalence

```
1 int add1(int a, int b) {  
2   return a + b;  
3 }  
4  
5 int add2(int a, int b) {  
6   return a * b;  
7 }
```

```
(declare-const a Int)  
(declare-const b Int)  
  
(declare-const add1 Int)  
(declare-const add2 Int)  
  
(assert (= add1 (+ a b)))  
(assert (= add2 (* a b)))  
(assert (= add1 add2))  
  
(check-sat)  
(get-model)
```

Yes, for  $a=0$  and  $b=0$ .  
What have we actually proven here?

## Reasoning about program equivalence

```
1 int add1(int a, int b) {  
2   return a + b;  
3 }  
4  
5 int add2(int a, int b) {  
6   return a * b;  
7 }
```

```
(declare-const a Int)  
(declare-const b Int)  
  
(declare-const add1 Int)  
(declare-const add2 Int)  
  
(assert (= add1 (+ a b)))  
(assert (= add2 (* a b)))  
(assert (not (= add1 add2)))  
  
(check-sat)  
(get-model)
```

Our goal is to prove the absence of counter examples  
(i.e., the defined constraints are unsatisfiable)!

## What's next?

- **Week 1:** Introduction **HW 1**
- **Week 2:** Abstract Interpretation
- **Week 3:** Abstract Interpretation **HW 2**
- **Week 4:** Testing
- **Week 5:** Delta Debugging **In-class exercise**
- **Week 6:** Invariants
- **Week 7:** Program Repair
- **Week 8:** Empirical Software Engineering
- **Week 9:** ML for Software Engineering
- **Week 10:** Wrap up **Project presentation**