

CSE 403

Software Engineering

Program Analysis

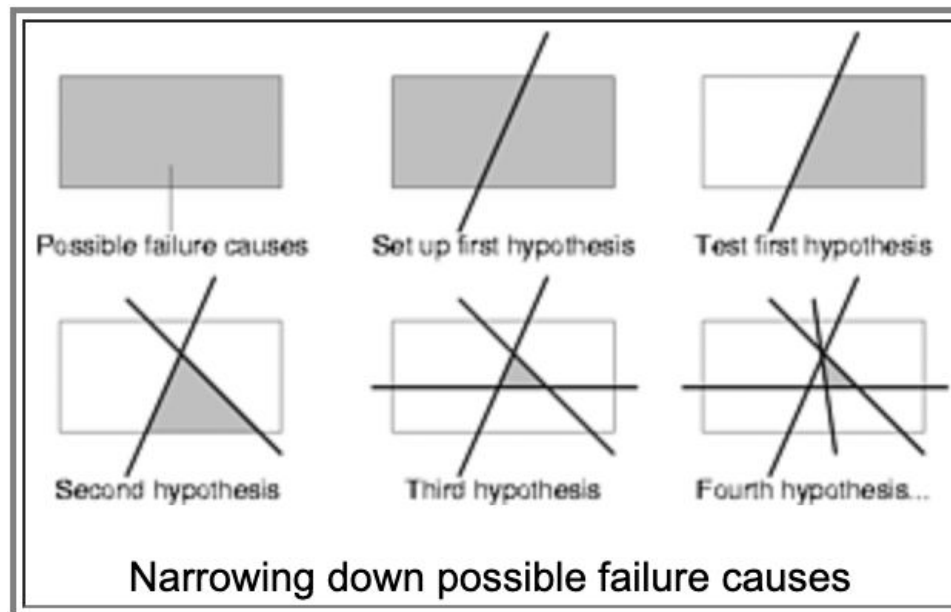
Today

- Delta Debugging: discussion
- Reasoning about programs
- Dynamic vs. static program analysis



Delta debugging: discussion

- Applicability: Is this useful (as a concept and/or automated tool)?
- Optimality: minimal vs. minimum test case.
- Complexity: Best-case vs. worst-case.
- Assumptions: monotonicity and determinism.



Reasoning about programs

Reasoning about programs

Use cases

- Verification/testing: ensure code is correct
- Prove facts to be true, e.g.:
 - x is never null
 - y is always greater than 0
 - input array a is sorted
- Debugging: understand why code is incorrect

Approaches

- Testing (403)
- (Delta) Debugging (403)
- Abstract interpretation (primer in 403, covered in 503)
- Theorem proving (primer in 403, covered in 507)
- ...

Forward vs. backward reasoning

Forward reasoning

- Knowing: **a fact** that is **true before execution**.
- Reasoning: **what must be true after execution**.
- Given a precondition, what postcondition(s) are true?

Backward reasoning

- Knowing: **a fact** that is **true after execution**.
- Reasoning: **what must have been true before execution**.
- Given a postcondition, what precondition(s) must hold?

Forward vs. backward reasoning

Forward reasoning

- More intuitive for most people
- Helps understand what will happen (simulates the code)
- Introduces facts that may be irrelevant to the goal
- Set of current facts may get large
- Takes longer to realize that the task is hopeless

Backward reasoning

- Usually more helpful
- Helps understand what should happen
- Given a specific goal, indicates how to achieve it
- Given an error, gives a test case that exposes it

Pre/Post-conditions and Invariants

Terminology

Pre-condition (to a function)

- A condition that must be true when entering (the function)
- May include expectations about the arguments

Post-condition (to a function)

- A condition that must be true when leaving (the function)

Loop invariant

- A condition that must be true for every loop iteration
- Must be true at the beginning and end of the loop body

Pre-conditions define execution validity. Post-conditions and loop invariants define expected properties of a correct implementation, given a valid execution.

Pre-conditions and post-conditions



```
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 }
```

Entry point

Exit point

What are pre-conditions
and post-conditions of
this method (at the entry
and exit points)?

(Loop) 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 }
```

Does this loop terminate?
What are pre-conditions,
post-conditions,
and loop invariants?

Summary

Pre-condition (to a function)

- A condition that must be true when entering (the function)
- May include expectations about the arguments

Post-condition (to a function)

- A condition that must be true when leaving (the function)

Loop invariant

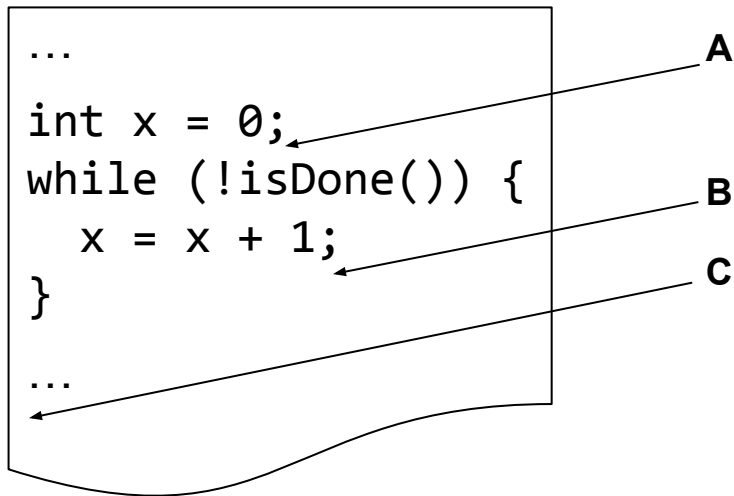
- A condition that must be true for every loop iteration
- Must be true at the beginning and end of the loop body

How are these related to software testing and debugging?

Dynamic vs. static program analysis

Properties of an ideal program analysis

- Soundness
- Completeness
- Termination



Static analyses usually sacrifice completeness (for soundness).

Dynamic vs. static analysis

Dynamic analysis

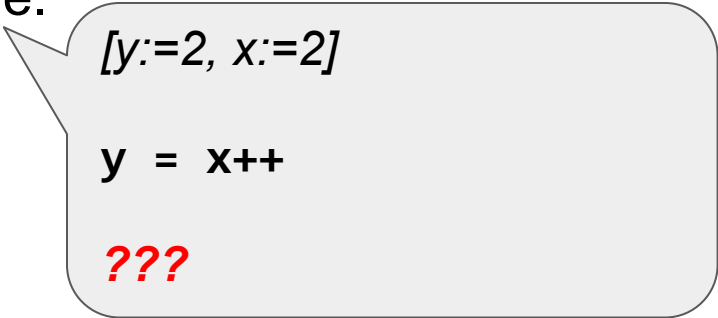
- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.



`[y:=2, x:=2]`

`y = x++`

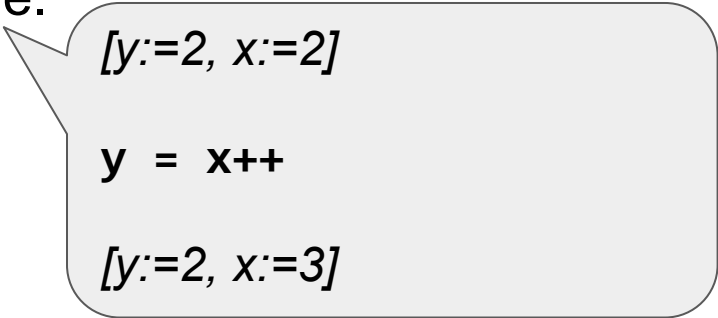
???

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.



$[y:=2, x:=2]$

$y = x++$

$[y:=2, x:=3]$

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

[y:=2, x:=2]

y = x++

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

[y:=even, x:=even]

y = x++

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

[y:=2, x:=2]

y = x++

[y:=2, x:=3]

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

[y:=even, x:=even]

y = x++

???

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

[y:=2, x:=2]

y = x++

[y:=2, x:=3]

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

[y:=even, x:=even]

y = x++

[y:=even, x:=odd]

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

[y:=2, x:=2]

y = x++

[y:=2, x:=3]

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

[y:=prime, x:=prime]

y = x++

???

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

[y:=2, x:=2]

y = x++

[y:=2, x:=3]

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

[y:=prime, x:=prime]

y = x++

*[y:=prime, x:=**anything**]*

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic vs. static analysis

Dynamic analysis

- Reason about the program based on **some** program **executions**.
- Observe **concrete behavior** at run time.
- Improve confidence in correctness.
- **Unsound*** but **precise**.

$[y:=2, x:=2]$

$y = x++$

$[y:=2, x:=3]$

Static analysis

- Reason about the program **without executing** it.
- Build an **abstraction of run-time states**.
- Reason over **abstract domain**.
- **Prove a property** of the program.
- **Sound*** but **imprecise**.

The statement “**f returns a non-negative value**” is weaker (but easier to establish) than the statement “**f returns the absolute value of its argument**”.

* Some static analyses are unsound; dynamic analyses can be sound.

Dynamic analysis: examples

Software testing

```
double avg(double[] nums) {  
    int n = nums.length;  
    double sum = 0;  
  
    int i = 0;  
    while (i < n)  
        sum = sum + nums[i];  
        i = i + 1;  
  
    double avg = sum / n;  
  
    return avg;  
}
```

A test for the avg function:

```
@Test  
public void testAvg() {  
    double nums =  
        new double[]{1.0, 2.0, 3.0};  
    double actual = Math.avg(nums);  
    double expected = 2.0;  
    assertEquals(expected, actual, EPS);  
}
```

Static analysis: examples

```
static OSStatus
SSLVerifySignedServerKeyExchange(...) {
    OSStatus err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature, signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify returned %d\n", (int)err);
        goto fail;
    }
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Apple's "goto fail" bug:
a security vulnerability
for 2 years!

Static analysis: examples

Rule/pattern-based analysis (PMD, Findbugs, Error Prone, etc.)

```
double avg(double[] nums) {  
    int n = nums.length;  
    double sum = 0;  
  
    int i = 0;  
    while (i < n)  
        sum = sum + nums[i];  
        i = i + 1;  
  
    double avg = sum / n;  
  
    return avg;  
}
```

```
double avg(double[] nums) {  
    int n = nums.length;  
    double sum = 0;  
  
    int i = 0;  
    while (i < n) {  
        sum = sum + nums[i];  
        i = i + 1;  
    }  
    double avg = sum / n;  
  
    return avg;  
}
```

Static analysis: examples

Compiler: type checking

```
double avg(double[] nums) {  
    int n = nums.length;  
    double sum = 0;  
  
    int i = 0.0;  
    while (i < n) {  
        sum = sum + nums[i];  
        i = i + 1;  
    }  
    double avg = sum / n;  
  
    return avg;  
}
```

```
double avg(double[] nums) {  
    int n = nums.length;  
    double sum = 0;  
  
    int i = 0;  
    while (i < n) {  
        sum = sum + nums[i];  
        i = i + 1;  
    }  
    double avg = sum / n;  
  
    return avg;  
}
```

Static analysis: examples



Program

```
x = 0;  
y = read_even();  
x = y + 1;  
y = 2 * x;  
x = y - 2;  
y = x / 2;
```

Are all statements necessary?

Static analysis: applications

Compiler checks and optimizations

- Liveness analysis (register reallocation)
- Reachability analysis (dead code elimination)
- Code motion (`while(cond){x = comp(); ...}`)

Dynamic vs. static analysis

Dynamic analysis

- Concrete domain
- Precise but unsound
- Slow if exhaustive

Static analysis

- Abstract domain
- Sound but imprecise
- Slow if precise

Dynamic vs. static analysis

Dynamic analysis

- Concrete domain
- Precise but unsound
- Slow if exhaustive

Static analysis

- Abstract domain
- Sound but imprecise
- Slow if precise

What possible value(s) does `getValue()` return?

Concrete domain

..., -2, -1, 0, 1, 2, ...

```
int getValue(int a) {  
    return (a % 3) * 2;  
}  
  
int x = getValue(7);
```

Abstract domain

even, odd, anything