Software Testing & Quality Assurance
(SE465)
General Information

Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Course Staff

- Instructor
  - Chengnian Sun (cnsun@uwaterloo.ca, DC 2339)

- Teaching assistants
  - Yiwen Dong (y225dong@uwaterloo.ca, Ph.D. student)
  - Victor (Yongqiang) Tian (y258tian@uwaterloo.ca, Ph.D. student)
  - Zhenyang Xu (z473xu@uwaterloo.ca, Ph.D. student)
  - Mengxiao Zhang (m492zhan@uwaterloo.ca, Ph.D. student)
  - Raymond Chang(rkchang@uwaterloo.ca, MMath student)

  - piazza post assignment: https://chengniansun.bitbucket.io/piazzacalc.html

Time & Location

Lectures
- Date: Monday & Wednesday
- Time and Location:
  - Section 1: 02:30-03:50 at E2 1736
  - Section 2: 10:00-11:20 at RCH 211

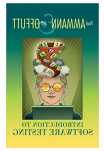
- Office hours: by appointments (do not hesitate to email me)
- Email:
  - Email subject prefixed with "**[SE465]:**"
  - From your uwaterloo email address

Textbook & Lecture Slides

- Required textbook
  - None

- Slides & notes
  - LEARN: https://learn.uwaterloo.ca

- Optional textbook
  - "Introduction to Software Testing"
  - Paul Ammann and Jeff Offutt
  - Cambridge University Press, 2008



Course Website

- Course is managed with LEARN (learn.uwaterloo.ca)
  - Information provided today may be changed.
  - Please check frequently
- Piazza for announcements, discussions and questions
  - Everyone should have been added to Piazza.
  - https://piazza.com/uwaterloo.ca/winter2023/se465

Evaluation (planned as in-person)

- 3 individual assignments: 21% (7% each)
  - Grace days
    - Up to 2 days late in total
    - 2 days for all three, not for each
    - To use it, just submit to the same Dropbox folder. We will check the submission timestamp
- Individual course project: 19%
  - No late submission
- midterm: 20%
  - Feb 28, 6:30 PM - 7:50 PM
  - UW M3 1006
- final: 40%

Evaluation (contingency plan)

- 3 individual assignments: 21% (7% each)

	in-person midterm and in-person final	in-person midterm and online final	online midterm and in-person final	online midterm and online final
assignments	21%	26%	24%	29%
project	19%	24%	21%	26%
midterm	20%	20%	15%	15%
final	40%	30%	40%	30%

- midterm held online → 20%-5%=15%
- final held online → 40%-10%=30%

Why This Course? – A Practical Perspective

- *"We have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."*

  – Bill Gates

- Google is similar, though most tests are written by developers.
- Moreover, you may be asked to write tests during interviews.

Goals of This Course

- How to test as a developer & become a better developer
- How to measure testing coverage
- How to use and write automated testing tools

Software Testing & Quality Assurance
(SE465)
Introduction to Software Testing

Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Bugs are ……

Prevalent

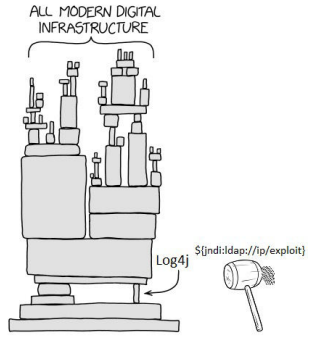
- 20M+ issues on GitHub were closed between 10/01/18 and 09/30/19

Costly

- US$ 59.5 billion dollars annually to US industry throughout 2000s

https://octoverse.github.com/
The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology, 2002

Recent Bugs – log4shell 2021



- Log4j is a logging library for Java, widely used
- allows remote code execution
  - affected Amazon Web Services, iCloud, Minecraft, etc
  - affected many websites in Canada
    - CRA, Employment and Social Development…
- "Experts described Log4Shell as the largest vulnerability ever;[8] LunaSec characterized it as "a desig proportions",[5] Tenable said the exploit was "the single biggest, most critical vulnerability ever",[18 most severe vulnerability ever"[19] and The Washington Post said that descriptions by security professionals "border on the apocalyptic".[8]"

https://www.cve.org/CVERecord?id=CVE-2021-44228
https://en.wikipedia.org/wiki/Log4Shell

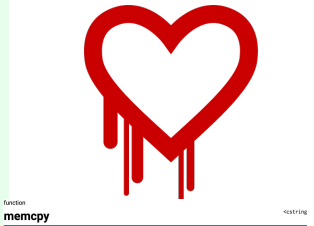
Recent Bugs – Heartbleed 2014

- OpenSSL is a widely-used security protocol library
- No bound check for memcpy(bp, pl, payload);

```
1465 +        if (hbtype == TLS1_HB_REQUEST)
1466 +                {
1467 +                unsigned char *buffer, *bp;
1468 +                int r;
1469 +
1470 +                /* Allocate memory for the response, size is 1 byte
1471 +                 * message type, plus 2 bytes payload length, plus
1472 +                 * payload, plus padding
1473 +                 */
1474 +                buffer = OPENSSL_malloc(1 + 2 + payload + padding);
1475 +                bp = buffer;
1476 +
1477 +                /* Enter response type, length and copy payload */
1478 +                *bp++ = TLS1_HB_RESPONSE;
1479 +                s2n(payload, bp);
1480 +                memcpy(bp, pl, payload);
```

kallus on Apr 10, 2014

Heartbleed explanation: This line copies payload bytes from pl to bp. But payload contains the length that the client says that pl has, it has not been checked that the client doesn't lie about the length of pl. So if the client sent less data than it said it would, whatever lies after pl in memory also gets copied to bp. A few lines below, buffer (which is bp and its header) gets sent back to the client. Thanks to http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html for explaining this.



function

memcpy

<cstring>

```
void * memcpy ( void * destination, const void * source, size_t num );
```

Copy block of memory

Copies the values of **num** bytes from the location pointed to by **source** directly to the memory block pointed to by **destination**.

The underlying type of the objects pointed to by both the **source** and **destination** pointers are irrelevant for this function; The result is a binary copy of the data.

The function does not check for any terminating null character in **source** - it always copies exactly **num** bytes.

To avoid overflows, the size of the arrays pointed to by both the **destination** and **source** parameters, shall be at least **num** bytes, and should not overlap (for overlapping memory blocks, memmove is a safer approach).

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
https://en.wikipedia.org/wiki/Heartbleed#Websites_and_other_online_services

Any time one uses memcpy, strcpy, strncpy, etc. it is a given that there is buffer-over/underflow risk and the cardinal rule should be observed: never trust user input. I'm sure the first place hackers look for holes is uses of memcpy. This one is not buried in a complicated wrapper or anything, it is just right there!

Recent Bugs – Heartbleed 2014

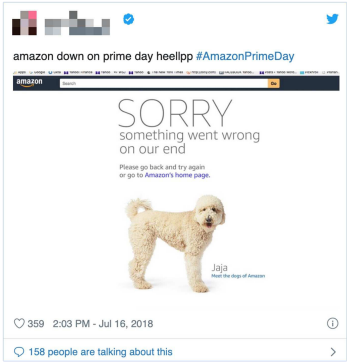
- OpenSSL is a widely-used security protocol library
- No bound check for memcpy(bp, pl, payload);
- Affected many websites, e.g., Yahoo, Stack Overflow, DuckDuckGo, …

Canada Revenue Agency (CRA)
- **The website was hacked due to this software bug, and had to be shut down,**
- **Deadline for filing individual tax returns was postponed.**

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
https://en.wikipedia.org/wiki/Heartbleed#Websites_and_other_online_services

Recent Bugs – Amazon Prime Day Down 2018

- Non-functional requirement
  - Scalability



Recent Bugs – British Airways 2019

- IT outage (6+ since 2017)
- +100 flights to be cancelled
- +200 other flights to be delayed

https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html

Recent Bugs – British Airways 2019

- IT outage (6+ since 2017)
- +100 flights to be cancelled
- +200 other flights to be delayed

Testing is an effective way of avoiding software failures.

https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html

Recent Bugs – British Airways 2019

- IT outage (6+ since 2017)
- +100 flights to be cancelled
- +200 other flights to be delayed

Testing is an effective way of avoiding software failures.

more importantly, cost effective

https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html

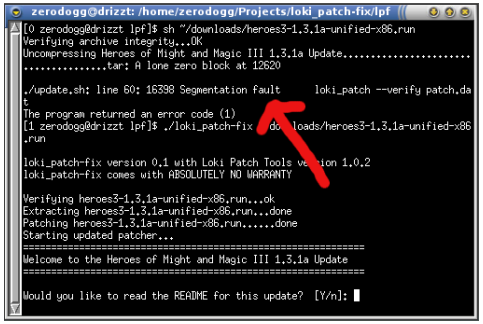
How Software Fails?

- Crash
  - e.g., mobile app aborts, core dump
- Hang
  - never terminates
- Data Corruption
  - corrupted resources, such as ill-formed files, and corrupted file systems
- Incorrect Functionality
  - does not behave as expected
- Performance Degradation
  - a long-running server gets slower
- Security Vulnerabilities
  - buffer overflow, signed integer overflow, null pointer dereference
- .....

Why Does Software Fail? (1)

- Invalid memory access, e.g., segfaults

```
int dereference_null() {
  int* p = nullptr;
  return *p;
}

int dangling_pointer() {
  int* p = new int(1);
  delete p;
  return *p;
}
```



Why Does Software Fail? (2)

- Concurrency bugs, e.g., deadlocks, data races, atomicity violation

Why Does Software Fail? (3)

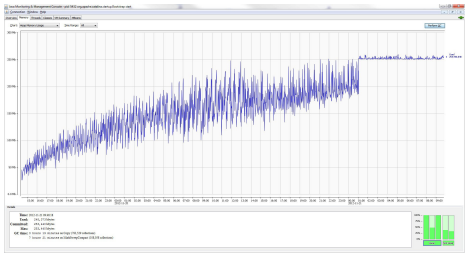
- Wrong implementation
  - Does not respect the requirement

```
int add(int x, int y) {
  return x - y;
}
```


Why Does Software Fail? (4)

- Memory Leaks
  - Exist in managed language, e.g., Java

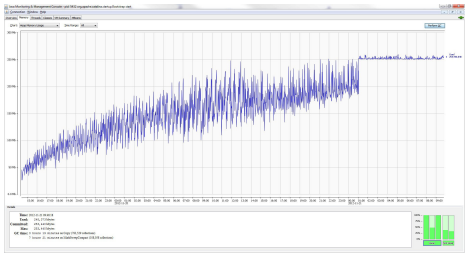
```
class CacheWithLeaks {
  private final HashMap cache = new HashMap();
  public void add(Object key, Object value) {
    cache.put(key, value);
  }
  ......
}
```



Why Does Software Fail? (4)

- Memory Leaks
  - Exist in managed language, e.g., Java

```
class CacheWithLeaks {
  private final HashMap cache = new HashMap();
  public void add(Object key, Object value) {
    cache.put(key, value);
  }
  ......
}


class CacheWoLeaks {
  private final WeakHashMap cache = new WeakHashMap();
  public void add(Object key, Object value) {
    cache.put(key, value);
  }
  ......
}
```



My Favorite Bugs – Both Greatly Simplified

```
// Java
void check(int i) {
  assert Math.abs(i) >= 0; // May fail.
}
```

```
// C++
int get_first() {
  std::vector<int> v;
  v.push_back(0);
  int* p = &v[0];
  for (int i = 1; i < 1000; ++i) {
    v.push_back(i);
  }
  assert(*p == 0); // May fail.
  return *p;
}
```


How to Avoid Software Failures?

- Test the software (in-house, externally)
  - Unit tests
  - Integration tests
  - Outsourcing

How to Avoid Software Failures?

- Test the software (in-house, externally)
- Code review
  - Improve code quality, with suggested better API, algorithms, design
  - May catch bugs

How to Avoid Software Failures?

- Test the software (in-house, externally)
- Code review
- Better design (“write better code!”)
  - Clean architecture
  - Low coupling among components

How to Avoid Software Failures?

- Test the software (in-house, externally)
- Code review
- Better design (“write better code!”)
- Include fewer features
  - More code you write, more bugs you may introduce
  - Program debloating → shrink attack surface

How to Avoid Software Failures?

- Test the software (in-house, externally)
- Code review
- Better design (“write better code!”)
- Include fewer features
- Better programming languages & IDEs
  - Statically typing languages, e.g., Java, Kotlin, Rust
    - fewer typing errors at runtime
    - Better documentation of API signatures
  - Dynamically typing languages → good luck
  - IntelliJ → code completion, suggestions


```
int risky_programming(char *input){
  char str[1000+1];  // One more for the null character.
  strcpy(str, input);  // Copy input.
  ...
}
```

ures?

- Test the software (in-house, externally)
- Code review
- Better design (“write better code!”)
- Include fewer features
- Better programming languages & IDEs
- Defensive programming (never trust user input)

```
int risky_programming(char *input){
  char str[1000+1];  // One more for the null character.
  strcpy(str, input);  // Copy input.
  ...
}
```

ures?

- Test the software (in-hous
- Code review
- Better design (“write bette
- Include fewer features
- Better programming languages & IDEs
- Defensive programming (never trust user input)

```
int defensive_programming(char *input){
  char str[1000+1];  // One more for the null character.
  // Copy input without exceeding the length of the destination.
  strncpy(str, input, sizeof(str));
  str[sizeof(str) - 1] = '\0';
  ...
}
```



```
int risky_programming(char *input){
  char str[1000+1];  // One more for the null character.
  strcpy(str, input);  // Copy input.
  ...
}
```

ures?

- Test the software (in-hous
- Code review
- Better design (“write bette
- Include fewer features
- Better programming languages & IDEs
- Defensive programming
  - Offensive programming

```
int defensive_programming(char *input){
  char str[1000+1];  // One more for the null character.
  // Copy input without exceeding the length of the destination.
  strncpy(str, input, sizeof(str));
  str[sizeof(str) - 1] = '\0';
  ...
}
```


```
int offensive_programming(char *input){
  if (the length of input is > 1000) {
    printf("The input is too long.");
    abort();
  }
  char str[1000+1];  // One more for the null character.
  strcpy(str, input);  // Copy input.
  ...
}
```


How to Avoid Software Failures?

- Test the software (in-house, externally)
- Code review
- Better design (“write better code!”)
- Include fewer features
- Better programming languages & IDEs
- Defensive programming
- Code reuse. Do not reinvent wheels.
  - Well tested

Mitigation: Failure is Inevitable

- Software (in reality) never completely works!

- Aim: Produce software that is good enough!

Coping with An Imperfect World!

- Disclaim liability
  - Open-source software does
  - Commercial software also does

"*25. LIMITATION ON AND EXCLUSION OF DAMAGES. You can recover from Microsoft and its suppliers only direct damages up to the amount you paid for the software. You cannot recover any other damages, including consequential, lost profits, special, indirect or incidental damages.*"

-- **Vista license**

Coping with An Imperfect World (2)

- Release patches
- Backup/replicate user data
- Failure recovery
  - Automatic runtime recovery

Ways of Testing Software

- Compile it
  - Type errors
  - Compiler warnings

Ways of Testing Software

- Compile it
- Run it on one input

Ways of Testing Software

- Compile it
- Run it on one input
- Run it on many inputs
  - Fuzz testing---automatically generate many inputs, and check for failures
  - Random Android GUI testing, e.g., Monkey

Ways of Testing Software

- Compile it
- Run it on one input
- Run it on many inputs
- Run it on a representative set of inputs

Ways of Testing Software

- Compile it
- Run it on one input
- Run it on many inputs
- Run it on a representative set of inputs
- Run it on all inputs (static analysis)

Other Testing Concerns

- Integration testing – how different components work together
- Nonfunctional properties
  - Performance
  - Scalability
  - Memory usage

Key Concepts: Coverage

- Idea: find a reduced space and cover it with tests
- Possible spaces: graphs, logic, input space, syntax

In This Course

- Learn about
  - Graph Coverage
  - Logic Coverage
  - Syntax-Based Coverage
  - Input-Space Based Coverage
  - Testing in Practice
  - State-of-the-art Techniques
  - Use & Build Tools
  - Concurrency
  - ……
- View this course as a window to the large world …

Tools

- Junit
- Clang/LLVM
- Valgrind
- Sanitizers, asan, msan, ubsan
- Error-prone
- ……

Openings

- Our research lab has multiple openings
  - CS499, URA, USRA, URF
  - Master, PhD
- Research areas & topics (incomplete):
  - Program reduction for debugging, e.g., delta debugging, Perses
  - Compiler testing for C, C++, Rust, Go compilers, JVM
  - Android app testing, e.g., energy testing, robustness testing
  - Android app performance analysis
  - Build system analysis, build breakage repair
  - Fast mutation testing
  - ……

Fault, Error and Failure

Chengnian Sun
cnsun@uwaterloo.ca

International Standard: ISO/IEC/IEEE 15026, First edition, 2019-03
Systems and software engineering – Part 1: Concepts and vocabulary

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Terminology – Failure

- Formal definition:
  - termination of the ability of a system to perform a required function
    - crash
  - or its inability to perform within previously specified limits
    - hang
  - an externally visible deviation from the system's specification

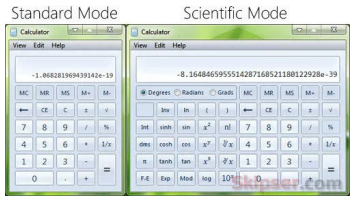
3.1560
failure
1. termination of the ability of a system to perform a required function or its inability to perform within previously specified limits; an externally visible deviation from the system's specification *[ISO/IEC 15026-1:2013 Systems and software engineering — Systems and software assurance — Part 1: Concepts and vocabulary,* 3.4.8*]* ***2.*** violation of a contract *[ISO/IEC 10746-2:2009 Information technology — Open Distributed Processing — Reference Model: Foundations,* 13.6.1*]*

Terminology – Failure

3.1560
failure
1. termination of the ability of a system to perform a required function or its inability to perform within previously specified limits; an externally visible deviation from the system's specification *[ISO/IEC 15026-1:2013 Systems and software engineering — Systems and software assurance — Part 1: Concepts and vocabulary,* 3.4.8*]* ***2.*** violation of a contract *[ISO/IEC 10746-2:2009 Information technology — Open Distributed Processing — Reference Model: Foundations,* 13.6.1*]*

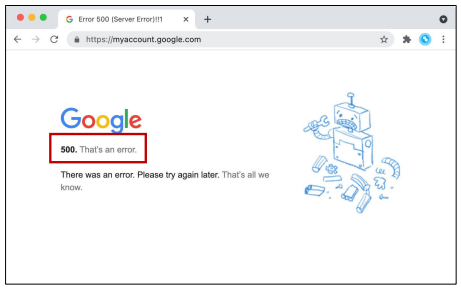
- An unacceptable behavior exhibited by a system (observed externally)
- $reliability \rightarrow \frac{\#\text{failures}}{time}$
- An important design objective is to achieve
  - a very low failure rate
  - and hence high reliability

Terminology – Failure: Examples

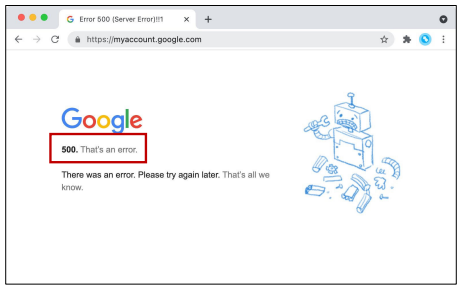


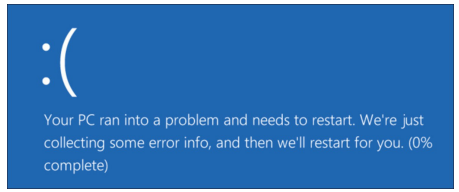
The Calculator on Windows 7,
`sqrt(4)-2=-1.068281969439142e-19`
`sqrt(4)-2=-8.164846595551428716852118012292 8e-39`

Terminology – Failure : Examples



Terminology – Failure : Examples





Terminology – Defect/Bug/Fault

- Formal definition:
  - incorrect step, process, or data definition in a computer program
  - defect in a hardware device or component
  - defect in a system or a representation of a system that if executed/activated could potentially result in an error
- Can be in the requirements, the design and the code
  - examples?

Terminology – Defect/Bug/Fault

- Formal definition:
  - incorrect step, process, or data definition in a computer program
  - defect in a hardware device or component
  - defect in a system or a representation of a system that if executed/activated could potentially result in an error
- Can be in the requirements, the design and the code
  - examples?
    - A requirement provided by the end-user that conflicts with another requirement or constraint. Example: The user wants to email all customers but does not want the system to collect customer emails.
    - The user passes a wrong business fact to the business analyst. Example: A wrong formula to calculate profit.
    - A bug in requirement could occur also when the user and the business analyst miss an entire business function. Example, when the user does not specify security requirement and the business analyst does not ask about this requirement. When such a system is completed, it may not be usable because it is not secure enough.

Terminology – Defect/Bug/Fault

- Formal definition:
  - incorrect step, process, or data definition in a computer program
  - defect in a hardware device or component
  - defect in a system or a representation of a system that if executed/activated could potentially result in an error
- Can be in the requirements, the design and the code
- A fault, if encountered, can cause a failure.
- Needs certain inputs to trigger a fault.
  - Why?

Terminology – Error

- Incorrect **internal** state that is the manifestation of some fault
- Not necessarily lead to failures

- Formal definition:
  - difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition
  - erroneous state of the system


```
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```



```
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Fault: `int i = 1;`

```java
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Fault: `int i = 1;`
Fix : `int i = 0;`


```java
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Fault: `int i = 1;`
Fix : `int i = 0;`

Program State: PC, i, count

PC has to be specific: “before xxx” or “after xxx”


```java
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Fault: `int i = 1;`
Fix : `int i = 0;`

Program State: PC, i, count

Input: x=null
fault , error, failure


```java
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Fault: `int i = 1;`
Fix : `int i = 0;`

Program State: PC, i, count

Input: x=null
fault not executed, no error, no failure

```
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Program State: PC, i, count

Input: x=null
fault not executed, no error, no failure

Input: x=[2, 7, 0]
fault , error, failure

Fault: `int i = 1;`
Fix : `int i = 0;`


```
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Program State: PC, i, count

Input: x=null
fault not executed, no error, no failure

Input: x=[2, 7, 0]
fault executed, error, no failure

Fault: `int i = 1;`
Fix : `int i = 0;`


```
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Program State: PC, i, count

Input: x=null
fault not executed, no error, no failure

Input: x=[2, 7, 0]
fault executed, error, no failure

Error State:

PC=after first iteration
i = 1
count = 0

Expected State:

PC=after first iteration
i = 0
count = 0

Fault: `int i = 1;`
Fix : `int i = 0;`


```
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Program State: PC, i, count

Input: x=null
fault not executed, no error, no failure

Input: x=[2, 7, 0]
fault executed, error, no failure

Input: x=[0, 7, 2]
fault , error, failure

Fault: `int i = 1;`
Fix : `int i = 0;`

```java
/**
 * Count zeros in the array.
 * Throws an exception if 'x' is null.
 */
public int countZeros(int[] x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) {
      count++;
    }
  }
  return count;
}
```

Program State: PC, i, count

Input: x=null
fault not executed, no error, no failure

Input: x=[2, 7, 0]
fault executed, error, no failure

Input: x=[0, 7, 2]
fault executed, error, failure

Error State:

PC=after first iteration
i = 1
count = 0

Expected State:

PC=after first iteration
i = 0
count = 1

Fault: `int i = 1;`
Fix : `int i = 0;`

Exercise

Expected Behavior:

- Returns the index of the last element in 'x' that equals 'y'.
- Returns -1 if no such element exists
- Throws an exception if 'x' is null

```java
public int findLast(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:

a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert findLast(new int[]{2, 3, 5}, 2) == 0`

Solution

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```


Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it

Solutions:

```
a) for (int i = x.length - 1; i >= 0; --i)
```

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault

Solutions:

```
a) for (int i = x.length - 1; i >= 0; --i)
```

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault

Solutions:
a) `for (int i = x.length - 1; i >= 0; --i)`
b) `x=null, y=_`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state

Solutions:
a) `for (int i = x.length - 1; i >= 0; --i)`
b) `x=null, y=_`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state

Solutions:
a) `for (int i = x.length - 1; i >= 0; --i)`
b) `x=null, y=_`
c) `x=[ ]` or `y` appears in `x` at a non-zero index
- `x=[ ], y=_`
  `i>0` and `i>=0` are both false
- `x=[2, 3, 5], y=3`
  `i>0` and `i>=0` are both true

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
- **`assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`**

Solutions:
a) `for (int i = x.length - 1; i >= 0; --i)`
b) `x=null, y=_`
c) `x=[ ]` or `y` appears in `x` at a non-zero index
- `x=[ ], y=_`
  `i>0` and `i>=0` are both false
- `x=[2, 3, 5], y=3`
  `i>0` and `i>=0` are both true

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Solutions:
a) `for (int i = x.length - 1; i >= 0; --i)`
b) `x=null, y=_`
c) `x=[ ]` or `y` appears in `x` at a non-zero index
   - `x=[ ], y=_`
     `i>0` and `i>=0` are both false
   - `x=[2, 3, 5], y=3`
     `i>0` and `i>=0` are both true
d) `x.length > 0 and y is not in x`
   - no failure as the result is always -1
   - but does not execute `if (x[0] == y)`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

Questions:
a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=1
- PC=--i

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=1
- PC=--i

State 6:
- x=[2,3,5]
- y=2
- i=1
- PC=i > 0

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=1
- PC=--i

State 6:
- x=[2,3,5]
- y=2
- i=1
- PC=i > 0

State 7:
- x=[2,3,5]
- y=2
- i=1
- PC=x[i]==y

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=2
- PC=--i

State 6:
- x=[2,3,5]
- y=2
- i=1
- PC=i > 0

State 7:
- x=[2,3,5]
- y=2
- i=1
- PC=x[i]==y

State 8:
- x=[2,3,5]
- y=2
- i=0
- PC=--i

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=1
- PC=--i

State 6:
- x=[2,3,5]
- y=2
- i=1
- PC=i > 0

State 7:
- x=[2,3,5]
- y=2
- i=1
- PC=x[i]==y

State 8:
- x=[2,3,5]
- y=2
- i=0
- PC=--i

State 9:
- x=[2,3,5]
- y=2
- i=0
- PC=i > 0

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case. Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=1
- PC=--i

State 6:
- x=[2,3,5]
- y=2
- i=1
- PC=i > 0

State 7:
- x=[2,3,5]
- y=2
- i=1
- PC=x[i]==y

State 8:
- x=[2,3,5]
- y=2
- i=0
- PC=--i

State 9:
- x=[2,3,5]
- y=2
- i=0
- PC=i > 0

State 10:
- x=[2,3,5]
- y=2
- i=NA
- PC=return -1

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case.
Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:
- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

State 0:
- x=[2,3,5]
- y=2
- i=NA
- PC=method entry

State 1:
- x=[2,3,5]
- y=2
- i=NA
- PC=loop entry

State 2:
- x=[2,3,5]
- y=2
- i=2
- PC=loop init

State 3:
- x=[2,3,5]
- y=2
- i=2
- PC=i > 0

State 4:
- x=[2,3,5]
- y=2
- i=2
- PC=x[i]==y

State 5:
- x=[2,3,5]
- y=2
- i=1
- PC=--i

State 6:
- x=[2,3,5]
- y=2
- i=1
- PC=i > 0

State 7:
- x=[2,3,5]
- y=2
- i=1
- PC=x[i]==y

State 8:
- x=[2,3,5]
- y=2
- i=0
- PC=--i

State 9:
- x=[2,3,5]
- y=2
- i=0
- PC=i > 0

State 10:
- x=[2,3,5]
- y=2
- i=NA
- PC=return -1

Questions:

a) identify the fault, and fix it

b) identify a test case that does not execute the fault

c) identify a test case that executes the fault, but does not result in an error state

d) identify a test case that results in an error, but not a failure

e) identify the first error state for the following test case.
Be sure to describe the complete program state.
- `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected State 10:
- x=[2,3,5]
- y=2
- i=0
- PC=x[i]==y

Another Exercise

The following function removes all occurrences of the second parameter `match` from the first parameter `list`. For example, removeAllMatches(["a", "b"], "a") returns ["b"].

- The parameter `list` will never be null, but its elements can be null.
- The parameter `match` will never be null.
- The passed-in list is modified, and returned.

```
1: public static ArrayList<String> removeAllMatches(
2:     ArrayList<String> list, String match) {
3:   for (int i = 0; i < list.size(); i++) {
4:     if (match.equals(list.get(i))) {
5:       list.remove(i);
6:     }
7:   }
8:   return list;
9: }
```


Another Exercise

The following function removes all occurrences of the second parameter `match` from the first parameter `list`. For example, removeAllMatches(["a", "b"], "a") returns ["b"].

- The parameter `list` will never be null, but its elements can be null.
- The parameter `match` will never be null.
- The passed-in list is modified, and returned.

```
1: public static ArrayList<String> removeAllMatches(
2:     ArrayList<String> list, String match) {
3:   for (int i = 0; i < list.size(); i++) {
4:     if (match.equals(list.get(i))) {
5:       list.remove(i);
6:     }
7:   }
8:   return list;
9: }
```

Q1: find the fault, and try to fix it.

The following function removes all occurrences of the sec
parameter `list`. For example, removeAllMatches(["a", "b"]

- The parameter `list` will never be null, but its eleme
- The parameter `match` will never be null.
- The passed-in list is modified, and returned.

```
1: public static ArrayList<String> removeAllMatches(
2:     ArrayList<String> list, String match) {
3:   for (int i = 0; i < list.size(); i++) {
4:     if (match.equals(list.get(i))) {
5:       list.remove(i);
6:     }
7:   }
8:   return list;
9: }
```

```
1: public static ArrayList<String> removeAllMatches(
2:     ArrayList<String> list, String match) {
3:   for (int i = 0; i < list.size(); i++) {
4:     if (match.equals(list.get(i))) {
5:       list.remove(i);
6:       i--;
7:     }
8:   }
9:   return list;
10:}
```

Q1: find the fault, and try to fix it.

The following function removes all occurrences of the sec
parameter `list`. For example, removeAllMatches(["a", "b"]

- The parameter `list` will never be null, but its eleme
- The parameter `match` will never be null.
- The passed-in list is modified, and returned.

```
1: public static ArrayList<String> removeAllMatches(
2:     ArrayList<String> list, String match) {
3:   for (int i = 0; i < list.size(); i++) {
4:     if (match.equals(list.get(i))) {
5:       list.remove(i);
6:     }
7:   }
8:   return list;
9: }
```

```
1: public static ArrayList<String> removeAllMatches(
2:     ArrayList<String> list, String match) {
3:   for (int i = 0; i < list.size(); i++) {
4:     if (match.equals(list.get(i))) {
5:       list.remove(i);
6:       i--;
7:     }
8:   }
9:   return list;
10:}
```

Q1: find the fault, and try to fix it.
Q2: identify an input which does not trigger the fault.
Q3: identify an input which triggers the fault but does not result in an error state.
Q4: identify an input which triggers the fault, results in an error, but does not trigger a failure.
Q5: identify an input which triggers the fault, results in an error, and triggers failure.

RIP Model

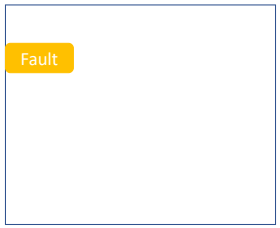
- Three conditions must be present for a failure to occur

- **Reachability**
  - the lines of the fault should be executed during runtime
- **Infection**
  - the execution of the fault induces an error state
- **Propagation**
  - the error state propagates to cause incorrect observable program output

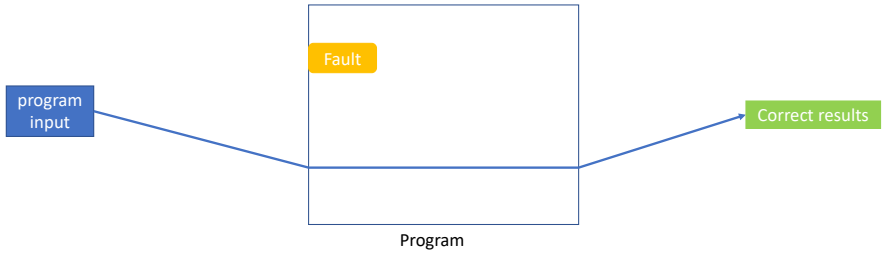
Visualization of Fault, Error and Failure

Program

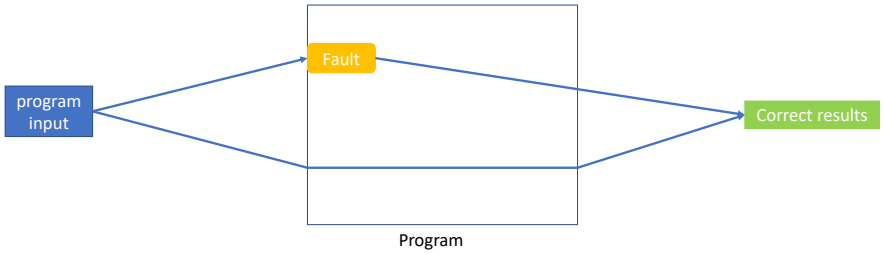
Visualization of Fault, Error and Failure



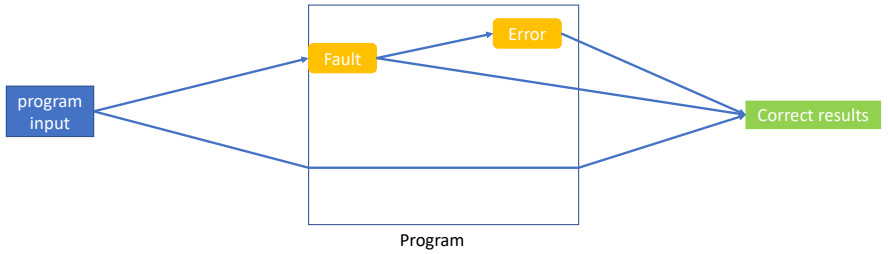
Visualization of Fault, Error and Failure



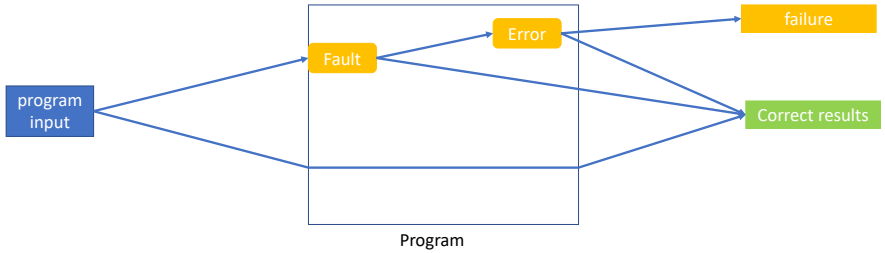
Visualization of Fault, Error and Failure



Visualization of Fault, Error and Failure



Visualization of Fault, Error and Failure



Addressing Faults at Different Stages

- Fault Avoidance
  - better design, better PL, …
  - for example, buffer overflow is impossible in Java
- Fault Detection
  - testing – show the presence of bugs, but not the absence
  - verification – prove the absence of bugs, e.g., CompCert (https://compcert.org/)
- Fault Tolerance
  - redundancy/replication, isolation

Testing vs. Debugging

- **Testing**
  - evaluating software by observing its execution
- **Debugging**
  - the process of finding a fault given a failure

Testing vs. Debugging

- Testing is difficult
  - Often, only certain inputs trigger the fault into creating a failure

```
if (x - 100 <= 0) {
  if (y - 100 <= 0) {
    if (x + y - 200 == 0) {
      crash(1);
    }
  }
}
```

Only x=100 & y=100 triggers the crash
Probability to trigger the crash: $\frac{1}{2^{64}}$

- Debugging is difficult
  - Given a failure, it is often difficult to know the fault

Control Flow Graph

Chengnian Sun
cnsun@

*Slides adapted from Prof. Patrick Lam's, Prof. Lin Tan's and Prof. Arie Gurfinkel's.

Control Flow Graph (CFG)

- Fundamental graph for representing source code
- statically extracted from the source code
- Nodes: zero or more statements
- Edges: an edge (n1, n2) indicates that n1 **may** be followed by n2 in an execution
- An execution trace of statements is a path on the CFG.

- Heavily used in software testing to measure test suites
  - statement coverage
  - branch coverage
  - path coverage
  - .....

Steps in Compilation

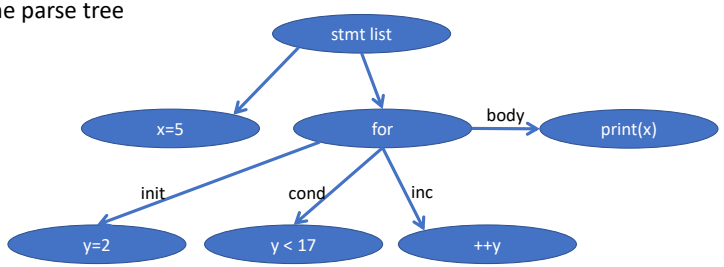
```
x = 5;
for (y = 2; y < 17; ++y)
  print(x);
```

- Lexing
  - input: stream of characters
  - output: stream of tokens
- Parsing
  - input: stream of tokens
  - output: concrete syntax tree (parse tree), determined by the grammar

Steps in Compilation

```
x = 5;
for (y = 2; y < 17; ++y)
  print(x);
```

- Construction of Abstract Syntax Tree (AST)
  - cleans up the parse tree



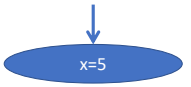
Steps in Compilation

- Construction of Abstract Syntax Tree (AST)
  - cleans up the parse tree
- Conversion to Control Flow Graph (CFG)
  - input: AST
  - lowering code, e.g., for → if-goto, desugar,
  - output: CFG
- Optimizations
  - input: CFG
  - output: CFG
- Convert to bytecode/machine code

```
x = 5;
for (y = 2; y < 17; ++y)
  print(x);
```

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print (x)
      ++y
      goto L0
L1: nop
```

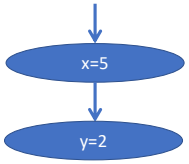

CFG Example




```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```

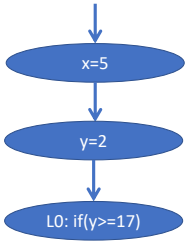

CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



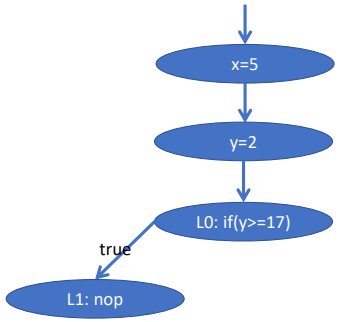
CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



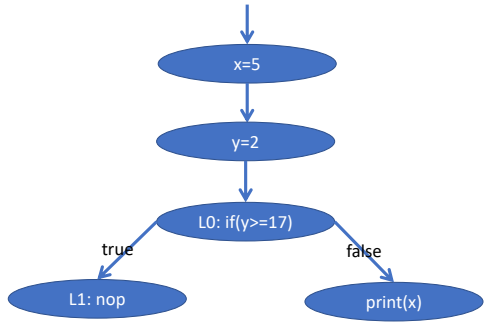
CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



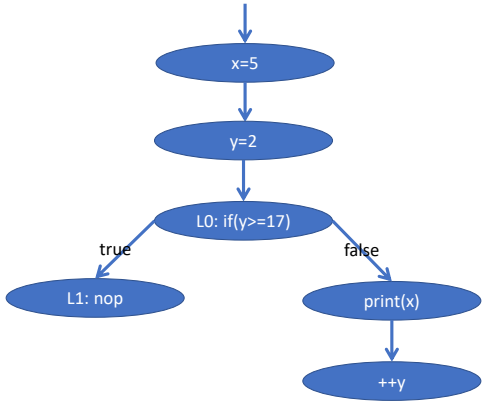
CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



CFG Example

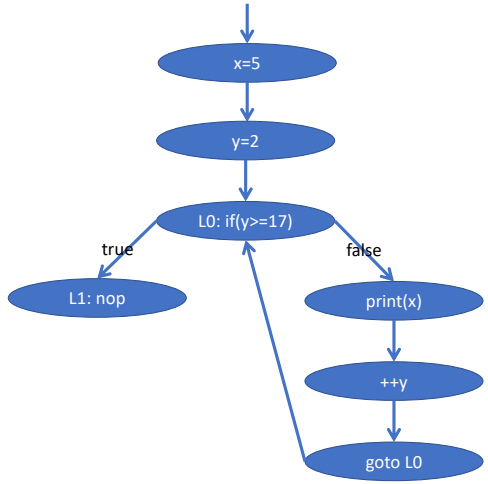
```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```

Edge labels are important.

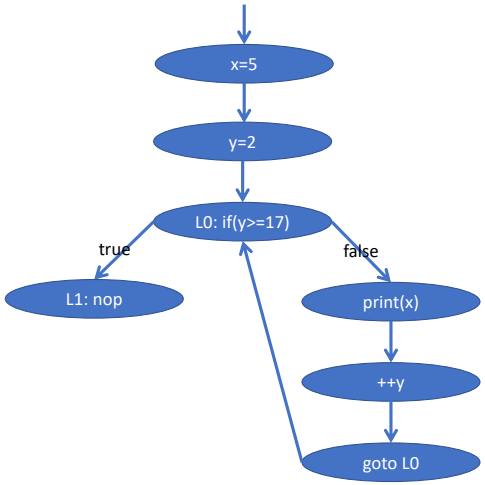


Basic Block

- We can simplify a CFG by grouping together statements which always execute together (in sequential programs)
- A basic block is a sequence of statements [s_1, s_2, …, s_{n-1}, s_n] that has
  - only ***one entry***: s_2, …, s_{n-1}, s_n can not be destinations of any jumps
  - only ***one exit***: only the last statement s_n can transfer control to blocks
- A basic block may have multiple successors

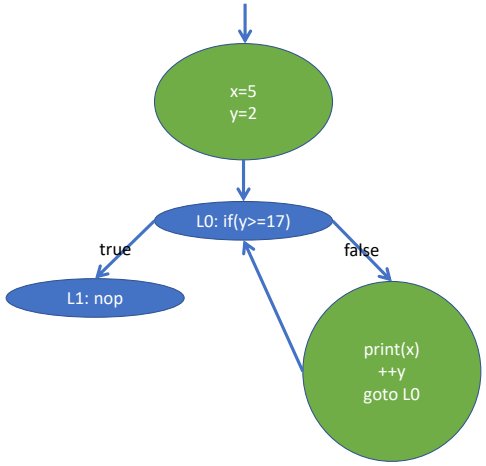
CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



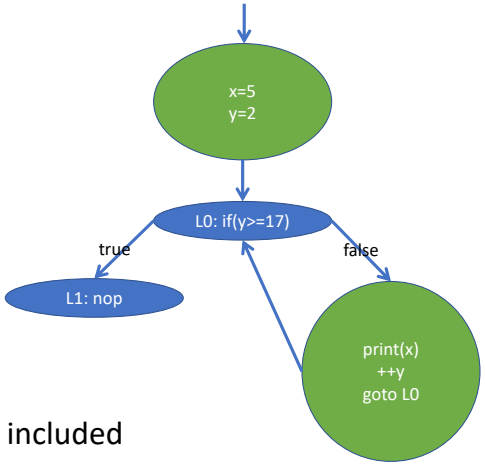
CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```



CFG Example

```
      x=5
      y=2
L0: if(y >= 17) goto L1
      print(x)
      ++y
      goto L0
L1: nop
```

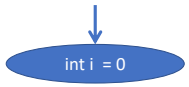


Question?

why is `if(y>=17) goto L1` not included in the previous basic block?

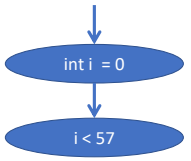
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



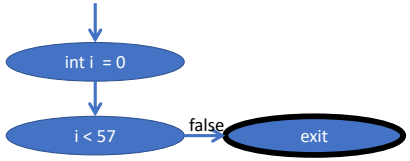
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



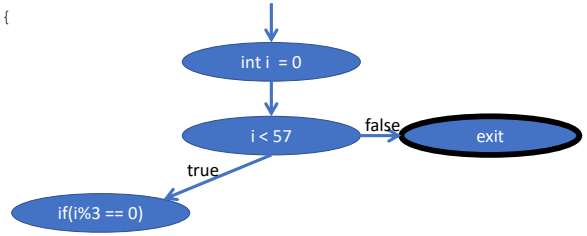
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



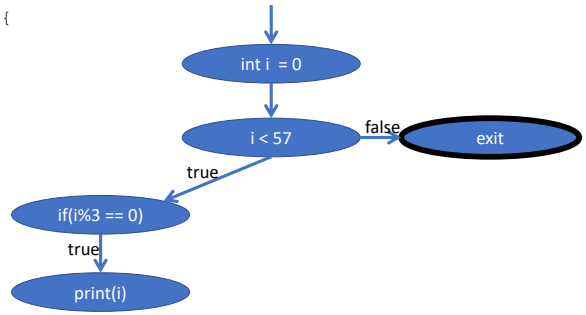
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



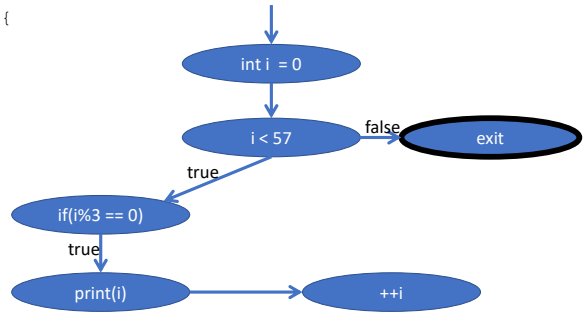
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



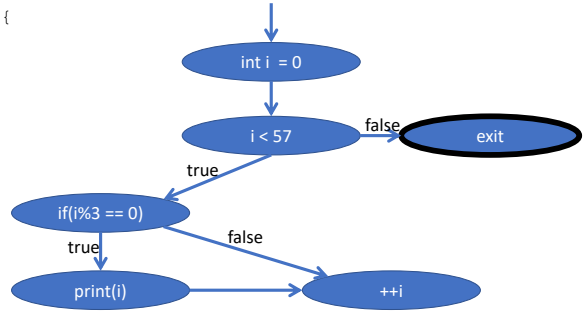
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



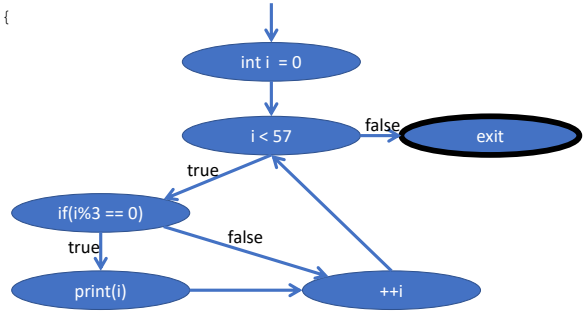
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



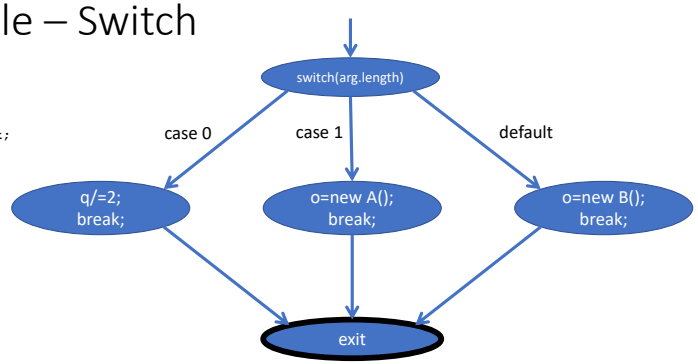
CFG Example – For Loop

```
for (int i = 0; i < 57; ++i) {
  if (i % 3 == 0) {
    print(i);
  }
}
```



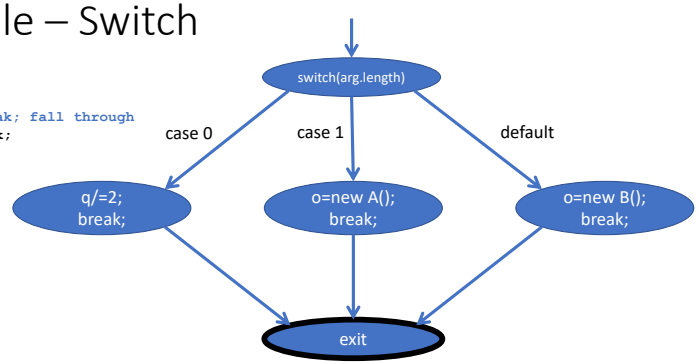
CFG Example – Switch

```
switch(arg.length()) {
  case 0: q/=2; break;
  case 1: o = new A(); break;
  default: o = new B(); break;
}
```



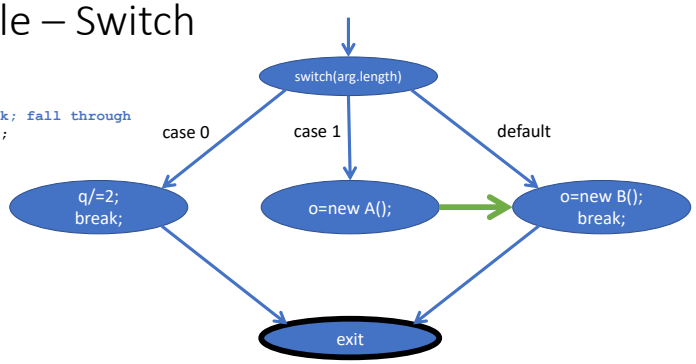
CFG Example – Switch

```
switch(arg.length()) {
  case 0: q/=2; break;
  case 1: o = new A(); //break; fall through
  default: o = new B(); break;
}
```



CFG Example – Switch

```
switch(arg.length()) {
  case 0: q/=2; break;
  case 1: o = new A(); //break; fall through
  default: o = new B(); break;
}
```



CFG Example – Exercise

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```


CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {

}
```


CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {
  ++count;
}
```


CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {
  goto L;
}

L: ++count;
```

CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {
  goto L;
}
if (x[i] > 0) {

}


L: ++count;
```


CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {
  goto L;
}
if (x[i] > 0) {
  goto L;
}


L: ++count;
```


CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {
  goto L;
}
if (x[i] > 0) {
  goto L;
}


L: ++count;
EXIT: ...
```

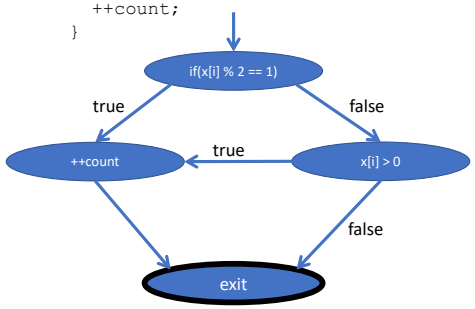

CFG Example – Short Circuit

```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```

```
if (x[i] % 2 == 1) {
  goto L;
}
if (x[i] > 0) {
  goto L;
} else {
  goto EXIT;
}

L: ++count;
EXIT: ...
```

CFG Example – Short Circuit

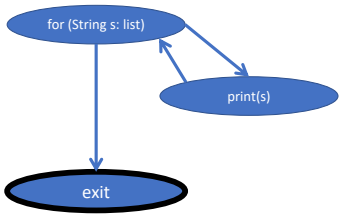
```
if (x[i] % 2 == 1 || x[i] > 0) {
  ++count;
}
```




```
if (x[i] % 2 == 1) {
  goto L;
}
if (x[i] > 0) {
  goto L;
} else {
  goto EXIT;
}

L: ++count;
EXIT: ...
```


CFG Example – Enhanced Loop

```
ArrayList<String> list = ...
for (String string : list) {
  print(string);
}
```

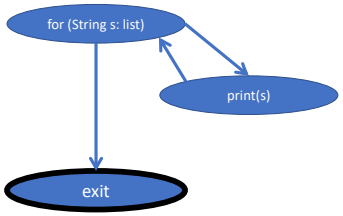


CFG Example – Enhanced Loop

```
ArrayList<String> list = ...
for (String string : list) {
  print(string);
}
```



```
ArrayList<String> list = ...
for (Iterator it = list.iterator(); it.hasNext();) {
  String string = it.next();
  print(string);
}
```



CFG Example – Enhanced Loop

```
ArrayList<String> list = ...
for (String string : list) {
  print(string);
}
```

```
ArrayList<String> list = ...
for (Iterator it = list.iterator(); it.hasNext();) {
  String string = it.next();
  print(string);
}
```

```
String[] list = ...
for (String string : list) {
  print(string);
}
```

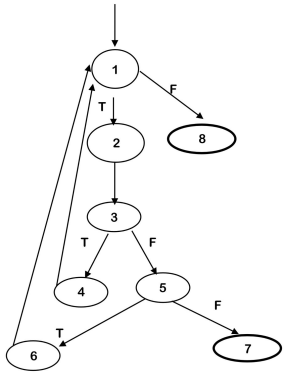
```
String[] list = ...
for (int i = 0; i < list.length; ++i) {
  String string = list[i];
  print(string);
}
```

CFG – Exercise – Draw a CFG with 8 Nodes

```
int binary_search(
    int a[],
    int low,
    int high,
    int target) {
  1  while (low <= high) {
  2    int middle = low + (high - low)/2;
  3    if (target < a[middle])
  4      high = middle - 1;
  5    else if (target > a[middle])
  6      low = middle + 1;
       else
  7      return middle;
     }
  8  return -1;
}
```


CFG – Exercise – Draw a CFG with 8 Nodes

```
int binary_search(
    int a[],
    int low,
    int high,
    int target) {
  1  while (low <= high) {
  2    int middle = low + (high - low)/2;
  3    if (target < a[middle])
  4      high = middle - 1;
  5    else if (target > a[middle])
  6      low = middle + 1;
       else
  7      return middle;
     }
  8  return -1;
}
```

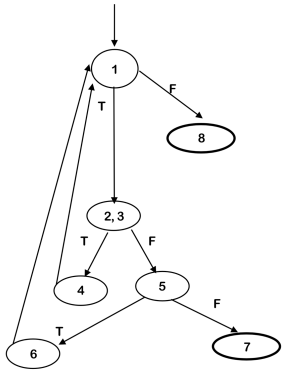


CFG – Exercise – Draw a CFG with 7 Nodes

```
int binary_search(
    int a[],
    int low,
    int high,
    int target) {
  1  while (low <= high) {
  2    int middle = low + (high - low)/2;
  3    if (target < a[middle])
  4      high = middle - 1;
  5    else if (target > a[middle])
  6      low = middle + 1;
       else
  7      return middle;
     }
  8  return -1;
}
```


CFG – Exercise – Draw a CFG with 7 Nodes

```
int binary_search(
    int a[],
    int low,
    int high,
    int target) {
  1  while (low <= high) {
  2    int middle = low + (high - low)/2;
  3    if (target < a[middle])
  4      high = middle - 1;
  5    else if (target > a[middle])
  6      low = middle + 1;
       else
  7      return middle;
     }
  8  return -1;
}
```



Infeasible Paths

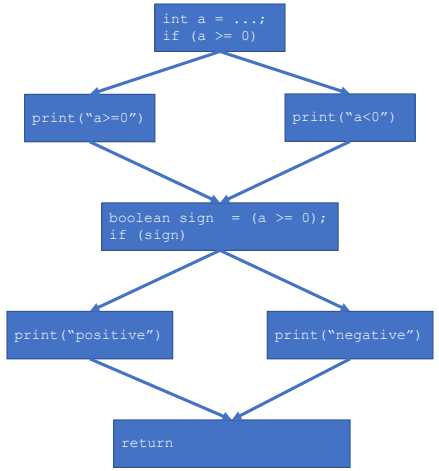
- Every executable sequence of statements corresponds to a path in CFG.
- Not all paths in CFG correspond to executable sequences
  - requires additional semantic information
  - “infeasible paths” are not an indication of a fault
- CFG usually overestimates the executable behavior

Infeasible Paths

```
int a = ...;

if (a >= 0) {
  println("a>=0");
} else {
  println("a<0");
}

boolean sign = (a >= 0);

if (sign) {
  prinln("positive");
} else {
  println("negative");
}
return;
```

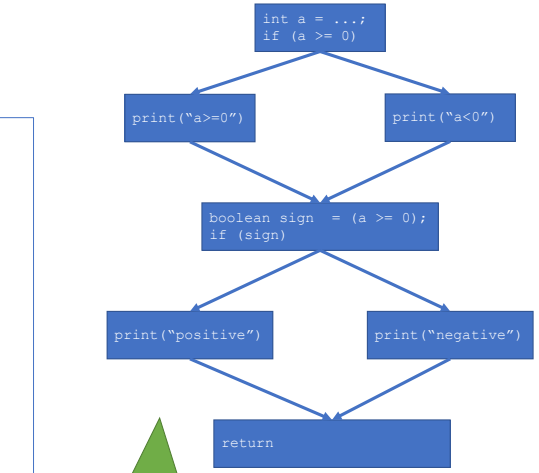

Infeasible Paths

```
int a = ...;

if (a >= 0) {
  println("a>=0");
} else {
  println("a<0");
}

boolean sign = (a >= 0);

if (sign) {
  prinln("positive");
} else {
  println("negative");
}
return;
```



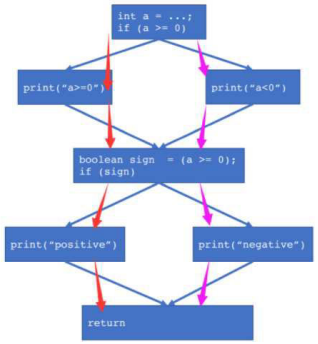
Infeasible Paths

```
int a = ...;

if (a >= 0) {
  println("a>=0");
} else {
  println("a<0");
}

boolean sign = (a >= 0);

if (sign) {
  prinln("positive");
} else {
  println("negative");
}
return;
```



I made a mistake in this CFG? What is it?

More about basic block



- Always construct CFG from AST.
- CFG:
  - Node: basic blocks
  - Edge: control flows

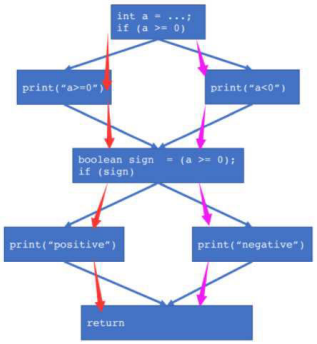
We learned that the final CFG of this example should have only two paths between the initial node and the last node. But what will the node in the middle ultimately look like? Say we want a minimal # of nodes.

Take the left path as an example, would it be a single basic block like this?
{print(a>=0);
boolean sign = (a>=0);
if(sign)
print("positive")}

Or, there has to be two blocks like the following:
Block A: {print(a>=0);
boolean sign = (a>=0);}

Block B: {print("positive")}

More about basic block



Basic Block 1?
print("a>=0")
boolean sign = …
if (sign)
print("positive")

Violates SESE (single entry single exit)

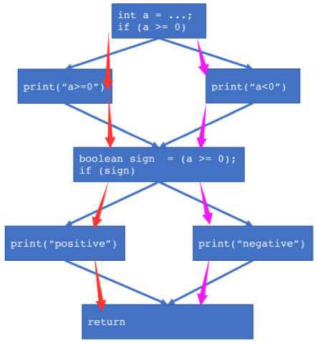
We learned that the final CFG of this example should have only two paths between the initial node and the last node. But what will the node in the middle ultimately look like? Say we want a minimal # of nodes.

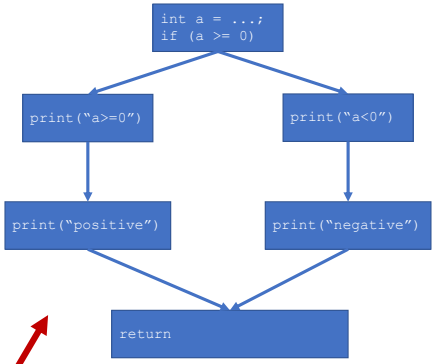
Take the left path as an example, would it be a single basic block like this?
{print(a>=0);
boolean sign = (a>=0);
if(sign)
print("positive")}

Or, there has to be two blocks like the following:
Block A: {print(a>=0);
boolean sign = (a>=0);}

Block B: {print("positive")}

More about basic block





We learned that the final CFG of this example should have only two paths between the initial node and the last node. But what will the node in the middle ultimately look like? Say we want a minimal # of nodes.

Take the left path as an example, would it be a single basic block like this?
{print(a>=0);
boolean sign = (a>=0);
if(sign)
print("positive")}

Or, there has to be two blocks like the following:
Block A: {print(a>=0);
boolean sign = (a>=0);}

Block B: {print("positive")}

Still wrong. This CFG represents a different program.

Recursive Function Calls?

- Follow AST
- Intra-procedurally, not inter-procedurally
- No need to inline

Structural Coverage

Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's and Arie Gurfinkel's.

Testing

- **Static Testing** [at compile time/ahead of time]
  - Automated Static Analysis
    - e.g., FindBugs, Error Prone, Clang Static Analyzer, Coverity, Facebook Infer
  - Code Review
- **Dynamic Testing** [at run time]
  - Black-box Testing, e.g., random testing
  - White-box Testing, e.g., symbolic execution
  - Grey-box Testing, e.g., coverage-guided fuzz testing
  - Unit testing, integration testing, system testing (end to end testing)
- Commonly, Testing ≡ Dynamic Testing

```java
public static void main (String[] args) {
  Set<Short> s = new HashSet<>();
  for (short i = 0; i < 100; i++) {
    s.add(i);
    s.remove(i - 1);
  }
  System.out.println(s.size());
```


Complete Testing/Exhaustive Testing?

- Completely test a nontrivial system
  - i.e., run the system on all possible inputs
- The number of potential inputs is infinite
- **Impossible**
  - Practical limitations: prohibitive in time and cost
  - Theoretical limitations: e.g., halting problem
- Need testing criteria

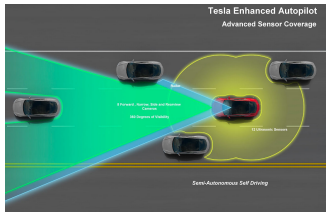
```java
if (x - 100 <= 0) {
  if (y - 100 <= 0) {
    if (x + y - 200 == 0) {
      crash();
    }
  }
}
```


Test Case

- **Test Case** [informal]
  - Input: What you feed to software
  - Output: What the software should output in response
- Our test cases have been easy to generate, but not always the case.
  - Some are hard to generate.

Testability

- The degree to which a system or a component facilitates testing
- **Observability**
  - how easy it is to observe the system's behavior
  - e.g., its outputs, effects on the environment, hardware and software
- **Controllability**
  - how easy to provide the system with needed inputs
  - how easy to get the system into the right state
- **Example**: embedded software
  - observability: behavior of hardware
  - controllability: input from sensors
  - solution: simulation



Test Case

- **Test Case** [informal]
  - Input: What you feed to software
  - Output: What the software should output in response
- **Test Set**: A set of test cases
- **Test Case**:
  - test case values
  - expected results
  - prefix values
  - postfix values
- **Expected Results**: The result that will be produced when executing the test if and only if the program satisfies its intended behavior.

Anatomy of a Test Case

$$\underbrace{\langle \text{on} \rangle}_{\text{prefix values}} \quad \underbrace{1\ 519\ 888\ 4567}_{\text{test case values}} \quad \underbrace{\underbrace{\langle \text{talk} \rangle}_{\text{verification values}} \quad \underbrace{\langle \text{end} \rangle}_{\text{exit codes}}}_{\text{postfix values}}$$

- Consider testing a cellphone from the *off* state.
- Test Case Values: The input values necessary to complete some execution of the software under test
  - Traditionally called Test Case
  - 1 519 888 4567
- Prefix Values : inputs to prepare software for test case values
  - the button <on>
- Postfix Values: inputs for software after test case values
  - **Verification Values**: inputs to show results of test case values
    - the button <talk>
  - **Exit Commands**: inputs to terminate program or to return it to initial state
    - the button <end>

A Real Test, but Junit first

```
@RunWith(JUnit4.class)
public class DemoTest {
  @Before
  public void setUp() {
    System.out.println("set up");
  }
  @After
  public void tearDown() {
    System.out.println("tear down");
  }
  @Test
  public void test_1() {
    System.out.println("test 1 in object " + this);
  }
  @Test
  public void test_2() {
    System.out.println("test 2 in objec " + this);
  }
}
```

A Real Test, but Junit first

```
@RunWith(JUnit4.class)
public class DemoTest {
  @Before
  public void setUp() {
    System.out.println("set up");
  }
  @After
  public void tearDown() {
    System.out.println("tear down");
  }
  @Test
  public void test_1() {
    System.out.println("test 1 in object " + this);
  }
  @Test
  public void test_2() {
    System.out.println("test 2 in objec " + this);
  }
}
```


A Real Test, but Junit first

```
@RunWith(JUnit4.class)
public class DemoTest {
  @Before
  public void setUp() {
    System.out.println("set up");
  }
  @After
  public void tearDown() {
    System.out.println("tear down");
  }
  @Test
  public void test_1() {
    System.out.println("test 1 in object " + this);
  }
  @Test
  public void test_2() {
    System.out.println("test 2 in objec " + this);
  }
}
```


A Real Test, but Junit first

```
@RunWith(JUnit4.class)
public class DemoTest {
  @Before
  public void setUp() {
    System.out.println("set up");
  }
  @After
  public void tearDown() {
    System.out.println("tear down");
  }
  @Test
  public void test_1() {
    System.out.println("test 1 in object " + this);
  }
  @Test
  public void test_2() {
    System.out.println("test 2 in objec " + this);
  }
}
```


A Real Test, but Junit first

```
@RunWith(JUnit4.class)
public class DemoTest {
  @Before
  public void setUp() {
    System.out.println("set up");
  }
  @After
  public void tearDown() {
    System.out.println("tear down");
  }
  @Test
  public void test_1() {
    System.out.println("test 1 in object " + this);
  }
  @Test
  public void test_2() {
    System.out.println("test 2 in objec " + this);
  }
}
```

Pseudocode to Run Tests

```
DemoTest test1 = new DemoTest();
test1.setUp();
test1.test_1();
test1.tearDown();

DemoTest test2 = new DemoTest();
test2.setUp();
test2.test_2();
test2.tearDown();
```

A Real Test, but Junit first

```
@RunWith(JUnit4.class)
public class DemoTest {
  @Before
  public void setUp() {
    System.out.println("set up");
  }
  @After
  public void tearDown() {
    System.out.println("tear down");
  }
  @Test
  public void test_1() {
    System.out.println("test 1 in object " + this);
  }
  @Test
  public void test_2() {
    System.out.println("test 2 in objec " + this);
  }
}
```

Pseudocode to Run Tests

```
DemoTest test1 = new DemoTest();
test1.setUp();
test1.test_1();
test1.tearDown();

DemoTest test2 = new DemoTest();
test2.setUp();
test2.test_2();
test2.tearDown();
```

Output

```
set up
test 1 in object p.DemoTest@4f2410ac
tear down

set up
test 2 in objec p.DemoTest@50134894
tear down
```


A Junit test case for MoreFilesTest in Guava

```
public class MoreFilesTest {

  private Path tempDir;

  @Before
  public void setUp() throws Exception {
    tempDir = Files.createTempDirectory("MoreFilesTest");
  }

  @Test
  public void testCreateParentDirectories_oneParentNeeded() {
    Path path = tempDir.resolve("parent/nonexistent.file");
    Path parent = path.getParent();
    assertFalse(Files.exists(parent));
    MoreFiles.createParentDirectories(path);
    assertTrue(Files.exists(parent));
  }

  @After
  protected void tearDown() throws Exception {
    if (tempDir != null) {
      // delete tempDir and its contents
      ……
    }
  }
}
```


A Junit test case for MoreFilesTest in Guava

```
public class MoreFilesTest {

  private Path tempDir;

  @Before
  public void setUp() throws Exception {
    tempDir = Files.createTempDirectory("MoreFilesTest");
  }

  @Test
  public void testCreateParentDirectories_oneParentNeeded() {
    Path path = tempDir.resolve("parent/nonexistent.file");
    Path parent = path.getParent();
    assertFalse(Files.exists(parent));
    MoreFiles.createParentDirectories(path);
    assertTrue(Files.exists(parent));
  }

  @After
  protected void tearDown() throws Exception {
    if (tempDir != null) {
      // delete tempDir and its contents
      ……
    }
  }
}
```

Prefix Value

A Junit test case for MoreFilesTest in Guava

```
public class MoreFilesTest {

  private Path tempDir;

  @Before
  public void setUp() throws Exception {
    tempDir = Files.createTempDirectory("MoreFilesTest");
  }

  @Test
  public void testCreateParentDirectories_oneParentNeeded() {
    Path path = tempDir.resolve("parent/nonexistent.file");
    Path parent = path.getParent();
    assertFalse(Files.exists(parent));
    MoreFiles.createParentDirectories(path);
    assertTrue(Files.exists(parent));
  }

  @After
  protected void tearDown() throws Exception {
    if (tempDir != null) {
      // delete tempDir and its contents
      ……
    }
  }
}
```

Prefix Value

Postfix Value

A Junit test case for MoreFilesTest in Guava

```
public class MoreFilesTest {

  private Path tempDir;

  @Before
  public void setUp() throws Exception {
    tempDir = Files.createTempDirectory("MoreFilesTest");
  }

  @Test
  public void testCreateParentDirectories_oneParentNeeded() {
    Path path = tempDir.resolve("parent/nonexistent.file");
    Path parent = path.getParent();
    assertFalse(Files.exists(parent));
    MoreFiles.createParentDirectories(path);
    assertTrue(Files.exists(parent));
  }

  @After
  protected void tearDown() throws Exception {
    if (tempDir != null) {
      // delete tempDir and its contents
      ……
    }
  }
}
```

Prefix Value

Test case value: path
Expected Results:
- before: no parent
- after: parent created

'Files.exits(parent)' might be a postfix value?

Postfix Value

Test Requirement

- Test Requirement: A test requirement is a specific element of a software artifact that a test case must satisfy or cover.
- TR denotes a set of test requirements
- Example 1
  - ICE Cream Machine with flavors: vanilla, chocolate, mint
  - One test requirement: test a chocolate cone
- Example 2
  - execute the true branch of the innermost if stmt

```
if (x - 100 <= 0) {
  if (y - 100 <= 0) {
    if (x + y - 200 == 0) {
      crash();
    }
  }
}
```


Coverage Criterion

- A coverage criterion is a rule or collection or rules that impose test requirements on a test set
  - Coverage criterion is a recipe for generating TR in a systematic way
- Example 1
  - Flavor criterion: cover all flavors (More general)
  - TR = {test a chocolate cone, test a vanilla cone, test a mint cone}
- Example 2
  - cover all branches
  - How many test requirements?

```
if (x - 100 <= 0) {
  if (y - 100 <= 0) {
    if (x + y - 200 == 0) {
      crash();
    }
  }
}
```


Measure Test Sets

- How to know how good a test set is?
  - Testing an ice cream stand: cover all flavors
  - Test Set 1:
    - 3 chocolate cones, 1 vanilla cone
  - Test Set 2:
    - 1 chocolate cone, 1 vanilla cone, 1 mint cone

Coverage

- **C**: a coverage criterion
- **TR**: defined by **C**
- **T**: a test set

- the test set T satisfies C iff for every test requirement $tr \in TR$, at least one $t \in T$ satisfies tr.

Coverage – Example, Ice Cream Stand

- **C**: a coverage criterion
  - cover all flavors
- **TR**: defined by **C**
  - test a chocolate cone, test a vanilla cone, test a mint cone
- **T**: a test set
  - Test Set 1: 3 chocolate cone, 1 vanilla cone
  - Test Set 2: 1 chocolate cone, 1 vanilla cone, 1 mint cone

- the test set T satisfies C iff for every test requirement $tr \in TR$, at least one $t \in T$ satisfies tr.

Infeasible Test Requirements

```
if (false)
  unreachableCall()
```

Real code from the Linux Kernel

```
while (0) {
  local_irq_disable();
}
```

Statement criterion cannot be satisfied for many programs.

- **C**: a coverage criterion
- **TR**: defined by **C**
- **T**: a test set

- **Coverage**: the test set T satisfies C iff for every test requirement $tr \in TR$, at least one $t \in T$ satisfies tr.

- **C**: a coverage criterion
- **TR**: defined by **C**
- **T**: a test set

- **Coverage**: the test set T satisfies C iff for every test requirement $tr \in TR$, at least one $t \in T$ satisfies tr.
- **Coverage Level**: the coverage level is the **ratio** of the number of test requirements satisfied by T to the size of TR
  - TR = {flavor=chocolate, flavor=vanilla, flavor=mint}
  - Test set 1 T1 = {3 chocolate cones, 1 vanilla cone}
  - Coverage Level = 2/3 = 66.7%

- Coverage levels help us evaluate the goodness of a test set, especially in the presence of infeasible test requirements.

Where do test requirements come from?

- Functional (black box, specification-based): from software specifications
  - Example: If spec requires robust recovery from power failure, test requirements should include simulated power failure
- Structural (white box): from code
  - Example: Traverse each program loop one or more times
- Model-based: from model of system
  - Models used in specification or design, or derived from code
  - Example: Exercise all transitions in communication protocol model
- Fault-based: from hypothesized faults (common bugs)
  - Example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs

Code Coverage

Basic code coverage

Advanced code coverage

- Line Coverage
- Statement
- Function/Method coverage
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- Path coverage
- Loop coverage
- Mutation coverage

Line Coverage

- Percentage of source code lines executed by test cases
  - For developer easiest to work with
  - Precise percentage depends on layout?
    - `int x = 10; if (z++ < x) y = x + z;`

```c
#include <stdio.h>


int main() {
  int i = 0;
  if (i != 0) {
    printf("i!=0\n");
  } else {
    printf("i==0\n");
  }
  return 0;
}
```

```
$ gcc -fprofile-arcs -ftest-coverage t.c -o t.o

$ ./t.o

$ ls t.gc*
t.gcda t.gcno

$ gcov t.c
File 't.c'
Lines executed:66.67% of 6
t.c:creating 't.c.gcov'
```



```c
#include <stdio.h>


int main() {
  int i = 0;
  if (i != 0) {
    printf("i!=0\n");
  } else {
    printf("i==0\n");
  }
  return 0;
}
```

```
        -:    0:Source:t.c
        -:    0:Graph:t.gcno
        -:    0:Data:t.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main() {
        1:    4:  int i = 0;
        1:    5:  if (i != 0) {
    #####:    6:    printf("i!=0\n");
    #####:    7:  } else {
        1:    8:    printf("i==0\n");
        -:    9:  }
        1:   10:  return 0;
        -:   11:}
```



```c
#include <stdio.h>


int main() {
  int i = 0;
  if (i != 0) {
    printf("i!=0\n");
  } else {
    printf("i==0\n");
  }
  return 0;
}
```

```
        -:    0:Source:t.c
        -:    0:Graph:t.gcno
        -:    0:Data:t.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main() {
        1:    4:  int i = 0;
        1:    5:  if (i != 0) {
    #####:    6:    printf("i!=0\n");
    #####:    7:  } else {
        1:    8:    printf("i==0\n");
        -:    9:  }
        1:   10:  return 0;
        -:   11:}
```

In practice, coverage not based on lines, but on control flow graph.

Statement or Node Coverage

- Each statement (or node in the CFG) must be executed at least once

```c
void foo(int z) {
  int x = 10;
  if (z++ < x) {
    x += z;
  }
  return;
}
```

Coverage Level: $\frac{\#executed\ statements}{\#statements}$

Statement or Node Coverage

- Each statement (or node in the CFG) must be executed at least once

```
void foo(int z) {
  int x = 10;
  if (z++ < x) {
    x += z;
  }
  return;
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

Coverage Level: $\frac{\#executed\ statements}{\#statements}$

Statement or Node Coverage

- Each statement (or node in the CFG) must be executed at least once

```
void foo(int z) {
  int x = 10;
  if (z++ < x) {
    x += z;
  }
  return;
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

```
@Test
void testFoo() {
  foo(5);
}
```

Coverage Level: $\frac{\#executed\ statements}{\#statements}$

Edge Coverage [Informal]

- Every edge or node should be executed at least once

```
void foo(int z) {
  int x = 10;
  if (z++ < x) {
    x += z;
  }
  return;
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

Coverage Level: $\frac{\#executed\ edges}{\#edges}$

Edge Coverage [Informal]

- Every edge or node should be executed at least once

```
void foo(int z) {
  int x = 10;
  if (z++ < x) {
    x += z;
  }
  return;
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

```
@Test
void testFoo() {
  foo(5);
}
```

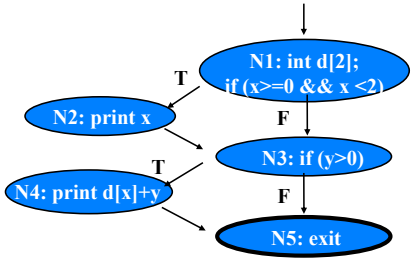
Coverage Level: $\frac{\#executed\ edges}{\#edges}$

Coverage Criterion Subsumption

- A coverage criterion C1 subsumes C2 iff every set of test cases that satisfies criterion C1 also satisfies C2.
- Must be true for every set of test cases
- Edge coverage (EC) subsumes Node coverage (NC)
  - Which one is stronger?
- Subsumption is a rough guide for comparing criteria, though it is difficult to use in practice.

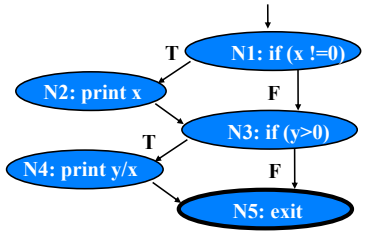
Stronger Coverage Criterion Helps Find More Bugs

- Path [N1, N2, N3, N4, N5]
  - satisfies node coverage, but not edge coverage
  - The corresponding test case passes
  - No bug found.
- Path [N1, N3, N4, N5]
  - Buffer overflow bug!



Stronger Coverage Criterion Helps Find More Bugs

- Path [N1, N2, N3, N4, N5]
  - satisfies node coverage, but not edge coverage
  - The corresponding test case passes
  - No bug found.
- Path [N1, N3, N4, N5], or [N1, N3, N4]
  - division-by-zero bug!



Graph Coverage

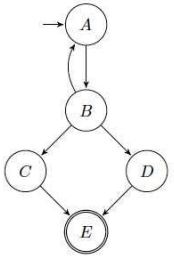
Chengnian Sun
cnsun@

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Graphs in Software

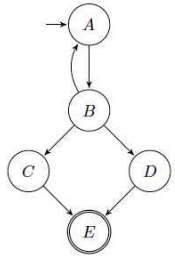
- control flow graphs from source
- design structures
- finite state machines
- state charts
- use cases

Graph Coverage



N : Set of nodes $\{A, B, C, D, E\}$
N_0 : Set of initial nodes $\{A\}$
N_f : Set of final nodes $\{E\}$
$E \subseteq N \times N$: Edges, e.g. (A, B) and (C, E);
C is the predecessor and E is the successor in (C, E).

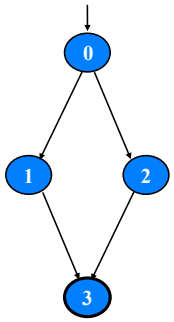
Graph Coverage



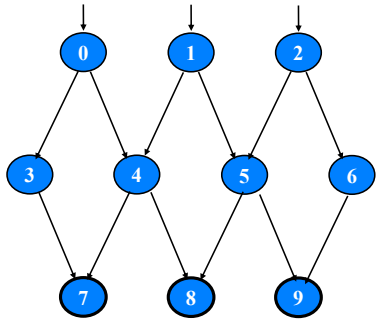
N : Set of nodes $\{A, B, C, D, E\}$
N_0 : Set of initial nodes $\{A\}$
N_f : Set of final nodes $\{E\}$
$E \subseteq N \times N$: Edges, e.g. (A, B) and (C, E);
C is the predecessor and E is the successor in (C, E).

A graph should have NON-EMPTY initial nodes, and NON-EMPTY final nodes.

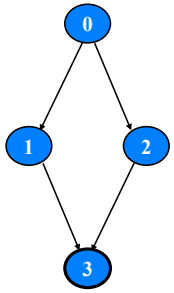
Three Example Graphs



N_0 = { 0 }
N_f = { 3 }

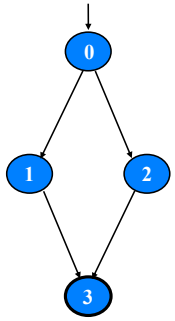


N_0 = { 0, 1, 2 }
N_f = { 7, 8, 9 }



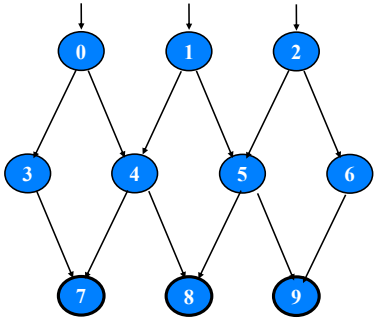
N_0 = { }
N_f = { 3 }

Three Example Graphs



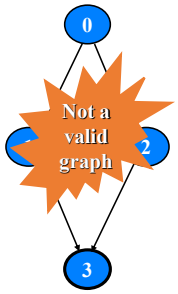
$N_0 = \{ 0 \}$

$N_f = \{ 3 \}$



$N_0 = \{ 0, 1, 2 \}$

$N_f = \{ 7, 8, 9 \}$

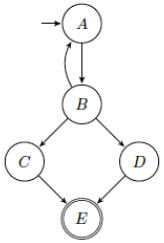


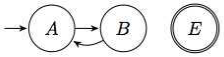
$N_0 = \{ \}$

$N_f = \{ 3 \}$

Subgraph

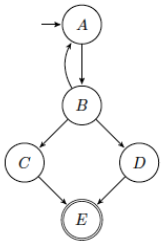
Subgraph: Let G' be a subgraph of G; then the nodes of G' must be a subset N_{sub} of N. Then the initial nodes of G' are $N_0 \cap N_{\text{sub}}$ and its final nodes are $N_f \cap N_{\text{sub}}$. The edges of G' are $E \cap (N_{\text{sub}} \times N_{\text{sub}})$.



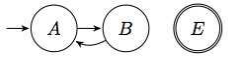


Subgraph

Subgraph: Let G' be a subgraph of G; then the nodes of G' must be a subset N_{sub} of N. Then the initial nodes of G' are $N_0 \cap N_{\text{sub}}$ and its final nodes are $N_f \cap N_{\text{sub}}$. The edges of G' are $E \cap (N_{\text{sub}} \times N_{\text{sub}})$.

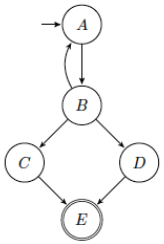


What if we remove [B, A]?

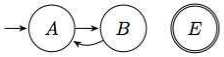


Subgraph

Subgraph: Let G' be a subgraph of G; then the nodes of G' must be a subset N_{sub} of N. Then the initial nodes of G' are $N_0 \cap N_{\text{sub}}$ and its final nodes are $N_f \cap N_{\text{sub}}$. The edges of G' are $E \cap (N_{\text{sub}} \times N_{\text{sub}})$.



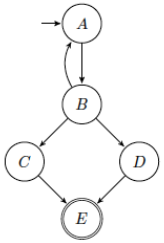
What if we remove [B, A]?



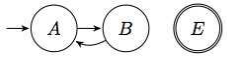
Not the subgraph we usually refer to.

Subgraph

Subgraph: Let G' be a subgraph of G; then the nodes of G' must be a subset N_{sub} of N. Then the initial nodes of G' are $N_0 \cap N_{\text{sub}}$ and its final nodes are $N_f \cap N_{\text{sub}}$. The edges of G' are $E \cap (N_{\text{sub}} \times N_{\text{sub}})$.



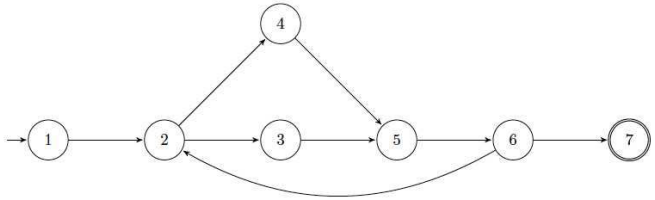
What if we remove [B, A]?



Not the subgraph we usually refer to.

A subgraph can be an invalid graph, e.g., N_sub={C}

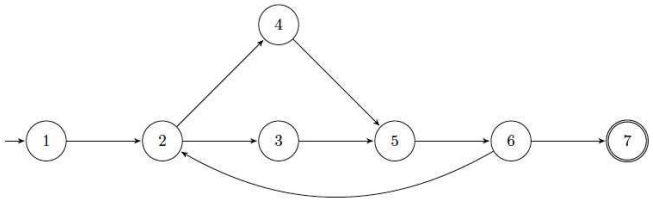
Path



- path 1: $[2, 3, 5]$, with length 2.
- path 2: $[1, 2, 3, 5, 6, 2]$, with length 5.
- not a path: $[1, 2, 5]$.

- A path is a sequence of nodes from a graph G whose adjacent pairs all belong to the set of edges E of G.
- length: number of edges

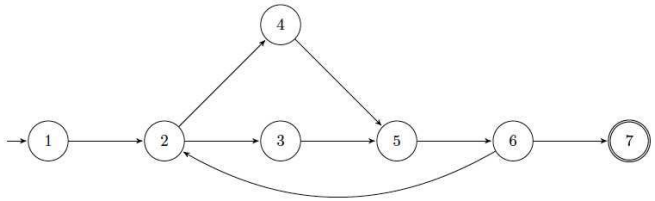
Subpath



- path 1: $[2, 3, 5]$, with length 2.
- path 2: $[1, 2, 3, 5, 6, 2]$, with length 5.
- not a path: $[1, 2, 5]$.

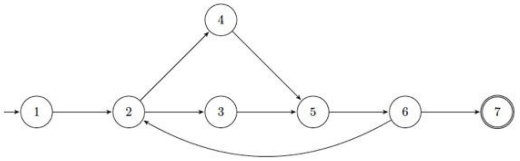
- A subpath is a subsequence of a path.
  - This textbook definition is ambiguous.
- Is [1,2,5] a subpath of [1,2,3,5,6,2]?

Subpath



- A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.
  - [1,2,5] is a subsequence of [1,2,3,5,6,2].
- But here should interpret subpath as a substring of a path, i.e., all edges should be in the path.
  - Therefore, a subpath is still a path.
  - [1,2,5] is NOT a subpath of [1,2,3,5,6,2].

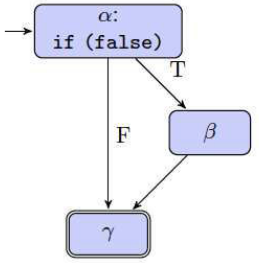
Test Path



- A test path is a path p [possibly of length 0] that starts at some node in N_0 and ends at some node in N_f.

- Test path examples:
  - [1, 2, 3, 5, 6, 7]
  - [1, 2, 3, 5, 6, 2, 3, 5, 6, 7]

Paths & Semantics



β is never executed!

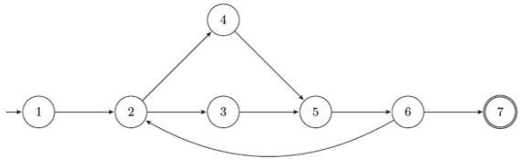
- Some paths in a control flow graph may not correspond to program *semantics*.
- In this course, we generally
  - only talk about the *syntax* of a graph -- its nodes and edges -- and
  - not its *semantics*.

Syntactical and Semantic Reachability

- A node n is *syntactically* reachable from ni if there exists a path from ni to n.
- A node n is *semantically* reachable if one of the paths from ni to n can be reached on some input.

- Standard graph algorithms, like breadth-first search and depth-first search, can compute syntactic reachability.
- Semantic reachability is undecidable.

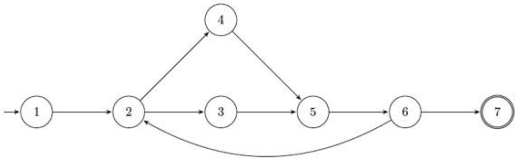
Reachability



- We define $reach_G(X)$ as the subgraph that is syntactically reachable from X.

- X is
  - a node,
  - an edge, or
  - a set of nodes or edges.

Reachability



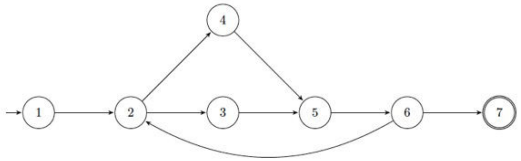
- We define $reach_{\mathrm{G}}(X)$ as the subgraph that is syntactically reachable from X.

- X is
  - a node,
  - an edge, or
  - a set of nodes or edges.

$reach_{\mathrm{G}}(1)$?

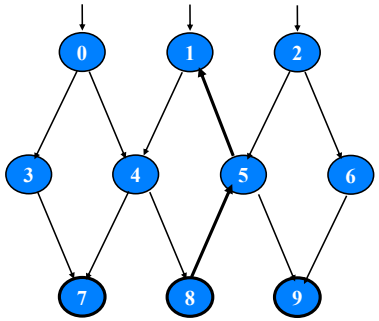
Syntactical Reachability

- $reach_{\mathrm{G}}(2)$
- $reach_{\mathrm{G}}(5)$
- $reach_{\mathrm{G}}(7)$



Reachability Example

- $reach_{\mathrm{G}}(0)$
- $reach_{\mathrm{G}}(\{0,2\})$
- $reach_{\mathrm{G}}([2,6])$
- $reach_{\mathrm{G}}(\{2,6\})$



$N_0 = \{ 0, 1, 2 \}$

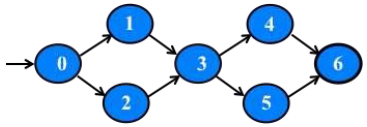
$N_f = \{ 7, 8, 9 \}$

$reach_{\mathrm{G}}(N_0)$

- When we talk about the nodes or edges in a graph G in a coverage criterion, we'll generally mean $reach_{\mathrm{G}}(N_0)$.

- The unreachable nodes tend to
  - be uninteresting; and
  - frustrate coverage criteria

SESEs, subpath, and tours



- SESE graphs: All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N_0 and N_f have exactly one node.

Double-diamond graph
Four test paths
p=[0, 1, 3, 4, 6]
[0, 1, 3, 5, 6]
[0, 2, 3, 4, 6]
[0, 2, 3, 5, 6]

- p visits node 3 and edge [0, 1]
- 3 ∈ p
- [0, 1] ∈ p

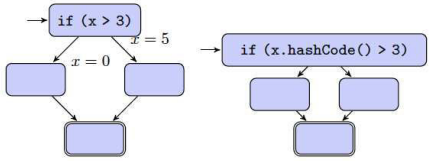
- p0 = [1, 3, 4], p0 is a subpath of p, and conversely, p **tours** p0.
- Any path tours itself.

Connect Test Cases and Test Paths

- Connect test cases and test paths with a mapping $path_G$ from test cases to test paths
  - e.g., $path_G(t)$ is the set of test paths corresponding to test case t.
    - Any path that can be triggered by t is in $path_G(t)$
    - Each time you run t, you get only one path
  - Usually just write $path$, as G is obvious from the context
  - Lift the definition of path to test set T by defining $path(T)$
  - Each test case gives at least one test path.
    - If the software is deterministic, then each test case gives exactly one test path;
    - otherwise, one test case might give multiple different test paths.

Deterministic and Nondeterministic CFG

Here's an example of deterministic and nondeterministic control-flow graphs:

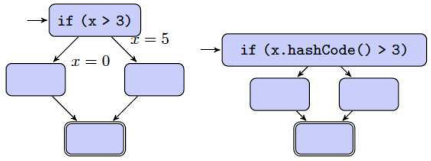


Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.

Deterministic and Nondeterministic CFG

Here's an example of deterministic and nondeterministic control-flow graphs:



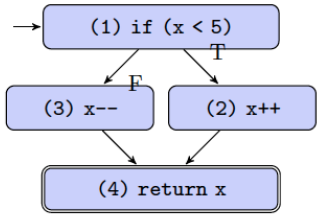

```
HashSet<Object> set = ...
for (String s : set) {
  print s;
}

LinkedHashSet<Object> set = ...
for (String s : set) {
  print s;
}
```

Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.

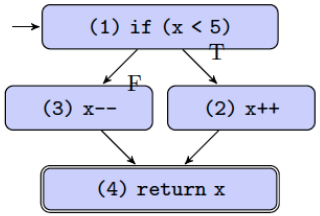
Connecting Test Cases, Test Paths & CFG

```
int foo(int x) {
  if (x < 5) {
    x ++;
  } else {
    x --;
  }
  return x;
}
```



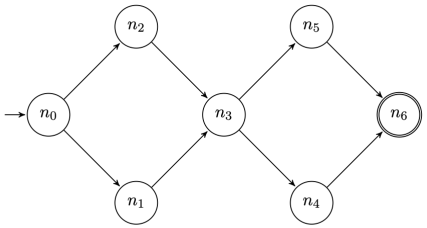
Connecting Test Cases, Test Paths & CFG

```
int foo(int x) {
  if (x < 5) {
    x ++;
  } else {
    x --;
  }
  return x;
}
```



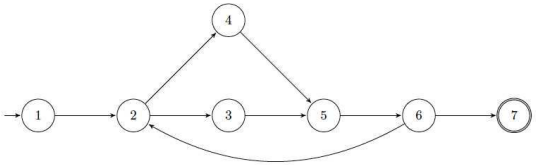
- Test case: $x = 5$; test path: $[(1), (3), (4)]$.
- Test case: $x = 2$; test path: $[(1), (2), (4)]$.

Node Coverage (NC)



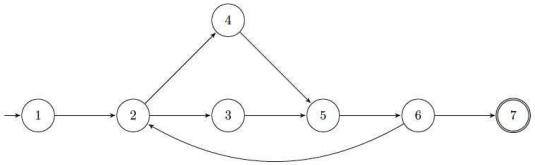
- $\forall n \in reach_G(N_0)$, *TR contains a requirement to visit node n.*
- *Node Coverage [NC]: TR contains each **reachable** node in G.*
- TR = {n0,n1,n2,n3,n4,n5,n6}

Edge Coverage (EC)



- *TR contains each **reachable** path of length up to 1, inclusive, in G.*
- *TR = {[1,2], [2,4], [2,3], [3,5], [4,5], [5,6] [6,7], [6,2]}*

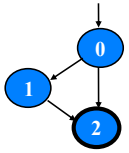
Edge Pair Coverage (EPC)



- *TR contains each **reachable** path of length up to 2, inclusive, in G.*
- *TR= {[1,2,3], [1,2,4], [2,3,5], [2,4,5], [3,5,6], [4,5,6], [5,6,2], [5,6,7], [6,2,3], [6,2,4]}*

NC vs. EC

- Edge coverage is slightly stronger than node coverage.
- NC and EC are only different when there is an edge and another subpath between a pair of nodes [as in an "if-else" statement]



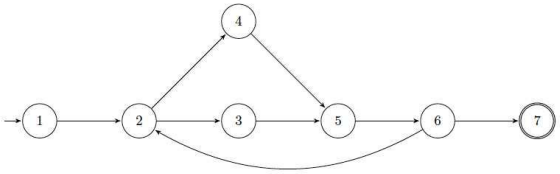
Node Coverage : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

Edge Coverage : TR = { [0,1], [0, 2], [1, 2] }
Test Paths = [0, 1, 2]
[0, 2]

Simple Path

- A path is simple if no node appears more than once in the path, except that the first and last nodes may be the same.
- Some properties of simple paths:
  - no internal loops;
  - can bound their length;
  - can create any path by composing simple paths; and
  - many simple paths exist [too many!]

Simple Path Examples

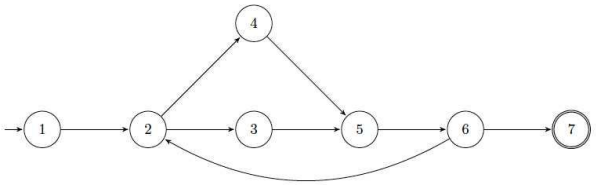


- Simple path examples:
  - [1, 2, 3, 5, 6, 7]
  - [1, 2, 4]
  - [2,3,5,6,2]
- Not simple Path: [1,2,3,5,6,2,4]

Prime Path

- Because there are so many simple paths, let's instead consider prime paths, which are simple paths of maximal length.
- A path is prime if it is simple and does not appear as a proper subpath of any other simple path.

Prime Path Examples



- Prime path examples:
  - [1, 2, 3, 5, 6, 7]
  - [1, 2, 4, 5, 6, 7]
  - [6, 2, 4, 5, 6]
- Not a prime path: [3, 5, 6, 7]

Prime Path Coverage (PPC)

- Prime Path Coverage [PPC]: TR contains each prime path in G.

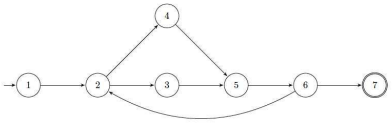
- There is a problem with using PPC as a coverage criterion: a prime path may be infeasible but contains feasible simple paths.

More Path Coverage Criterions

- Complete Path Coverage [CPC]: TR contains all paths in G.

- Specified Path Coverage [SPC]: TR contains a specified set S of paths.

Prime Path Example

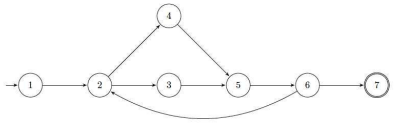


Simple paths

Len 0
[1]
[2]
[3]
[4]
[5]
[6]
[7] !

! means path terminates

Prime Path Example

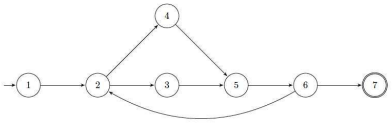


Simple paths

Len 0	Len 1
[1]	[1,2]
[2]	[2,4]
[3]	[2,3]
[4]	[3,5]
[5]	[4,5]
[6]	[5,6]
[7] !	[6,7] !
	[6,2]

! means path terminates

Prime Path Example



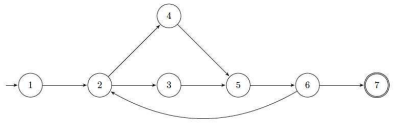
Simple paths

Len 0	Len 1
[1] x	[1,2]
[2] x	[2,4]
[3] x	[2,3]
[4] x	[3,5]
[5] x	[4,5]
[6] x	[5,6]
[7] !	[6,7] !
	[6,2]

x means not prime paths.

! means path terminates

Prime Path Example



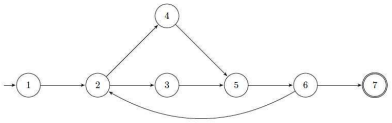
Simple paths

Len 0	Len 1	Len 2
[1] x	[1,2]	[1,2,4]
[2] x	[2,4]	[1,2,3]
[3] x	[2,3]	[2,4,5]
[4] x	[3,5]	[2,3,5]
[5] x	[4,5]	[3,5,6]
[6] x	[5,6]	[4,5,6]
[7] !	[6,7] !	[5,6,7] !
	[6,2]	[5,6,2]
		[6,2,4]
		[6,2,3]

x means not prime paths.

! means path terminates

Prime Path Example



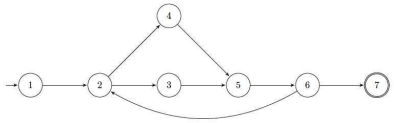
Simple paths

Len 0	Len 1	Len 2
[1] x	[1,2] x	[1,2,4]
[2] x	[2,4] x	[1,2,3]
[3] x	[2,3] x	[2,4,5]
[4] x	[3,5] x	[2,3,5]
[5] x	[4,5] x	[3,5,6]
[6] x	[5,6] x	[4,5,6]
[7] !	[6,7] !	[5,6,7] !
	[6,2] x	[5,6,2]
		[6,2,4]
		[6,2,3]

x means not prime paths.

! means path terminates

Prime Path Example



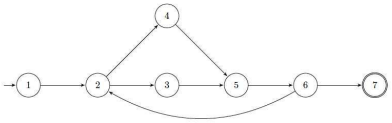
Simple paths

Len 0	Len 1	Len 2	Len 3
[1] x	[1,2] x	[1,2,4]	[1,2,4,5]
[2] x	[2,4] x	[1,2,3]	[1,2,3,5]
[3] x	[2,3] x	[2,4,5]	[2,4,5,6]
[4] x	[3,5] x	[2,3,5]	[2,3,5,6]
[5] x	[4,5] x	[3,5,6]	[3,5,6,7] !
[6] x	[5,6] x	[4,5,6]	[3,5,6,2]
[7] !	[6,7] !	[5,6,7] !	[4,5,6,7] !
	[6,2] x	[5,6,2]	[4,5,6,2]
		[6,2,4]	[5,6,2,4]
		[6,2,3]	[5,6,2,3]
			[6,2,4,5]
			[6,2,3,5]

x means not prime paths.

! means path terminates

Prime Path Example



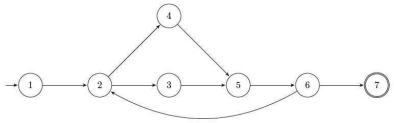
Simple paths

Len 0	Len 1	Len 2	Len 3
[1] x	[1,2] x	[1,2,4] x	[1,2,4,5]
[2] x	[2,4] x	[1,2,3] x	[1,2,3,5]
[3] x	[2,3] x	[2,4,5] x	[2,4,5,6]
[4] x	[3,5] x	[2,3,5] x	[2,3,5,6]
[5] x	[4,5] x	[3,5,6] x	[3,5,6,7] !
[6] x	[5,6] x	[4,5,6] x	[3,5,6,2]
[7] !	[6,7] !	[5,6,7] !	[4,5,6,7] !
	[6,2] x	[5,6,2] x	[4,5,6,2]
		[6,2,4] x	[5,6,2,4]
		[6,2,3] x	[5,6,2,3]
			[6,2,4,5]
			[6,2,3,5]

x means not prime paths.

! means path terminates

Prime Path Example



Simple paths

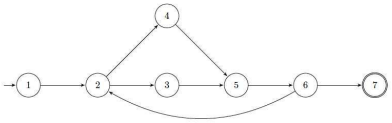
Len 0	Len 1	Len 2	Len 3	Len 4
[1] x	[1,2] x	[1,2,4] x	[1,2,4,5]	[1,2,4,5,6]
[2] x	[2,4] x	[1,2,3] x	[1,2,3,5]	[1,2,3,5,6]
[3] x	[2,3] x	[2,4,5] x	[2,4,5,6]	[2,4,5,6,7] !
[4] x	[3,5] x	[2,3,5] x	[2,3,5,6]	[2,4,5,6,2] *
[5] x	[4,5] x	[3,5,6] x	[3,5,6,7] !	[2,3,5,6,7] !
[6] x	[5,6] x	[4,5,6] x	[3,5,6,2]	[2,3,5,6,2] *
[7] !	[6,7] !	[5,6,7] !	[4,5,6,7] !	[3,5,6,2,4]
	[6,2] x	[5,6,2] x	[4,5,6,2]	[3,5,6,2,3] *
		[6,2,4] x	[5,6,2,4]	[4,5,6,2,4] *
		[6,2,3] x	[5,6,2,3]	[4,5,6,2,3]
			[6,2,4,5]	[5,6,2,4,5] *
			[6,2,3,5]	[5,6,2,3,5] *
				[6,2,4,5,6] *
				[6,2,3,5,6] *

*** denotes path cycles.**

x means not prime paths.

! means path terminates

Prime Path Example



Simple paths

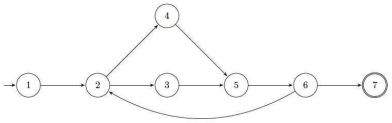
Len 0	Len 1	Len 2	Len 3	Len 4	Len 5
[1] x	[1,2] x	[1,2,4] x	[1,2,4,5] x	[1,2,4,5,6]	[1,2,4,5,6,7] !
[2] x	[2,4] x	[1,2,3] x	[1,2,3,5] x	[1,2,3,5,6]	[1,2,3,5,6,7] !
[3] x	[2,3] x	[2,4,5] x	[2,4,5,6] x	[2,4,5,6,7] !	
[4] x	[3,5] x	[2,3,5] x	[2,3,5,6] x	[2,4,5,6,2] *	
[5] x	[4,5] x	[3,5,6] x	[3,5,6,7] !	[2,3,5,6,7] !	
[6] x	[5,6] x	[4,5,6] x	[3,5,6,2] x	[2,3,5,6,2] *	
[7] !	[6,7] !	[5,6,7] !	[4,5,6,7] !	[3,5,6,2,4]	
	[6,2] x	[5,6,2] x	[4,5,6,2] x	[3,5,6,2,3] *	
		[6,2,4] x	[5,6,2,4] x	[4,5,6,2,4] *	
		[6,2,3] x	[5,6,2,3] x	[4,5,6,2,3]	
			[6,2,4,5] x	[5,6,2,4,5] *	
			[6,2,3,5] x	[5,6,2,3,5] *	
				[6,2,4,5,6] *	
				[6,2,3,5,6] *	

* denotes path cycles.

x means not prime paths.

! means path terminates

Prime Path Example



Simple paths

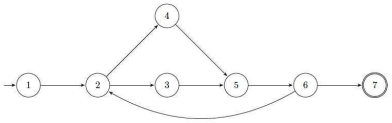
Len 0	Len 1	Len 2	Len 3	Len 4	Len 5
[1] x	[1,2] x	[1,2,4] x	[1,2,4,5] x	[1,2,4,5,6] x	[1,2,4,5,6,7] !
[2] x	[2,4] x	[1,2,3] x	[1,2,3,5] x	[1,2,3,5,6] x	[1,2,3,5,6,7] !
[3] x	[2,3] x	[2,4,5] x	[2,4,5,6] x	[2,4,5,6,7] !	
[4] x	[3,5] x	[2,3,5] x	[2,3,5,6] x	[2,4,5,6,2] *	
[5] x	[4,5] x	[3,5,6] x	[3,5,6,7] !	[2,3,5,6,7] !	
[6] x	[5,6] x	[4,5,6] x	[3,5,6,2] x	[2,3,5,6,2] *	
[7] !	[6,7] !	[5,6,7] !	[4,5,6,7] !	[3,5,6,2,4]	
	[6,2] x	[5,6,2] x	[4,5,6,2] x	[3,5,6,2,3] *	
		[6,2,4] x	[5,6,2,4] x	[4,5,6,2,4] *	
		[6,2,3] x	[5,6,2,3] x	[4,5,6,2,3]	
			[6,2,4,5] x	[5,6,2,4,5] *	
			[6,2,3,5] x	[5,6,2,3,5] *	
				[6,2,4,5,6] *	
				[6,2,3,5,6] *	

* denotes path cycles.

x means not prime paths.

! means path terminates

Prime Path Example



Simple paths

Len 0	Len 1	Len 2	Len 3	Len 4	Len 5
[1] x	[1,2] x	[1,2,4] x	[1,2,4,5] x	[1,2,4,5,6] x	[1,2,4,5,6,7] !
[2] x	[2,4] x	[1,2,3] x	[1,2,3,5] x	[1,2,3,5,6] x	[1,2,3,5,6,7] !
[3] x	[2,3] x	[2,4,5] x	[2,4,5,6] x	[2,4,5,6,7] !	
[4] x	[3,5] x	[2,3,5] x	[2,3,5,6] x	[2,4,5,6,2] *	
[5] x	[4,5] x	[3,5,6] x	[3,5,6,7] !	[2,3,5,6,7] !	
[6] x	[5,6] x	[4,5,6] x	[3,5,6,2] x	[2,3,5,6,2] *	
[7] !	[6,7] !	[5,6,7] !	[4,5,6,7] !	[3,5,6,2,4]	
	[6,2] x	[5,6,2] x	[4,5,6,2] x	[3,5,6,2,3] *	
		[6,2,4] x	[5,6,2,4] x	[4,5,6,2,4] *	
		[6,2,3] x	[5,6,2,3] x	[4,5,6,2,3]	
			[6,2,4,5] x	[5,6,2,4,5] *	
			[6,2,3,5] x	[5,6,2,3,5] *	
				[6,2,4,5,6] *	
				[6,2,3,5,6] *	

* denotes path cycles.

x means not prime paths.

! means path terminates

Check paths
- without 'x'; or
- with '*'

12 Prime Paths

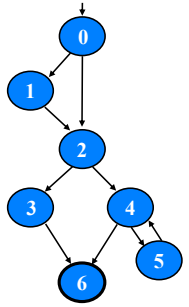
53 Simple Paths

Prime Path Example (2)

- This graph has 38 simple paths
- Only 9 *prime paths*

Homework: derive all the prime paths.

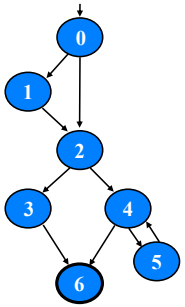
Prime Paths
[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]



Node Coverage

TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]



Node Coverage

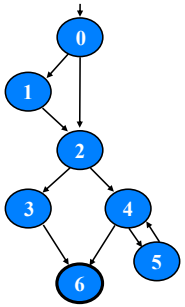
TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { [0,1], [0,2], [1,2], [2,3], [2,4], [3,6], [4,5], [4,6], [5,4] }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]



Node Coverage

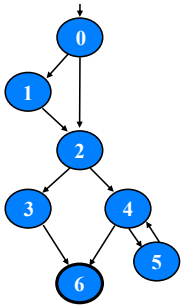
TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { [0,1], [0,2], [1,2], [2,3], [2,4], [3,6], [4,5], [4,6], [5,4] }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]



Edge-Pair Coverage

TR = { [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6],
[2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] }

Test Paths:

[0, 1, 2, 3, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 6]

Node Coverage

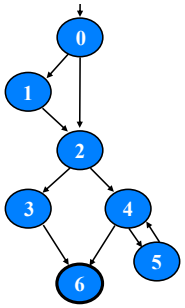
TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { [0,1], [0,2], [1,2], [2,3], [2,4], [3,6], [4,5], [4,6], [5,4] }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]



Edge-Pair Coverage

TR = { [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6],
[2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] }

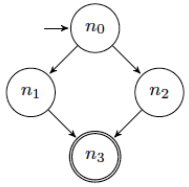
Test Paths:

[0, 1, 2, 3, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 6]

Complete Path Coverage

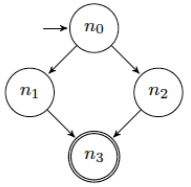
Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] ……

Prime Path Coverage vs. Complete Path Coverage



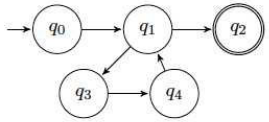
- Prime paths:
- $\text{path}(t_1) =$
- $\text{path}(t_2) =$
- $T_1 = \{t_1, t_2\}$ satisfies both PPC and CPC.

Prime Path Coverage vs. Complete Path Coverage



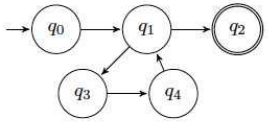
- Prime paths: [n0, n1, n3], [n0, n2, n3]
- $\text{path}(t_1) =$ [n0, n1, n3]
- $\text{path}(t_2) =$ [n0, n2, n3]
- $T_1 = \{t_1, t_2\}$ satisfies both PPC and CPC.

Prime Path Coverage vs. Complete Path Coverage



- Prime paths:
- $\text{path}(t_3) =$
- $\text{path}(t_4) =$
- $T_1 = \{t_3, t_4\}$ satisfies both PPC but not CPC.

Prime Path Coverage vs. Complete Path Coverage



- Prime paths: [q0, q1, q2], [q0, q1, q3, q4], [q3, q4, q1, q2], [q1, q3, q4, q1], [q3, q4, q1, q3], [q4, q1, q3, q4]
- $\text{path}(t_3) =$ [q0, q1, q2]
- $\text{path}(t_4) =$ [q0, q1, q3, q4, q1, q3, q4, q1, q2]
- $T_1 = \{t_3, t_4\}$ satisfies both PPC but not CPC.

Data Flow Coverage Criteria

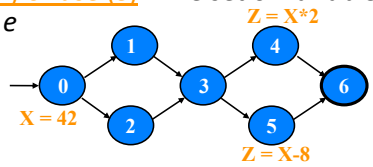
Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Data Flow Criteria

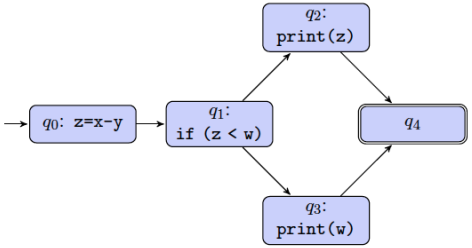
Goal: Try to ensure that values are computed and used correctly

- Definition (or *def* for short) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed
- def (*n*) or def (*e*) : The set of variables that are defined by node *n* or edge *e*
- use (*n*) or use (*e*) : The set of variables that are used by node *n* or edge *e*

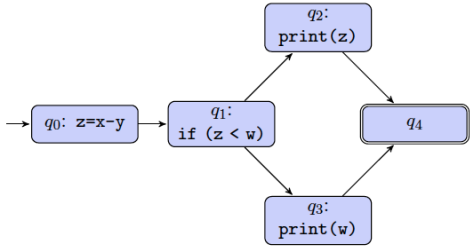


Defs: def (0) = {X}
def (4) = {Z}
def (5) = {Z}
Uses: use (4) = {X}
use (5) = {X}

Another Example



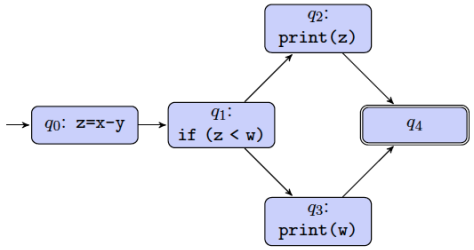
Another Example



def(q0) =
def(q1) =
def(q2) =
def(q3) =
def(q4) =

use(q0) =
use(q1) =
use(q2) =
use(q3) =
use(q4) =

Another Example



def(q0) = {z}
def(q1) = {}
def(q2) = {}
def(q3) = {}
def(q4) = {}

use(q0) = {x, y}
use(q1) = {z, w}
use(q2) = {z}
use(q3) = {w}
use(q4) = {}

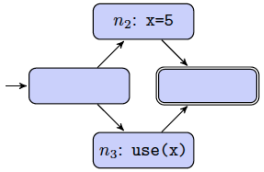
Reach

A def of a variable may or may not reach a particular use.

Reach – Case One

A def of a variable may or may not reach a particular use.

- **No** path from def to use
- Real example code?

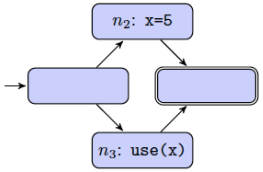


Reach – Case One

A def of a variable may or may not reach a particular use.

- **No** path from def to use
- Real example code?

```
#!/usr/bin/env bash
# file: t.sh
FIRST_ARG="${1}" # First Argument.

if [[ "${FIRST_ARG}" == 1 ]]
then
    echo "${x}" #node 3
else
    x=5          #node 2
fi
echo "${x}"
```



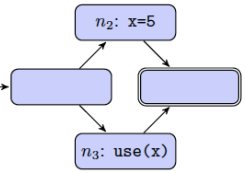
Reach – Case One

A def of a variable may or may not reach a particular use.

- **No** path from def to use
- Real example code?

```
#!/usr/bin/env bash
# file: t.sh
FIRST_ARG="${1}" # First Argument.

if [[ "${FIRST_ARG}" == 1 ]]
then
    echo "${x}" #node 3
else
    x=5          #node 2
fi
echo "${x}"
```

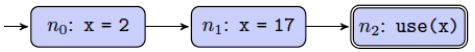
```
int x = 0;
if (…) {
  x = 5;
} else {
  print(x);
}
```



Reach – Case Two

A def of a variable may or may not reach a particular use.

- The variable is changed by another def before it reaches the use.
- Example:
  - 'x' is redefined in n1.



DU-pair

- DU pair: A pair of locations (l_i, l_j) such that a variable v is defined at l_i and used at l_j.

$$\text{du-pair} \left[\begin{array}{ll} \texttt{x = 5} & \texttt{// def(x)} \\ \vdots & \\ \texttt{print(x)} & \texttt{// use(x)} \end{array} \right.$$

Def-clear and Reach

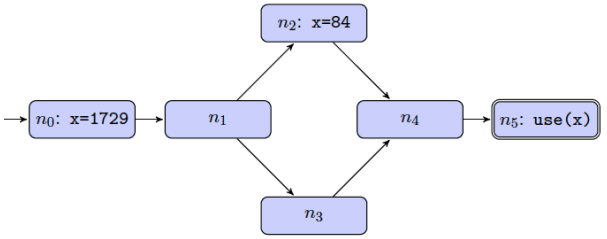
- Def-clear: A path p from l_i to l_j is ***def-clear*** with respect to variable v if for every node n_k and every edge e_k on p from l_i to l_j, where k ≠ i and k ≠ j, v is not in def(n_k) or in def(e_k).

Def-clear and Reach

- Def-clear: A path p from l_i to l_j is ***def-clear*** with respect to variable v if for every node n_k and every edge e_k on p from l_i to l_j, where k ≠ i and k ≠ j, v is not in def(n_k) or in def(e_k).

- Reach: If there is a def-clear path from l_i to l_j with respect to v, the def of v at l_i reaches the use at l_j.

Does the def at n0 reach the use at n5?

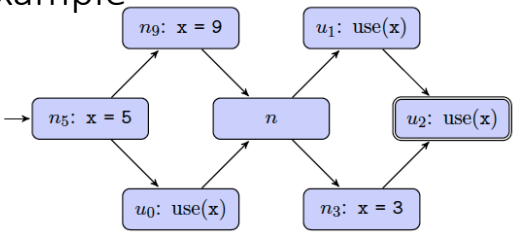


DU Paths

- du-path : A simple path that is def-clear with respect to v from a def of v to a use of v
  - *associated with a variable*
  - *simple (otherwise there are too many)*
  - *may be any number of uses in a du-path*

- ***Def-pair set*** du (n_i, n_j, v) – the set of du-paths from n_i to n_j
- ***Def-path set*** du (n_i, v) – the set of du-paths that start at n_i

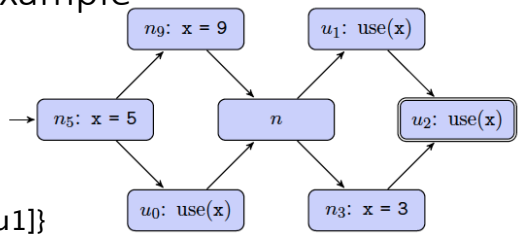
Def-pair & Def-path Example



- du(n5, u1, x) =
- du(n5,x) =

- du(n3,x) =

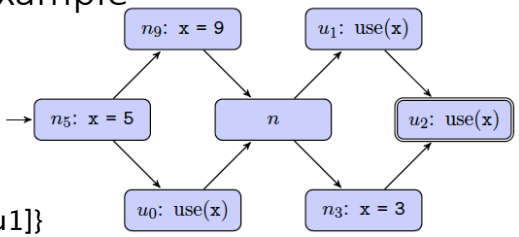
Def-pair & Def-path Example



- du(n5, u1, x) = {[n5, u0, n, u1]}
- du(n5,x) =

- du(n3,x) =

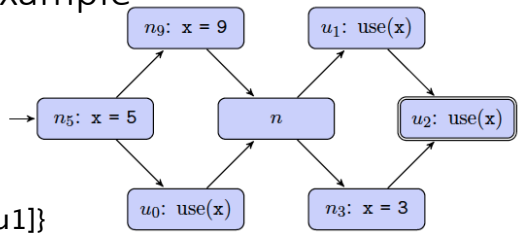
Def-pair & Def-path Example



- du(n5, u1, x) = {[n5, u0, n, u1]}
- du(n5,x) = du(n5, u0, x) U du (n5, u1, x) U du (n5, u2, x)

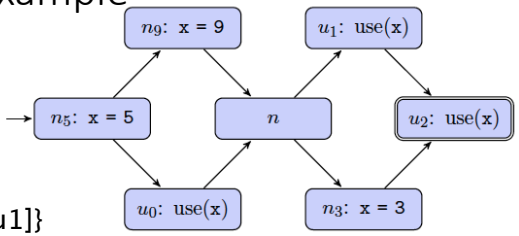
- du(n3,x) =

Def-pair & Def-path Example



- du(n5, u1, x) = {[n5, u0, n, u1]}
- du(n5,x) = du(n5, u0, x) U du (n5, u1, x) U du (n5, u2, x)
  = {[n5, u0], [n5, u0, n, u1], [n5, u0, n, u1, u2]}
- du(n3,x) =

Def-pair & Def-path Example

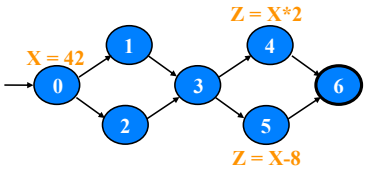


- du(n5, u1, x) = {[n5, u0, n, u1]}
- du(n5,x) = du(n5, u0, x) U du (n5, u1, x) U du (n5, u2, x)
  = {[n5, u0], [n5, u0, n, u1], [n5, u0, n, u1, u2]}
- du(n3,x) = du (n3, u2, x)= {[n3, u2]}

Data Flow Test Criteria

- First, we make sure every def reaches a use:
  - All-Defs Coverage (ADC) : For each set of du-paths S = du (n, v), TR contains at least one path d in S.
- Then we make sure that every def reaches all possible uses:
  - All-Uses Coverage (AUC) : For each set of du-paths to uses S = du (ni, nj, v), TR contains at least one path d in S.
- Finally, we cover all the du-paths between defs and uses:
  - All-du-Paths Coverage (ADUPC) : For each set S = du (ni, nj, v), TR contains every path d in S.

ADC, AUC, ADUPC Example

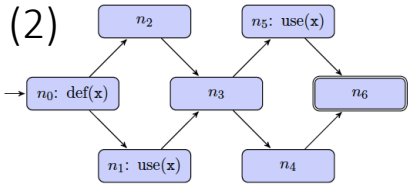


ADC for X
[0, 1, 3, 4]

AUC for X
[0, 1, 3, 4]
[0, 1, 3, 5]

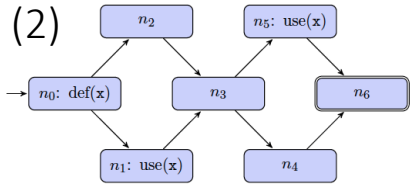
All-du-paths for X
[0, 1, 3, 4]
[0, 2, 3, 4]
[0, 1, 3, 5]
[0, 2, 3, 5]

ADC, AUC, ADUPC Example (2)



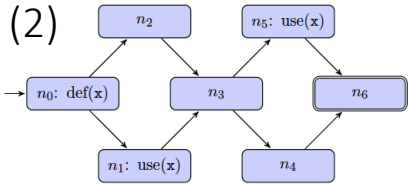
- ADC requires:
- AUC requires:
- ADUPC requires:

ADC, AUC, ADUPC Example (2)



- ADC requires: {[n0,n1]}
- AUC requires:
- ADUPC requires:

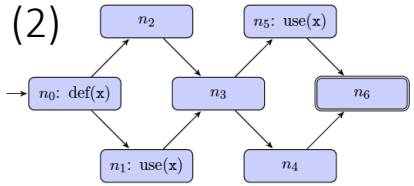
ADC, AUC, ADUPC Example (2)



- ADC requires: {[n0,n1]}
- AUC requires: {[n0,n1], [n0, n2,n3,n5]}
- ADUPC requires:

25

ADC, AUC, ADUPC Example (2)



- ADC requires: {[n0,n1]}
- AUC requires: {[n0,n1], [n0, n2,n3,n5]}
- ADUPC requires: {[n0,n1], [n0,n2,n3,n5], [n0,n1,n3,n5]}

List all du-paths even if they are a subpath of another.

26

ADC, AUC, ADUPC Example (3)

ADC requires: {

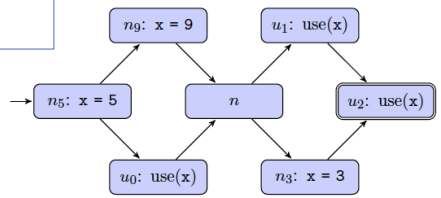
}

AUC requires {

}

ADUPC requires {

}



ADC, AUC, ADUPC Example (3)

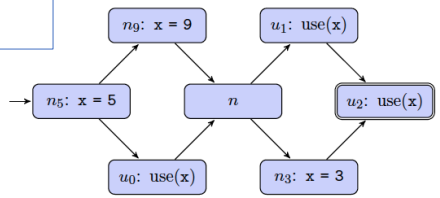
ADC requires: {
[n5,u0],
[n9, n, u1],
[n3, u2]
}

AUC requires {

}

ADUPC requires {

}

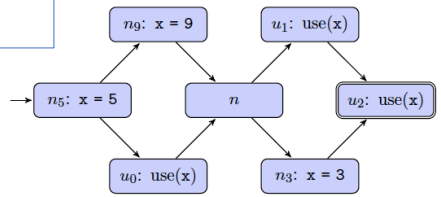


ADC, AUC, ADUPC Example (3)

ADC requires: {
[n5,u0],
[n9, n, u1],
[n3, u2]
}

AUC requires {
[n5,u0],
[n5,u0, n, u1],
[n5,u0, n, u1, u2],
[n9, n, u1],
[n9, n, u1, u2],
[n3, u2]
}

ADUPC requires {
}

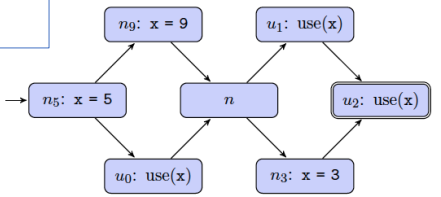


ADC, AUC, ADUPC Example (3)

ADC requires: {
[n5,u0],
[n9, n, u1],
[n3, u2]
}

AUC requires {
[n5,u0],
[n5,u0, n, u1],
[n5,u0, n, u1, u2],
[n9, n, u1],
[n9, n, u1, u2],
[n3, u2]
}

ADUPC requires {
[n5,u0],
[n5,u0, n, u1],
[n5,u0, n, u1, u2],
[n9, n, u1],
[n9, n, u1, u2],
[n3, u2]
}



Touring DU-Paths

A test path ***p*** ***du-tours*** subpath ***d*** with respect to ***v*** if ***p*** tours ***d*** and the subpath taken is def-clear with respect to ***v***

Three criteria - assume best effort touring

Use every def

Get to every use

Follow all du-paths

Definitions and Uses in Source Code

- **Definitions.** Here are some statements which correspond to definitions.
  - x = 5: x occurs on the left-hand side of an assignment statement;
  - foo(T x) { ...}: implicit definition for x at the start of a method;
  - bar(x): during a call to bar, x might be defined if x is a C++ reference parameter.
  - subsumed by others: x is an input to the program.
- **Uses.** The book lists a number of cases of uses, but it boils down to “x occurs in an expression that the program evaluates.''
  - E.g., RHS of an assignment, as part of a method parameter, or in a conditional

Basic blocks and defs/uses

- Basic blocks can rule out some definitions and uses as irrelevant.
- **Defs**: consider the last definition of a variable in a basic block. (If we're not sure whether x and y are aliased, keep both of them.)
- **Uses**: consider only uses that aren't dominated by a definition of the same variable in the same basic block, e.g. y = 5; use(y) is not interesting.

Compiler tidbit

- In a compiler, we use intermediate representations to simplify expressions, including definitions and uses.
- For instance, we would simplify:
  - x = foo(y + 1, z * 2)
- To:
  - a = y+1
  - b = z*2
  - x = foo(a,b)

- Answer questions (a)-(f) for the graph defined by the following sets:
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}
  - def(0)=def(3)=use(5)=use(7) = {X}
- Also consider the following test paths:
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

(a) Draw the graph.

- Answer questions (a)-(f) for the graph defined by the following sets:
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}
  - def(0)=def(3)=use(5)=use(7) = {X}
- Also consider the following test paths:
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

(b) List all of the du-paths with respect to x. (Note: Include all du-paths, even those that are subpaths of some other du-paths).

- Answer questions (a)-(f) for the graph defined by the following sets:
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}
  - def(0)=def(3)=use(5)=use(7) = {X}
- Also consider the following test paths:
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(b) List all of the du-paths with respect to x. (Note: Include all du-paths, even those that are subpaths of some other du-paths).

- Answer questions (a)-(f) for the graph defined by the following sets:
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}
  - def(0)=def(3)=use(5)=use(7) = {X}
- Also consider the following test paths:
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(c) For each test path, determine which du-paths that test path du-tours. Consider direct touring only.
Hint: A table is a convenient format for describing the relationship.

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(c) For each test path, determine which du-paths that test path du-tours. Consider direct touring only.
Hint: A table is a convenient format for describing the relationship.

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(d) List a minimal test set that satisfies *all-defs* coverage with respect to x (Direct tours only). Use the given test paths.

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)}
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(d) List a minimal test set that satisfies *all-defs* coverage with respect to x (Direct tours only). Use the given test paths.

- {t1, t4} or {t1, t5} or {t3, t4} or {t3, t5}

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)}
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(e) List a minimal test set that satisfies *all-uses* coverage with respect to x. (Direct tours only). Use the given test paths.

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)}
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(e) List a minimal test set that satisfies *all-uses* coverage with respect to x. (Direct tours only). Use the given test paths.

- {t1, t3, t5}

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)}
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

(f) List a minimal test set that satisfies *all-du-paths* coverage with respect to x. (Direct tours only). Use the given test paths.

- Answer questions (a)-(f) for
  - N ={0, 1, 2, 3, 4, 5, 6, 7}
  - N0 = {0}
  - Nf = {7}
  - E = {(0,1), (1, 2), (1, 7), (2, 3)}
  - def(0)=def(3)=use(5)=use(7)
- Also consider the following t
  - t1 = [0,1,7]
  - t2 = [0,1,2,4,6,1,7]
  - t3 = [0,1,2,4,5,6,1,7]
  - t4 = [0,1,2,3,2,4,6,1,7]
  - t5 = [0,1,2,3,2,3,2,4,5,6,1,7]
  - t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]

Test Path	Du-tour directly
t1 = [0,1,7]	ii
t2 = [0,1,2,4,6,1,7]	/
t3 = [0,1,2,4,5,6,1,7]	i
t4 = [0,1,2,3,2,4,6,1,7]	iv
t5 = [0,1,2,3,2,3,2,4,5,6,1,7]	iii, v
t6 = [0,1,2,3,2,4,6,1,2,4,5,6,1,7]	/

- (b) du (0, 5, x): i: [0,1,2,4,5]
  du (0, 7, x): ii: [0,1,7]
  du (3, 5, x): iii: [3,2,4,5]
  du (3, 7, x): iv: [3,2,4,6,1,7]
  v: [3,2,4,5,6,1,7]

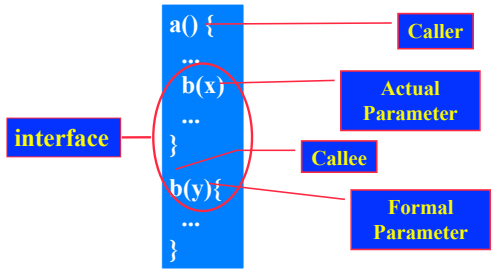
(f) List a minimal test set that satisfies *all-du-paths* coverage with respect to x. (Direct tours only). Use the given test paths.
- {t1, t3, t4, t5}

Inter-procedural Data Flow

- Data flow couplings among units are more complicated than control flow couplings.
  - When values are passed, they "change names".
  - Finding which uses a def can reach is very difficult.
    - polymorphism/virtual functions

- **Caller : A unit that invokes another unit**
- **Callee : The unit that is called**
- **Callsite : Statement or node where the call appears**
- **Actual parameter : Variable in the caller**
- **Formal parameter : Variable in the callee**

Example Call Site



- Applying data flow criteria to def-use pairs between units is too expensive.
- Too many possibilities
- But this is integration testing, and we really only care about the interface …

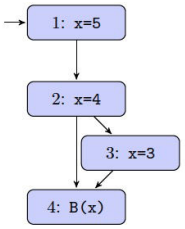
Inter-procedural DU Pairs

- If we focus on the interface, then we just need to consider the last definitions of variables before calls and returns and first uses inside units and after calls.

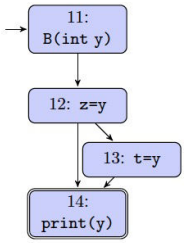
- Last-def : The set of nodes that define a variable x and has a def-clear path from the node through a callsite to a use in the other unit
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value

- First-use : The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.

Last-defs & First-uses

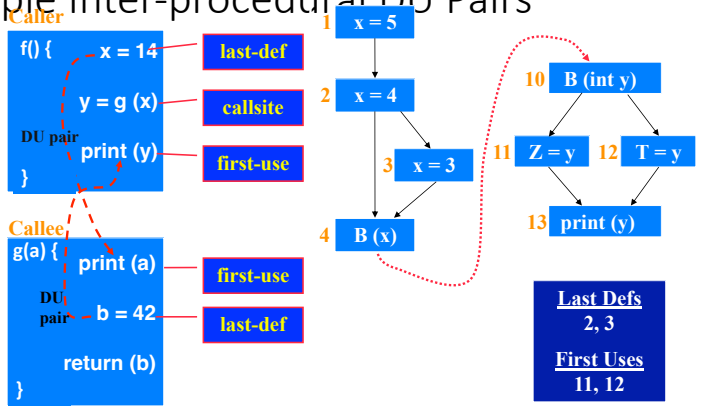


last-defs are 2, 3



the first-use is 12

Example Inter-procedural DU Pairs



Example – Quadratic

```
1 // Program to compute the quadratic root for
two numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv[1]);
16       Y = Integer.parseInt (argv[2]);
17       Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {
21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
25       ok = Root (X, Y, Z);
26       if (ok)
27         System.out.println
28            ("Quadratic: " + Root1 + Root2);
29       else
30          System.out.println ("No Solution.");
31    }
32
33 // Three positive integers, finds quadratic root
34    private static boolean Root (int A, int B, int C)
35    {
36       float D;
37       boolean Result;
38       D = (float) Math.pow ((double)B,
             (double2-4.0)*A*C );
39       if (D < 0.0)
40       {
41          Result = false;
42          return (Result);
43       }
44       Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
45       Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
46       Result = true;
47       return (Result);
48    } //End method Root
49
50 } // End class Quadratic
```

Example – Quadratic

```
1 // Program to compute the quadratic root for
two numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv[1]);
16       Y = Integer.parseInt (argv[2]);
17       Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {
21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
25       ok = Root (X, Y, Z);
26       if (ok)
27         System.out.println
28            ("Quadratic: " + Root1 + Root2);
29       else
30          System.out.println ("No Solution.");
31    }
32
33 // Three positive integers, finds quadratic root
34    private static boolean Root (int A, int B, int C)
35    {
36       float D;
37       boolean Result;
38       D = (float) Math.pow ((double)B,
             (double2-4.0)*A*C );
39       if (D < 0.0)
40       {
41          Result = false;
42          return (Result);
43       }
44       Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
45       Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
46       Result = true;
47       return (Result);
48    } //End method Root
49
50 } // End class Quadratic
```

first-use

last-defs

first-uses

last-defs

Example – Quadratic

Pairs of locations: method name, variable name, statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

```
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15      X = Integer.parseInt (argv[1]);
16      Y = Integer.parseInt (argv[2]);
17      Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {
21      X = 10;
22      Y = 9;
23      Z = 12;
24    }
```

last-defs

```
25        ok = Root (X, Y, Z);
26        if (ok)
27          System.out.println
28              ("Quadratic: " + Root1 + Root2);
29        else
30          System.out.println ("No Solution.");
31  }
32
33 // Three positive integers, finds quadratic root
34  private static boolean Root (int A, int B, int C)
35  {
36    float D;
37    boolean Result;
38    D = (float) Math.pow ((double)B,
          (double2-4.0)*A*C );
39    if (D < 0.0)
40    {
41      Result = false;
42      return (Result);
43    }
44    Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
45    Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
46    Result = true;
47    return (Result);
48  } / /End method Root
49
50  } // End class Quadratic
```

first-uses

Example – Quadratic

Pairs of locations: method name, variable name, statement

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

first-use

last-defs

```
25        ok = Root (X, Y, Z);
26        if (ok)
27          System.out.println
28              ("Quadratic: " + Root1 + Root2);
29        else
30          System.out.println ("No Solution.");
31  }
32
33 // Three positive integers, finds quadratic root
34  private static boolean Root (int A, int B, int C)
35  {
36    float D;
37    boolean Result;
38    D = (float) Math.pow ((double)B,
          (double2-4.0)*A*C );
39    if (D < 0.0)
40    {
41      Result = false;
42      return (Result);
43    }
44    Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
45    Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
46    Result = true;
47    return (Result);
48  } / /End method Root
49
50  } // End class Quadratic
```

Quadratic – Coupling DU-pairs

Pairs of locations: method name, variable name, statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

Strengths and Weaknesses of Graph Coverage:

- Must create graph (at least conceptually)
- Node coverage is usually easy, but cycles make it hard to get good coverage in general.
- Incomplete node or edge coverage points to deficiencies in a test set, though high node or edge coverage does not mean that you have done well in testing

Automated Testing & Bug Detection Tools

Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from Prof. Patrick Lam's, Prof. Lin Tan's and Jakub Kuderski's.

Dynamic Analysis Tools

Automated Test Case Generation

- Test cases can be generated automatically, but
- How to generate interesting test inputs
  - Black box – truly random, common/interesting test patterns
  - Grey box – guided by coverage, new inputs should cover new code paths
  - White box – symbolic reasoning about program code, new inputs are guaranteed to cover new paths

```
void crashme (char* s) {
  if (s[0] == 'b')
    if (s[1] == 'a')
      if (s[2] == 'd')
        if (s[3] == '!')
          abort();
}
```


Automated Test Case Generation

- Test cases can be generated automatically, but
- How to generate interesting test inputs
  - Black box – truly random, common/interesting test patterns
  - Grey box – guided by coverage, new inputs should cover new code paths
  - White box – symbolic reasoning about program code, new inputs are guaranteed to cover new paths
- How to generate automatic/generic **test oracles/expected test results**
  - do not crash! (easy to check, but often not informative/soon enough)
  - do not misuse memory (buffer overflow, use-after-free, …)
  - no data races
  - user written assertions!
  - domain specific specifications and oracles

How to detect bad memory accesses

- Will this program crash?
  - depends on memory layout
  - might crash
- Unpredictable behavior makes it difficult to test and diagnose the problem.
  - Big issue for automatic testing!
  - Nondeterministic.

```
#include <stdlib.h>

int main() {
  int* x = malloc(10 * sizeof(int));
  int* y = malloc(5 * sizeof(int));
  *y = *(x + 12);
  free(x);
  free(y);
  return 0;
}
```


Valgrind

- An instrumentation framework for dynamic analysis tools
- Interprets a program on “synthetic” CPU
- Analysis tools inspect CPU instructions and insert additional checks at very low level
- Execution of every instruction is interpreted in a sandbox and error report is produced when suspicious behavior is detected
- Pros:
  - very detailed analysis
  - easy to use -- no need to change build system much
- Cons: 10x or more slowdown in performance

Valgrind -- Memcheck

- Memcheck is a memory error detector. It helps you make your programs, particularly those written in C and C++, more correct.
  - illegal read / illegal write errors
  - use of uninitialized values
  - illegal frees
  - overlapping source and destination blocks
  - memory leak detection
- https://valgrind.org/

```
$ gcc -g -O0 <your program> -o a.out
$ valgrind -tool=memcheck ./a.out
```



```
#include <stdlib.h>

int main() {
  int* x = malloc(10 * sizeof(int));
  int* y = malloc(5 * sizeof(int));
  *y = *(x + 12);
  free(x);
  free(y);
  return 0;
}
```

```
[cnsun@ecelinux1 ~/temp]$ gcc -g -O0 t.c -o t.exe
[cnsun@ecelinux1 ~/temp]$ valgrind --tool=memcheck ./t.exe
==13729== Memcheck, a memory error detector
……
==13729== Invalid read of size 4
==13729==    at 0x4005A5: main (t.c:6)
==13729==  Address 0x5205070 is 8 bytes after a block of size 40 alloc'd
==13729==    at 0x4C29EA3: malloc (vg_replace_malloc.c:309)
==13729==    by 0x40058E: main (t.c:4)
==13729==
==13729==
==13729== HEAP SUMMARY:
==13729==     in use at exit: 0 bytes in 0 blocks
==13729==   total heap usage: 2 allocs, 2 frees, 60 bytes allocated
==13729==
==13729== All heap blocks were freed -- no leaks are possible
==13729==
==13729== For counts of detected and suppressed errors, rerun with: -v
==13729== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Memcheck performs bounds checks and use-after-free checks for heap arrays. It also finds uses of uninitialized values created by heap or stack allocations.

But it does not perform bounds checking for stack or global arrays.

The tool ***exp-sgcheck*** might help, but is experimental

https://valgrind.org/docs/manual/sg-manual.html

Valgrind – False Negative – Global OOB

```
int a[100] = {-1};
int main(int argc, char** argv) {
  int b = a[argc + 100];
  return b;
}
```

```
cnsun@host|:~/temp$ gcc -g -O0 global_oob.c
cnsun@host|:~/temp$ valgrind --tool=memcheck ./a.out
==26458== Memcheck, a memory error detector
==26458== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==26458== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==26458== Command: ./a.out
==26458==
==26458==
==26458== HEAP SUMMARY:
==26458==     in use at exit: 0 bytes in 0 blocks
==26458==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==26458==
==26458== All heap blocks were freed -- no leaks are possible
==26458==
==26458== For counts of detected and suppressed errors, rerun with: -v
==26458== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```


Valgrind – False Negative -- Stack OOB

```
int main(int argc, char **argv) {
  int stack_array[100] = {-1};
  int another[100] = {-1};
  stack_array[1] = 0;
  return stack_array[argc + 100];
}
```

```
$ gcc -g -O0 stack_oob.c -o a.out && valgrind --tool=memcheck ./a.out
==26626== Memcheck, a memory error detector
==26626== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==26626== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==26626== Command: ./a.out
==26626==
==26626==
==26626== HEAP SUMMARY:
==26626==     in use at exit: 0 bytes in 0 blocks
==26626==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==26626==
==26626== All heap blocks were freed -- no leaks are possible
==26626==
==26626== For counts of detected and suppressed errors, rerun with: -v
==26626== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Sanitizers – Address Sanitizer (Asan)

- *Compile-time* instrumentation
- Supported by Clang and GCC

- Out-of-bounds accesses to heap, **stack**, and **globals**
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks

Asan – Implementation

- Valgrind **dynamically** emulates a CPU and checks, at each memory access, whether that access is legitimate or not.
- Asan rewrites relevant memory access at **compile-time** to call a checking library. (faster, 2x slowdown)

Original
```
*address = ...;
```

Pseudocode by Asan (or even Valgrind)
```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```


Asan – Implementation

- Valgrind **dynamically** emulates a CPU and checks, at each memory access, whether that access is legitimate or not.
- Asan rewrites relevant memory access at **compile-time** to call a checking library. (faster, 2x slowdown)

Original
```
*address = ...;
```

Pseudocode by Asan (or even Valgrind)
```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

Similar for load/memory read
```
... = *address;
```


Asan – Shadow Memory

- Virtual memory is divided into two disjoint classes
  - Mem is the normal application memory
  - Shadow is memory that keeps track of meta-data (information) about main memory. For each byte addr of Mem, Shadow contains a descriptor Shadow[addr]
- Poisoning a byte addr of Mem means writing a special value to corresponding place in Shadow.

Asan – Shadow Me

Pseudocode by Asan (or even Valgrind)
```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

- Virtual memory is divided into two disjoint classes
  - Mem is the normal application memory
  - Shadow is memory that keeps track of meta-data (information) about main memory. For each byte addr of Mem, Shadow contains a descriptor Shadow[addr]
- Poisoning a byte addr of Mem means writing a special value to corresponding place in Shadow.

Asan – Shadow Me

Pseudocode by Asan (or even Valgrind)

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

- Virtual memory is divided into two disjoint classes
  - Mem is the normal application memory
  - Shadow is memory that keeps track of meta-data (information) about main memory. For each byte addr of Mem, Shadow contains a descriptor Shadow[addr]
- Poisoning a byte addr of Mem means writing a special value to corresponding place in Shadow.

```
shadow_address = MemToShadow(address)
if (ShadowIsPoisoned(shadow_address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```


Asan – Shadow Me

Pseudocode by Asan (or even Valgrind)

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

- Virtual memory is divided into two disjoint classes
  - Mem is the normal application memory
  - Shadow is memory that keeps track of meta-data (information) about main memory. For each byte addr of Mem, Shadow contains a descriptor Shadow[addr]
- Poisoning a byte addr of Mem means writing a special value to corresponding place in Shadow.

Valgrind has shadow memory too.

```
shadow_address = MemToShadow(address)
if (ShadowIsPoisoned(shadow_address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```


Asan Example

```
int a[100] = {-1};
int main(int argc, char** argv) {
  int b = a[argc + 100];
  return b;
}
```

```
$ clang-9 -fsanitize=address -g -O0 global_oob.c && ./a.out
=================================================================
==2335==ERROR: AddressSanitizer: global-buffer-overflow on address 0x0000006f3cf4 at pc
0x0000004c20e4 bp 0x7ffe3d2b5640 sp 0x7ffe3d2b5638
READ of size 4 at 0x0000006f3cf4 thread T0
    #0 0x4c20e3 in main /home/cnsun/temp/global_oob.c:3:11
    #1 0x7f3747449b96 in __libc_start_main /build/glibc-OTsEL5/glibc-
2.27/csu/../csu/libc-start.c:310
    #2 0x41aaa9 in _start (/home/cnsun/temp/a.out+0x41aaa9)

0x0000006f3cf4 is located 4 bytes to the right of global variable 'a' defined in
'global_oob.c:1:5' (0x6f3b60) of size 400
SUMMARY: AddressSanitizer: global-buffer-overflow /home/cnsun/temp/global_oob.c:3:11 in
main
Shadow bytes around the buggy address:
```


Asan Example – Global OOB

```
int a[100] = {-1};
int main(int argc, char** argv) {
  int b = a[argc + 100];
  return b;
}
```

```
$ clang-9 -fsanitize=address -g -O0 global_oob.c && ./a.out
=================================================================
==2335==ERROR: AddressSanitizer: global-buffer-overflow on address 0x0000006f3cf4 at pc
0x0000004c20e4 bp 0x7ffe3d2b5640 sp 0x7ffe3d2b5638
READ of size 4 at 0x0000006f3cf4 thread T0
    #0 0x4c20e3 in main /home/cnsun/temp/global_oob.c:3:11
    #1 0x7f3747449b96 in __libc_start_main /build/glibc-OTsEL5/glibc-
2.27/csu/../csu/libc-start.c:310
    #2 0x41aaa9 in _start (/home/cnsun/temp/a.out+0x41aaa9)

0x0000006f3cf4 is located 4 bytes to the right of global variable 'a' defined in
'global_oob.c:1:5' (0x6f3b60) of size 400
SUMMARY: AddressSanitizer: global-buffer-overflow /home/cnsun/temp/global_oob.c:3:11 in
main
Shadow bytes around the buggy address:
```

Asan Example – Stack OOB

```
int main(int argc, char **argv) {
  int stack_array[100] = {-1};
  int another[100] = {-1};
  stack_array[1] = 0;
  return stack_array[argc + 100];
}
```

```
$ clang-9 -g -O0 -fsanitize=address stack_oob.c && ./a.out
=================================================================
==2363==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffde0cc14b4 at pc 0x0000004c2422 bp 0x7ffde0cc12f0 sp 0x7ffde0cc12e8
READ of size 4 at 0x7ffde0cc14b4 thread T0
   #0 0x4c2421 in main /home/cnsun/temp/stack_oob.c:5:10
   #1 0x7fa627f01b96 in __libc_start_main /build/glibc-OTsEL5/glibc-2.27/csu/../csu/libc-start.c:310
   #2 0x41aaa9 in _start (/home/cnsun/temp/a.out+0x41aaa9)

Address 0x7ffde0cc14b4 is located in stack of thread T0 at offset 436 in frame
   #0 0x4c207f in main /home/cnsun/temp/stack_oob.c:1

  This frame has 2 object(s):
   [32, 432) 'stack_array' (line 2) <== Memory access at offset 436 overflows this variable
   [496, 896) 'another' (line 3)
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/cnsun/temp/stack_oob.c:5:10 in main
Shadow bytes around the buggy address:
```


Valgrind vs. Asan

	Valgrind	Asan
instrumentation	dynamic	static
slowdown	10x ~ 20x	2x
Heap OOB	yes	yes
Stack OOB	no	yes
Global OOB	no	yes
use after free	yes	yes
uninitialized memory read	yes	no
leaks	yes	yes

https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools

Other Sanitizers

- Memory sanitizer
  - For **uninitialized** memory reads
  - `clang -g -O0 -fsanitize=memory`
- Thread sanitizer
  - To detect data races
  - `clang -g -O0 -fsanitize=thread`

Other Sanitizers

- Undefined behavior sanitizer
  - To detect undefined behaviors, e.g., integer overflow
  - `clang -g -O0 -fsanitize=undefined`

```
#include <stdio.h>
#include <limits.h>
int main() {
  int input = INT_MAX - 5;
  printf("%d\n", input + 6);
  return 0;
}
```

```
$ clang-9 -g -O0 -fsanitize=undefined overflow.c && ./a.out
overflow.c:5:24: runtime error: signed integer overflow: 2147483642 + 6 cannot be represented in type 'int'
-2147483648
```

Automated Test-Case Generation

Automated Test-Case Generation

- Manual test case generation can be laborious and difficult:
  - Takes human time and effort
  - Requires understanding the tested code
- Alternative: Automated Test Case Generation
  - Making computer do the work

Fuzzing – Fuzz Testing

- Set of automated testing techniques that tries to identify abnormal program behaviors by evaluation how the tested program responds to various inputs.

> We didn't call it fuzzing back in the 1950s, but it was our standard practice to **test programs** by **inputting** decks of **punch cards taken from the trash.** (...) our random/trash decks often turned up **undesirable behavior**. Every programmer I knew (and there weren't many of us back then, so I knew a great proportion of them) used the trash-deck technique.
>
> Gerald M. Weinberg

Fuzzing – Fuzz Testing

- Set of automated testing techniques that tries to identify abnormal program behaviors by evaluation how the tested program responds to various inputs.
- Challenges:
  - Finding interesting inputs
  - Exploring whole system, not just individual tools or functions
  - Reducing the size of test cases
  - Reducing duplication – test cases may exercise the same parts of codebase

> We didn't call it fuzzing back in the 1950s, but it was our standard practice to **test programs** by **inputting** decks of **punch cards taken from the trash.** (...) our random/trash decks often turned up **undesirable behavior**. Every programmer I knew (and there weren't many of us back then, so I knew a great proportion of them) used the trash-deck technique.
>
> Gerald M. Weinberg

Naïve Fuzzers

- Black-box testing technique: does not try to reason about tested programs.
- Idea: feed random inputs in and monitor tested programs for abnormal behaviors.
- Pros:
  - Easy to implement
  - Fast
- Issues:
  - Relies on the 'luck' of random input
  - May run the same things over and over again
  - 'Shallow' program exploration

Naïve Fuzzers -- Example

```
1: while(true) {
2:   write a random string to "t.c"
3:   gcc -O3 t.c
4:   if gcc crashes when compiling "t.c" {
5:     a bug is found
6:   }
7: }
```

Fuzzers

- American Fuzzy Lop (AFL):
  - http://lcamtuf.coredump.cx/afl/
  - a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal program states in the targeted binary.
- libFuzzer
  - Similar to AFL, but fuzzes the program in the same process, thus faster.
  - A part of LLVM compiler infrastructure

Challenge #1: Feeding in inputs

- How to take random inputs and make the tested program consume it?
- AFL: passes input in a file (between processes).
- LibFuzzer: requires "fuzz targets – entry points that accept an array of bytes"

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

  - libFuzzer executes the fuzz target multiple times with different inputs (in-process fuzzing).

Challenge #2: Detecting abnormal behavior

- What can 'abnormal' mean? We need an oracle.
  - Crashes
  - Triggers a user-provided assertion failure
  - 'Hangs' -- execution takes longer than anticipated
  - Allocates too much memory
- Early crash detection – use sanitizers
  - asan, msan, tsan, ubsan, lsan, …

Challenge #3: Ensuring progression

- How can we know that fuzzing is exploring more program states over time?
- coverage
  - node coverage
  - edge coverage
  - data flow coverage
  - ……

Fuzzing in a nutshell

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S

$T_{\text{✗}} = \emptyset$
$T = S$
if $T = \emptyset$ **then**
    add empty file to T
end if
repeat
    $t = \text{CHOOSENEXT}(T)$
    $p = \text{ASSIGNENERGY}(t)$
    for i from 1 to p **do**
        $t' = \text{MUTATE\_INPUT}(t)$
        if t' crashes **then**
            add t' to $T_{\text{✗}}$
        else if $\text{ISINTERESTING}(t')$ **then**
            add t' to T
        end if
    end for
until *timeout* reached or *abort*-signal

Output: Crashing Inputs $T_{\text{✗}}$

Marcel Böhme, Van-Thuan Pham, Abhik Roychoudhury:
Coverage-Based Greybox Fuzzing as Markov Chain. IEEE Trans. Software Eng. 45(5): 489-506 (2019)

Challenge #4: Producing interesting inputs

- Fuzzers start at a provided test-case and keep mutating it.
- Examples of mutations:
  - Bit flipping: single bit, multiple bits at a time
  - Byte flips, byte swaps, byte rotates
  - Simple arithmetic: treating groups of bytes as numbers and adding values from predefined ranges: e.g., -128 to +128
  - Known interesting integers: e.g., -1, 0, 256, MAX_INT, MAX_INT - 1, MIN_INT
  - Combining multiple test-cases together

Fuzzing in a nutshell

```
void crashme (char* s) {
  if (s[0] == 'b')
    if (s[1] == 'a')
      if (s[2] == 'd')
        if (s[3] == '!')
          abort();
}
```

Marcel Böhme, Van-Thuan Pham, Abhik Roychoudhury:
Coverage-Based Greybox Fuzzing as Markov Chain. IEEE Trans. Software Eng. 45(5): 489-506 (2019)

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S
1: $T_{\text{✗}} = \emptyset$
2: $T = S$
3: **if** $T = \emptyset$ **then**
4: add empty file to T
5: **end if**
6: **repeat**
7: $t = \text{CHOOSENEXT}(T)$
8: $p = \text{ASSIGNENERGY}(t)$
9: **for** i from 1 to p **do**
10: $t' = \text{MUTATE\_INPUT}(t)$
11: **if** t' crashes **then**
12: add t' to $T_{\text{✗}}$
13: **else if** $\text{ISINTERESTING}(t')$ **then**
14: add t' to T
15: **end if**
16: **end for**
17: **until** *timeout* reached or *abort*-signal
Output: Crashing Inputs $T_{\text{✗}}$

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S
1: $T_{\text{✗}} = \emptyset$
2: $T = S$
3: **if** $T = \emptyset$ **then**
4: add empty file to T
5: **end if**
6: **repeat**
7: $t = \text{CHOOSENEXT}(T)$
8: $p = \text{ASSIGNENERGY}(t)$
9: **for** i from 1 to p **do**
10: $t' = \text{MUTATE\_INPUT}(t)$
11: **if** t' crashes **then**
12: add t' to $T_{\text{✗}}$
13: **else if** $\text{ISINTERESTING}(t')$ **then**
14: add t' to T
15: **end if**
16: **end for**
17: **until** *timeout* reached or *abort*-signal
Output: Crashing Inputs $T_{\text{✗}}$

•George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, Michael Hicks:
Evaluating Fuzz Testing. ACM Conference on Computer and Communications Security 2018: 2123-2138

Core fuzzing algorithm:

```
corpus ← initSeedCorpus()
queue ← ∅
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
    end if
end while
```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Fuzzing has been an active research area.

Static Analysis Tools

Chengnian Sun
cnsun@

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Apple's 'goto fail' Bug

Apple's 'goto fail' tells us nothing good about Cupertino's software delivery process

The fact that Apple's infamous SSL validation bug actually got out into the real world is pretty terrifying.

By Matt Baxter-Reynolds for Post-PC Developments | March 19, 2014 -- 10:00 GMT (03:00 PDT) | Topic: Security

There's been a lot of press about Apple's "goto fail." To catch you up, this was a bug that caused the proper validation of SSL certificates OS X and iOS to fail.

The result was that it was easier to trick any iOS and Mac devices to accept invalid certificates, and to present them as valid to the user.

Most of the analysis focused on the use of C's *goto* statement, particularly around whether

MORE FROM MATT BAXTER-REY

Cloud
Should m
business
Microsoft
tools?

Mobility
LG G Wat
The simpl

https://www.zdnet.com/article/apples-goto-fail-tells-us-nothing-good-about-cupertinos-software-delivery-process/


```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    // code ommitted for brevity...

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,            /* plaintext */
                       dataToSignLen,         /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Oops...

Never gets called (but needed to be)...

Despite the name, always returns "it's OK!!!"

https://web.nvd.nist.gov/view/vuln/detail? vulnId=CVE-2014-1266
https://nakedsecurity.sophos.com/2014/02/24/anatomy- of-a-goto-fail-apples-ssl-bug-explai

How to Detect

- Compiler Warning
  - -Wunreachable-code
- PC-Lint:warning 539: Did not expect positive indentation
- PVS-Studio:V640: Logic does not match formatting

https://web.nvd.nist.gov/view/vuln/detail? vulnId=CVE-2014-1266
https://nakedsecurity.sophos.com/2014/02/24/anatomy- of-a-goto-fail-apples-ssl-bug-explai

How to Avoid?

- Do not use goto
- Use '{' '}' for if-statement
- Format your code

Apple’s ‘goto fail’ Bug

Apple's 'goto fail' tells us nothing good about Cupertino's software delivery process

The fact that Apple's infamous SSL validation bug actually got out into the real world is pretty terrifying.

By Matt Baxter-Reynolds for Post-PC Developments | March 19, 2014 -- 10:00 GMT (03:00 PDT) | Topic: Security

There's been a lot of press about Apple's "goto fail." To catch you up, this was a bug that caused the proper validation of SSL certificates OS X and iOS to fail.

The result was that it was easier to trick any iOS and Mac devices to accept invalid certificates, and to present them as valid to the user.

Most of the analysis focused on the use of C's *goto* statement, particularly around whether

MORE FROM MATT BAXTER-REY

Cloud
Should m
business
Microsoft
tools?

Mobility
LG G Wat
The simpl

https://www.zdnet.com/article/apples-goto-fail-tells-us-nothing-good-about-cupertinos-software-delivery-process/

Static Analysis

- Static analysis refers to the examination of a piece of software without executing it.
- In the world of security, it refers to discovering security related bugs in software without actually running the software.
- Static code analysis is a white box method of testing, meaning that the tester has access to the underlying framework, design, and structure of the software.
- The process typically includes examining the code structure, studying the various data and control flows, and referring to the configuration settings to discover various types of security bugs.

SpotBugs (FindBugs) https://spotbugs.github.io/

- An open-source static **bytecode** analyzer for Java from the U. of Maryland.
- Looks for defects based on bug patterns
  - Bug patterns come from real bugs
- Bug patterns are grouped into categories
  - correctness, bad practice, performance…
- Assigned a priority: high, medium or low
  - High-medium priority has low false positive rates
- False positives
  - A code snippet is warned on, but is not a problem
  - (false negative: a real bug, which should be reported, is not reported)

Bug Patterns

- broad and common patterns
  - a read or write on a null pointer (limited support)
  - typos
  - methods whose return values should not be ignored
    - java.io.InputStream.read(), *.read(byte[] buffer)
    - Strings.replace(…), String.trim(…) // String is immutable in Java
- also specific bug patterns
  - Eclipse documented bug fixes
  - Every chapter in Effective Java (a book)
- to catch common errors
  - wrong Boolean operators, forgetting parentheses
  - misunderstanding of APIs

Some Bug Categories

- Correctness: the code is doing something wrong
- Code Smell: the code violates common practice
- Concurrency
- Performance
- Security threats

Example

```
if (listeners == null)
    listeners.remove(listener)
```

JDK 1.6.0, b105, sun.awt.x11.XMSelection

Example

```
public String sendMessage(User user, String body, Date time) {
  return sendMessage(user, body, null);
}

public String sendMessage(User user, String body, Date time, List attachments) {
  String xml = buildXML(body, attachments);
  String response = sendMessage(user, xml);
  return response;
}
```


Example – Infinite Recursive Loop (High-Priority Correctness)

```
public String sendMessage(User user, String body, Date time) {
  return sendMessage(user, body, null);
}

public String sendMessage(User user, String body, Date time, List attachments) {
  String xml = buildXML(body, attachments);
  String response = sendMessage(user, xml);
  return response;
}
```

Example

```
public String foundType() {
  return this.foundType();
}
```


Example

```
public String foundType() {
  return this.foundType();
}

public String foundType() {
  return this.foundType;
}
```

A real bug in JDK 1.6.0 b13

Example

```
if (name != null || name.length > 0)
```


Example

```
if (name != null || name.length > 0)
```

Should be

```
if (name != null && name.length > 0)
```

Found in `com.sun.corba.se.impl.naming.cosnaming.NamingContextImpl`

Example

- `if (part == null | part.equals(“”) )`
- `if (part == null || part.equals(“”) )`
- Found in `com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser`

Example

```
if (g != null)
    paintScrollBars(g, colors)
g.dispose()
```


SpotBugs – Nullness Checking (Example)

```
public void calledAnywhereIDoNotCare() {
  //[...]
  //parameter could be null but shouldn't ever be by logic
  method(parameter); //FindBugs says the problem is here
  //[...]
}

public final ReturnType method(final ParameterType parameter) {
  //this method do nothing but simply call anotherMethod()
  return anotherMethod(parameter, false);
}

public final ReturnType anotherMethod(final ParameterType parameter, boolean boolPa
  if (parameter == null) {
    //just in case logic is wrong
    throw new NullPointerException("I know it shouldn't be null by logic, but it is
  }
  //do something very usefull
  //[...]
}
```

https://stackoverflow.com/questions/18587851/findbugs-why-did-i-get-np-null-param-deref-and-what-would-be-better

Example

```
// BoundedThreadPool

// a big lock here that is a constant string.
private final String lock = “LOCK”;

synchronized(lock) {
    ……
}
```

Example

```
try { … }
catch (RuntimeException e) {
  AnotherException("Something happened: ", e);
}
```


Example

```
try { … }
catch (RuntimeException e) {
  AnotherException("Something happened: ", e);
}
```

```
try { … }
catch (RuntimeException e) {
  throw AnotherException("Something happened: ", e);
}
```


Last Example

```
BigDecimal amount = new BigDecimal();

for (XXX entry: entries) {
    amount.add(entry.getAmount());
}

print amount;
```


PMD https://pmd.github.io/

- "PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports Java, JavaScript, PLSQL, Apache Velocity, XML, XSL.
- Additionally it includes CPD, the copy-paste-detector. CPD finds duplicated code in Java, C, C++, C#, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Ruby, Scala, Objective C, Matlab, Python, Go."
- Searching for **patterns** on Abstract Syntax Tree (AST)

PMD https://pmd.github.io/

- SimplifyConditional: [design ruleset] detect redundant null checks

```
1 class Foo {
2   void bar(Object x) {
3     if (x != null && x instanceof Bar) {
4       // just drop the "x != null" check
5     }
6   }
7 }
```


PMD https://pmd.github.io/

```
void traverse(Tree tree) {
  List<TreeNode> worklist = new ....();
  worklist.add(tree.getRoot);
  while (worklist.size() > 0) { // while(!worklist.isEmpty())
    TreeNode node = worklist.remove();
    ......
  }
}
```


CheckerFramework https://checkerframework.org/

- An extension to Java's type system.
  - Nullness checker for null pointer errors
  - Initialization checker to ensure all fields are set in the constructor
  - Lock checker for concurrency and lock errors
  - Regexp checker to prevent use of syntactically invalid regular expressions
  - Format string checker for validity of the format string
  - …
- Based on data-flow analysis on source code (AST)
- Runs as a plugin of javac

Example -- CheckerFramework

```
import org.checkerframework.checker.nullness.qual.*;
public class GetStarted {
  void sample() {
    @NonNull Object ref = new Object();
  }
}
```

Example -- CheckerFramework

```
import org.checkerframework.checker.nullness.qual.*;
public class GetStarted {
  void sample() {
    @NonNull Object ref = null; // new Object();
  }
}
```

```
GetStarted.java:5: incompatible types.
found   : @Nullable <nulltype>
required: @NonNull Object
        @NonNull Object ref = null;
                              ^
1 error
```


ErrorProne https://errorprone.info/

- hooks into your standard build, so all developers run it without thinking
- tells you about mistakes immediately after they're made
- produces suggested fixes, allowing you to build tooling on it
- Plugin of javac, pattern-based on AST

ErrorProne https://errorprone.info/

- hooks into your standard build, so
  thinking
- tells you about mistakes immediat
- produces suggested fixes, allowing
- Plugin of javac, pattern-based on A

```
public class ArrayEqualsPositiveCases {

  public void intArray() {
    int[] a = {1, 2, 3};
    int[] b = {1, 2, 3};

    // BUG: Diagnostic contains: Arrays.equals(a, b)
    if (a.equals(b)) {
      System.out.println("arrays are equal!");
    } else {
      System.out.println("arrays are not equal!");
    }

    // BUG: Diagnostic contains: Arrays.equals(a, b)
    if (Objects.equal(a, b)) {
      System.out.println("arrays are equal!");
    } else {
      System.out.println("arrays are not equal!");
    }
  }
```


Syntax-Based Testing

Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Syntax-Based Testing

- Structured Program Input
  - e.g., compilers, JS interpreters (JSON), shell terminals
  - Grammars for input space
    - e.g., regular expression, context-free grammar
- Syntax-Based Testing
  - leverage input-space grammar to create inputs (both valid and invalid)

Regular Expression

- regex, or regexp, defines a pattern specifying what strings can be matched
- text matching
  - a regex can match plain text
    - Dan → Dan, Danny, laDan, etc
- Full text matching with anchors
  - might want to match a whole line or string
    - ^Dan$ → Dan
    - only Dan, not Danny or laDan
  - ^: anchors to the front of the line, or the start of the string
  - $: anchors to the end of the line, or the end of the string

Regular Expression

- Order of results
  - the search will begin at the start of the string
- Every character is important
  - Any plain text in the expression is treated literally.
  - Nothing is neglected
    - e.g., one whitespace " " and two whitespaces " "are different

Regular Expression – Char Classes

allows specification of only certain allowable chars

- [dofZ] matches only the letters d, o, f, and Z
  - If you have a string 'dog', then [dofZ] only matches 'd'
- [A-Za-z] matches any letter
- [a-fA-F0-9] matches any hexadecimal character
- [^*$/\\] matches anything but *, $, /, or \
  - The ^ in the front of the char class specifies "not"
  - In a char class, you only need to escape \(]-^

Regular Expression – Char Classes

- Special character classes match specific characters
- \d matches a single digit
- \w matches a word character (A-Z, a-z, _)
- \b matches a word boundary \bword\b (if want to search for a word)
  - \ba\b matches !a!, does not match aa, ab, or bab
- \s matches a whitespace character (space, tab, newline)
- . wildcard matches everything except newlines

- To match “anything but…” capitalize the char class
  - e.g., \D matches anything that is not a digit

Regular Expression – Char Classes

- e\w\w → matches ear, eye, etc
- \s\d → 1, 2, 3 strikes: matches “ 2”
- [\s\d] → 1, 2, 3 strikes: matches “1”
- \d\d\d-\d\d\d-\d\d\d\d→ 519-007-0007

Regular Expression – Repetition

- \d{2,3} → matches two or three digits
- \w{5,} → matches five letters or longer
- \d{9} → matches nine digits

General Quantifiers
- \d* → zero occurrences or more
- \w+ → one or more occurrences
- \w? → zero or one occurrence

Regular Expression – Alternation

Alternation allows multiple possibilities
- true|false → matches either true or false

Regular Expression

- Consider this Perl Regular expression
  - ^(deposit|debit) [0-9]{3} \$[0-9]+\.[0-9]{2}$
- What are some valid strings?

Grammar

^(deposit|debit) [0-9]{3} \$[0-9]+\.[0-9]{2}$

- What is the limitation of regular expressions?
- Context-Free Grammar
  - `action = dep | deb`
  - `dep = "deposit" account amount`
  - `deb = "debit" account amount`
  - `account = digit{3}`
  - `amount = "$" digit+ "." digit{2}`
  - `digit = ["0"-"9"]`

Using Grammar

- Two ways to use input grammars for software testing and maintenance:
  - Recognizer: can include them in a program to validate inputs;
  - Generator: can create program inputs for testing.

Grammar Terminology

- A finite set N of *nonterminal symbols*.
- A finite set Σ of *terminal symbols* that is disjoint from N.
- A finite set P of *production rules*, each rule of the form

  $$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

  where * is the Kleene star operator and $\cup$ denotes set union, so $(\Sigma \cup N)^*$ represents zero or more symbols, and N means one *nonterminal* symbol. That is, each production rule maps from one string of symbols to another, where the first string contains at least one nonterminal symbol. In the case that the body consists solely of the empty string—i.e., that it contains no symbols at all—it may be denoted with a special notation (often Λ, e or ϵ) in order to avoid confusion.
- A distinguished symbol $S \in N$ that is the *start symbol*.

Grammar Terminology

Start symbol: actions
non-terminals: ?
terminals: ?
production rules: ?
derivable strings:?

- **actions** = action*
- action = dep | deb
- dep = "deposit" account amount
- deb = "debit" account amount
- account = digit{3}
- amount = "$" digit+ "." digit{2}
- digit = ["0"-"9"]

Syntax-Based Testing

- Creating a grammar for a system that does not have one, is a useful QA exercise.
- How to use input-space grammars?
  - create inputs (both “valid” and invalid)
  - Why is *valid* put in quotes?

Syntax-Based Testing

- Creating a grammar for a system that does not have one, is a useful QA exercise.
- How to use input-space grammars?
  - create inputs (both “valid” and invalid)
  - Why is *valid* put in quotes?

```
// a C program
int main() {
  return main(1);
}
```

```
// a C program
int main() {
  return INT_MAX + 1;
}
```


Syntax-Based Testing

- Creating a grammar for a system that does not have one, is a useful QA exercise.
- How to use input-space grammars?
  - create inputs (both “valid” and invalid)
  - Why is *valid* put in quotes?
- Coverage Criteria
  - Terminal Symbol Coverage (TSC). TR contains each terminal of grammar G.
  - Production Coverage (PDC). TR contains each production of grammar G.
  - Derivation Coverage (DC). TR contains every possible string derivable from G.

```
actions = action*
action = dep | deb
dep = "deposit" account amount
deb = "debit" account amount
account = digit{3}
amount = "$" digit+ "." digit{2}
digit = ["0"-"9"]
```

Start symbol: actions
non-terminals: actions action dep deb account amount
terminals: deposit debit $. 0 1 2 3 4 5 6 7 8 9
production rules:
- actions->action*
- action->dep
- action->deb
- dep->…
- deb->…
- account->…
- amount->…
- digit -> …

derivable strings:infinite

TSC: TR={all terminals}
PDC: TR={all production rules}

- Does TSC subsume PDC? Or vice versa?
- If PDC is edge coverage, is TSC node coverage?

Testing Invalid Inputs

Brenan Keller @brenankeller · Nov 30, 2018
A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

477 25.5K 63.7K

Testing Invalid Inputs

- Your program accepts strings described by a regex or a grammar
  - Was it implemented correctly?
  - Maybe, if you use a parser generator or regex, now or in the future
- Reasons to test invalid inputs
  - Implementors often overlook invalid inputs

Brenan Keller @brenankeller · Nov 30, 2018
A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

477 25.5K 63.7K

Testing Invalid Inputs -- Solutions

- Mutate grammars and generate test strings from the mutated grammars.
- Use correct grammar, but mis-derive a rule node once – gives “closer” inputs (since you only miss once)

Brenan Keller @brenankeller · Nov 30, 2018
A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

477 25.5K 63.7K

Terminology

- Ground String: A (valid) string belonging to the language of the grammar.
- Mutation Operator: A rule that specifies syntactic variations of strings generated from a grammar.
- Mutant: The result of one application of a mutation operator to a ground string.

- Mutants may be generated either by modifying existing strings or by changing a string while it is being generated.

Some Grammar Mutation Operators

ground string: deposit 123 $12.35

- Nonterminal Replacement
  - *dep = "deposit" account amount =>*
  - *dep = "deposit" amount amount*
  - Use your judgement to replace nonterminals with similar nonterminals.

deposit $9.22 $12.35

- Terminal Replacement
  - *amount = "$"digit+"."digit{2} =>*
  - *amount = "$"digit+"$"digit{2}*

deposit 123 $12$35

More Grammar Mutation Operators

ground string: deposit 123 $12.35

- Terminal and Nonterminal Deletion; e.g.
  - *dep = "deposit" account amount =>*
  - *dep = "deposit" amount*

deposit $12.35

- Terminal and Nonterminal Duplication; e.g.
  - *dep = "deposit" account amount =>*
  - *dep = "deposit" account account amount*

deposit 123 123 $12.35

Test Oracle for Syntax-Based Testing

- For invalid inputs, the program should reject them, or maybe you can try to repair the inputs automatically if necessary.
  - Example?
  - I personally don't think repairing inputs is right. I prefer always reject invalid ones.
  - There might be some exceptions?

Test Oracle for Syntax-Based Testing

- For invalid inputs, the program should reject them, or maybe you can try to repair the inputs automatically if necessary.
  - Example?
- Question:
  - If my program rejects every invalid inputs generated by mutation, does this guarantee that my project is bullet-proof of invalid inputs?

Test Oracle for Syntax-Based Testing

- For invalid inputs, the program should reject them, or maybe you can try to repair the inputs automatically if necessary.
  - Example?
- Question:
  - If my program rejects every invalid inputs generated by mutation, does this guarantee that my project is bullet-proof of invalid inputs?
  - Semantically invalid inputs.

How to Generate Semantically Invalid Inputs?

How to Generate Semantically Invalid Inputs?

ground string: deposit 123 $12.35

- One solution: mutating ground strings wrt. the grammar
- Terminal Replacement
  - *dep = "deposit" account amount =>*
  - *dep = "debit" account amount*

mutant: debit 123 $12.35

- Terminal Swap

mutant: debit 123 $35.12

How to Generate Semantically Invalid Inputs?

ground string: deposit 123 $12.35

- One solution: mutating ground strings wrt. the grammar
- Terminal Replacement
  - *dep = "deposit" account amount =>*
  - *dep = "debit" account amount*

  mutant: debit 123 $12.35
- Terminal Swap

  mutant: debit 123 $35.12

Challenge: you do not know if the mutant is indeed semantically invalid

Test Oracle for Maybe Semantically Invalid Inputs

- Non-functional properties
  - crashes
  - safety/correctness of memory operations, e.g., asan, msan, valgrind
  - performance or memory usage, e.g., hang

An Example: tkfuzz

- A fuzzing tool to generate “maybe” semantically invalid inputs for compilers
  - For compilers, inputs are programs.
- Given a ground string (which is a C/C++ program), tkfuzz randomly substitutes an identifier token with another different identifier token in the token list.
  - identifier: functions, variables, type names

•Chengnian Sun, Vu Le, Qirun Zhang, Zhendong Su:
Toward understanding compiler bugs in GCC and LLVM. ISSTA 2016: 294-305

An Example: tkfuzz

- A fuzzing tool to generate “maybe” semantically invalid inputs for compilers
  - For compilers, inputs are programs.
- Given a ground string (which is a C/C++ program), tkfuzz randomly substitutes an identifier token with another different identifier token in the token list.
  - identifier: functions, variables, type names

```
int main() {
  int a = 0;
  const char* b = "hi";
  const char* c = b;
  return c[0];
}
```

```
int main() {
  int a = 0;
  const char* b = "hi";
  const char* c = main;
  return c[0];
}
```

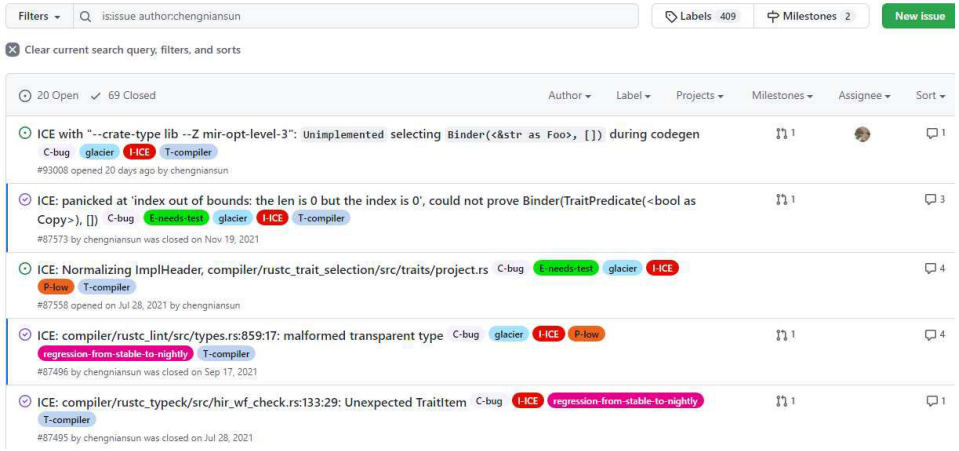
•Chengnian Sun, Vu Le, Qirun Zhang, Zhendong Su:
Toward understanding compiler bugs in GCC and LLVM. ISSTA 2016: 294-305

An Example: tkfuzz

- Evaluation:
  - Ground strings: 36,966 test programs in GCC
- Very effective
  - 18 bugs found in only a month

Table 8: Bugs Found by tkfuzz

	Bug ID	Component	Status
1	LLVM-24610	frontend	fixed
2	LLVM-24622	frontend	fixed
3	LLVM-24797	new-bugs	–
4	LLVM-24798	c++	fixed
5	LLVM-24803	new-bugs	–
6	LLVM-24884	new-bugs	–
7	LLVM-24943	new-bugs	–
8	LLVM-25593	new-bugs	–
9	LLVM-25634	new-bugs	–
10	GCC-67405	target	fixed
11	GCC-67581	c++	fixed
12	GCC-67619	middle-end	fixed
13	GCC-67639	middle-end	confirmed
14	GCC-67653	middle-end	fixed
15	GCC-67845	c++	fixed
16	GCC-67846	c++	fixed
17	GCC-67847	c++	fixed
18	GCC-68013	tree-opt	fixed



A Further Step: Skeletal Program Enumeration (SPE)

- Given a token sequence, enumerate all programs
  - exhaustively swap identifier pairs
  - pruning
    - bounded length of grounded strings,
    - eliminating duplicates, otherwise, exponential number of mutants

```
a := 10;          □ := 10;          b := 10;          a := 10;
b := 1;           □ := 1;           a := 1;           b := 1;
while(a) do       while(□) do       while(b) do       while(b) do
  a := a − b;       □ := □ − □;       b := b − a;       b := a − b;
```

(a) Program P (b) Skeleton $\mathbb{P}$ (c) Program P_1 (d) Program P_2

Qirun Zhang, Chengnian Sun, Zhendong Su: Skeletal program enumeration for rigorous compiler testing. PLDI 2017: 347-361

A Further Step: Skeletal Program Enumeration (SPE)

- In six months

Compiler	Summary					Classification		
	Reported	Fixed	Duplicate	Invalid	Reopened	Crash	Wrong code	Performance
GCC	136	93	10	2	1	127	6	3
Clang	81	26	3	1	1	79	2	0

Qirun Zhang, Chengnian Sun, Zhendong Su: Skeletal program enumeration for rigorous compiler testing. PLDI 2017: 347-361

Test Oracle for Maybe Semantically Invalid Inputs

- Non-functional properties
  - crashes
  - safety/correctness of memory operations, e.g., asan, msan, valgrind
  - performance or memory usage, e.g., hang
- What if I want to test functional properties?

Test Oracle for Maybe Semantically Invalid Inputs

- Non-functional properties
  - crashes
  - safety/correctness of memory operations, e.g., asan, msan, valgrind
  - performance or memory usage, e.g., hang
- What if I want to test functional properties, i.e., correctness?
  - **Differential testing**
    - Find the counterparts,
    - e.g., gcc and clang
    - e.g., different optimization levels, 'gcc –O0' vs 'gcc –O3"

Differential Test: Example, A real bug in GCC

- Generated by SPE mutation
- Compiled with both GCC and Clang
- Run the two executables, and compare the outputs

```
1  int a = 0;
2  extern int b __attribute__ ((alias (''a'')));
3
4  int main ()
5  {
6    int *p = &a, *q = &b;
7    *p = 1;
8    *q = 2;
9
10 //return b;
11   return a; // Bug: the program exits with 1
12 }
```


Generality of Differential Testing

- Applicable if you can find a different implementation
  - e.g., different sorting algorithms
  - e.g., different HTTP servers
- Threats
  - Different implementations may react differently to semantically invalid inputs
  - e.g., undefined behaviors in C/C++, and C/C++ compilers

Mutation Testing

Chengnian Sun
cnsun@

*Slides adapted from https://sttp.site/mutation-testing/

How Good are your tests?

- If your test set achieves 100% coverage level for NC, PPC or even CPC
  - what does this imply in terms of the quality of the test set?


```
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```



```
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

```
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
}

@Test
public void testZero(){
  int[] values = divide(1, 0);
}
```

```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
}

@Test
public void testZero(){
  int[] values = divide(1, 0);
}
```

100% CPC. But NO fault detection capability.


```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
}

@Test
public void testZero(){
  int[] values = divide(1, 0);
}
```

100% CPC. But NO fault detection capability.

With Assertions: Test Set v1

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```



```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
}

@Test
public void testZero(){
  int[] values = divide(1, 0);
}
```

100% CPC. But NO fault detection capability.

With Assertions: Test Set v1

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```

With Assertions: Test Set v2

```java
@Test
public void testGetValues_v2(){
  int[] values = divide(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```



```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a * b; //correct: a/b
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
}

@Test
public void testZero(){
  int[] values = divide(1, 0);
}
```

100% CPC. But NO fault detection capability.

With Assertions: Test Set v1

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```

With Assertions: Test Set v2

```java
@Test
public void testGetValues_v2(){
  int[] values = divide(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```

```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a * b; //correct: a/b
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
}

@Test
public void testZero(){
  int[] values = divide(1, 0);
}
```

100% CPC. But NO fault detection capability.

With Assertions: Test Set v1

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```

With Assertions: Test Set v2

```java
@Test
public void testGetValues_v2(){
  int[] values = divide(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```



```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a * b; //correct: a/b
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

Fault Detection Capability

- Why do we have such a difference in the test results?
- The two tests have the same code coverage and the same type of assertions.
- The second test set has better input parameters and assertions that allow to detect the injected fault.

With Assertions: Test Set v1

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```

With Assertions: Test Set v2

```java
@Test
public void testGetValues_v2(){
  int[] values = divide(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```



```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a * b; //correct: a/b
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```

What about this bug?

```java
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a - b; //correct: a/b
  int remainder = a % b;
  return new int[] {quotient, remainder};
}
```


With Assertions: Test Set v1

```java
@Test
public void testGetValues(){
  int[] values = divide(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```

With Assertions: Test Set v2

```java
@Test
public void testGetValues_v2(){
  int[] values = divide(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
@Test
public void testZero(){
  int[] values = divide(1, 0);
  assertNull(values);
}
```


Mutation Testing

- Idea: Inserting artificial defects (mutants) in the production code to assess the quality of the test code
- Effective test suite: at least one of its test cases fails when executing the test suite against the mutants
- Mutation testing is like testing the tests

Mutation Testing – An Example

Mutation Operations

Mutants

- Inserting artificial defects (mutants) in the production code to assess the quality of the test code
- How significant should the changes be?
  - One line of code?
  - Multiple lines?

Hypothesis

- Hypothesis 1
  - The Competent Programmer Hypothesis (CPH): given a specification, a programmer develops a program that is either correct or differs from the correct programs by a combination of simple errors
- Hypothesis 2
  - Coupling effect: test cases that detect simple types of faults can also discover more complex forms of faults

Terminology

- Mutant: Given a program P, a mutant P’ is obtained by introducing a simple syntactic change to P
- Syntactic change: small changes that make the mutated code valid (i.e., it can be compiled)
- Change: alterations to the production code that mimic typical human mistakes (glitches)


```java
public class Fraction {
  int numerator;
  int denominator; // always positive
  public Fraction(int numerator, int denominator){...}
  public Fraction invert() {
    if (numerator == 0) {
      throw new ArithmeticException("...");
    }
    if (numerator==Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }
    if (numerator<0) {
      return new Fraction(-denominator, -numerator);
    }
    return new Fraction(denominator, numerator);
  }
}
```

MIN_VALUE = -2147483648
MAX_VALUE = 2147483647

https://stackoverflow.com/questions/31540421/why-does-the-negative-of-integer-min-value-give-the-same-value


```java
public class Fraction {
  int numerator;
  int denominator; // always positive
  public Fraction(int numerator, int denominator){...}
  public Fraction invert() {
    if (numerator == 0) {
      throw new ArithmeticException("...");
    }
    if (numerator==Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }
    if (numerator<0) {
      return new Fraction(-denominator, -numerator);
    }
    return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_negative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_zero(){
  Fraction f = new Fraction(0, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
@Test
public void testInvert_minValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
```





```java
public class Fraction {
  int numerator;
  int denominator; // always positive
  public Fraction(int numerator, int denominator){...}
  public Fraction invert() {
    if (numerator == 0) {
      throw new ArithmeticException("...");
    }
    if (numerator==Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }
    if (numerator<0) {
      //return new Fraction(-denominator, -numerator);
      return new Fraction(-denominator, numerator);
    }
    return new Fraction(denominator, numerator);
  }
}
```

Mutant 1:
We remove the sign (-) from the numerator.

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_negative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_zero(){
  Fraction f = new Fraction(0, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
@Test
public void testInvert_minValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
```

```java
public class Fraction {
  int numerator;
  int denominator; // always positive
  public Fraction(int numerator, int denominator){...}
  public Fraction invert() {
    if (numerator == 0) {
      throw new ArithmeticException("...");
    }
    if (numerator==Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }
    if (numerator<0) {
      //return new Fraction(-denominator, -numerator);
      return new Fraction(-denominator, numerator);
    }
    return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_negative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_zero(){
  Fraction f = new Fraction(0, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
@Test
public void testInvert_minValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
```

Mutant 1:
We remove the sign (-) from the numerator.


```java
public class Fraction {
  int numerator;
  int denominator; // always positive
  public Fraction(int numerator, int denominator){...}
  public Fraction invert() {
    //if (numerator == 0) {
    if (numerator == 1) {
      throw new ArithmeticException("...");
    }
    if (numerator==Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }
    if (numerator<0) {
      return new Fraction(-denominator, -numerator);
    }
    return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_negative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_zero(){
  Fraction f = new Fraction(0, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
@Test
public void testInvert_minValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
```

Mutant 2:
We change the constant in the if condition.


```java
public class Fraction {
  int numerator;
  int denominator; // always positive
  public Fraction(int numerator, int denominator){...}
  public Fraction invert() {
    //if (numerator == 0) {
    if (numerator == 1) {
      throw new ArithmeticException("...");
    }
    if (numerator==Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }
    if (numerator<0) {
      return new Fraction(-denominator, -numerator);
    }
    return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_negative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.00001);
}
@Test
public void testInvert_zero(){
  Fraction f = new Fraction(0, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
@Test
public void testInvert_minValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  assertThrows(ArithmeticException.class,
               () -> {f.invert();});
}
```

Mutant 2:
We change the constant in the if condition.

Generating Mutants Automatically

- Mutation operators: rules to apply syntactic changes to the code under tests
- Real fault based operators: operators that apply changes very similar to defects seen in the past for the same code
- Language-specific operators: mutations for the inheritance in Java, mutations for pointers in C, etc.

Mutation Operators

Basic Operators

- Arithmetic Operator Replacement (AOR)
- Relational Operator Replacement (ROR)
- Conditional Operator Replacement (COR)
- Assignment Operator Replacement (AOR)
- Scalar Variable Replacement (SVR)

Arithmetic Operator Replacement (AOR)

- This operator replaces an arithmetic operation (+, -, *, /, %) in the production code with an alternative operator

Arithmetic Operator Replacement (AOR)

- This operator replaces an arithmetic operation (+, -, *, /, %) in the production code with an alternative operator

```
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;

  return new int[] {quotient, remainder};
}
```

+, - , * , %

+, - , * , /

Arithmetic Operator Replacement (AOR)

- This operator replaces an arithmetic operation (+, -, *, /, %) in the production code with an alternative operator

```
public static int[] divide(int a, int b){
  if (b == 0){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;

  return new int[] {quotient, remainder};
}
```

quotient=a;

remainder=b;

Relational Operator Replacement (ROR)

This operator replaces relational operators (<,>,<=,>=,==,!=) in the production code with an alternative operator

```
public static int[] divide(int a, int b){
  if (b == 0){                          <=,.+,!=,>,<
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;

  return new int[] {quotient, remainder};
}
```


Conditional Operator Replacement (COR)

This operator replaces conditional operators (&&, ||, &, |, !, ^) in the production code with an alternative operator.

```
public static int[] divide(int a, int b){
  if (b == 0 && b == Integer.MIN_VALUE){
    return null;                        &, ||, |
  }
  int quotient = a / b;
  int remainder = a % b;

  return new int[] {quotient, remainder};
}
```


Assignment Operator Replacement (AOR)

This operator replaces assignment operators (=,+=,-=,*=,…) in the production code with an alternative operator

```
public static int[] divide(int a, int b){
  if (b == 0 && b == Integer.MIN_VALUE){
    return null;          +=,-=,/=, …
  }
  int quotient = a / b;
  int remainder = a % b;
                          +=,-=,/=, …
  return new int[] {quotient, remainder};
}
```


Scalar Variable Replacement (SVR)

Each variable reference is replaced with another variable reference of the same type and that is already declared in the code

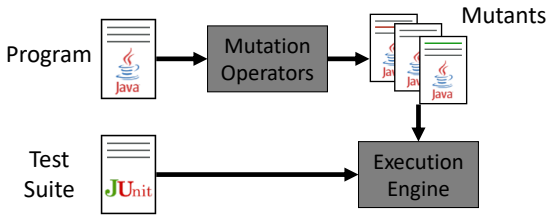
```
public static int[] divide(int a, int b){
  if (b == 0 && b == Integer.MIN_VALUE){
    return null;
  }
  int quotient = a / b;
  int remainder = a % b;

  return new int[] {quotient, remainder};
}
```

Object-Oriented Operators

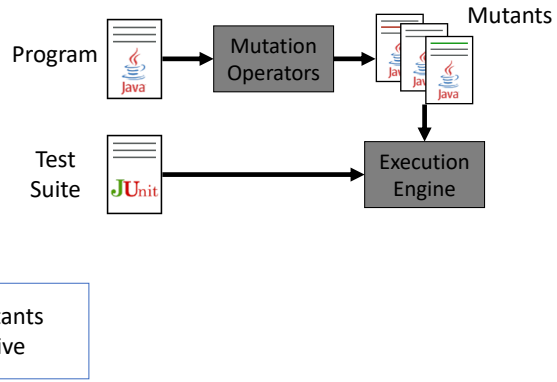
- Access Modifier Change
- Hiding Variable Deletion
- Hiding Variable Insertion
- Overriding Method Deletion
- Parent Constructor Deletion
- Declaration Type Change
- …

Mutation Score

Mutation Process



Mutation Process



Mutation Score = $\frac{\#Killed\ Mutants}{\#\ Mutants}$

- Mutation Analysis: assessing the quality of the test suite by computing the mutation score
- Mutation Testing: improving the quality of the test suite using mutants (i.e., adding, changing tests)

Equivalent Mutants

- Equivalent Mutant: a mutant M that is functionally equivalent to the original program P
- Program Equivalence: two programs A and B are functionally equivalent if they always produce the same output on every input

Equivalent Mutants: An Example

Original Program

```
public void method(int a){
    int index = 10;
    while (…){
        …
        index--;
        if (index == 0)
            break;
    }
}
```

Equivalent Mutant

```
public void method(int a){
    int index = 10;
    while (…){
        …
        index--;
        if (index <= 0)
            break;
    }
}
```


Mutation Score = $\frac{\#Killed\ Mutants}{\#\ Mutants}$

- Can we detect equivalent mutant automatically?
- Unfortunately, the answer is NO. Detecting equivalent mutants is an undecidable problem, just like detecting infeasible paths.

Mutation Score = $\frac{\#Killed\ Mutants}{\#\ Mutants}$

In our textbook:

- Strongly Killing Mutants: Given a mutant *m* for a program *P* and a test *t*, *t* is said to strongly kill *m* iff the output of *t* on *P* is different from the output of *t* on *m*.
- Strong Mutation Coverage (SMC): For each mutant *m,* TR contains a test which strongly kills *m*.

Uninteresting Mutants (for A2)

Try to avoid generating the following types of mutants

- Stillborn: such mutants cannot compile (or immediately crash)
- Trivial: killed by almost any test case
- Equivalent: indistinguishable from original program

Are Mutants Realistic?


Is Mutation an Appropriate Tool for Testing Experiments?

J.H. Andrews et al. ICSE 2005

"Our analysis suggests that mutants, when using carefully selected mutation operators and after removing equivalent mutants, can provide a good indication of the fault detection ability of a suite"

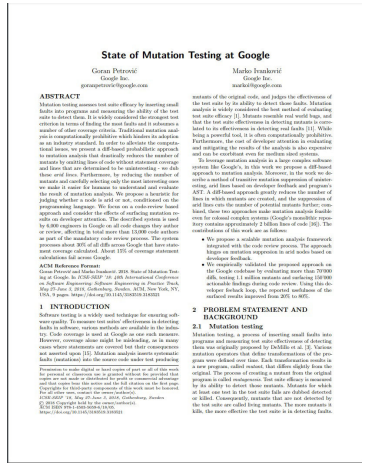

Are Mutants a Valid Substitute for Real Faults in Software Testing?

R. Just, et al. FSE 2014

"Mutant detection is positively correlated with real fault detection, independently of code coverage"

"This correlation is stronger than the correlation between statement coverage and real fault detection"

Is Mutation Testing Practical?


State of Mutation Testing at Google

The Cost of Mutation Testing

Let's assume we have:

- A code base with 300 Java classes
- 10 tests for each class
- On average, each test requires 0.2 seconds for its execution
- The total test set execution costs 300 * 10 * 0.2 = 600s (10 minutes)

The Cost of Mutation Testing

Let's assume we have:

- A code base with 300 Java classes
- 10 tests for each class
- On average, each test requires 0.2 seconds for its execution
- The total test set execution costs 300 * 10 * 0.2 = 600s (10 minutes)

- Let's assume we have, on average, 20 mutants per each class. The total cost of mutation analysis is:
  - **300 * 10 * 0.2 * 20 = 12000s = 3h 20 minutes**

Speeding-up Mutation Testing

Observation: a mutant cannot be killed if the test set does not cover the statement where the syntactic change is injected (reachability condition)

Speeding-up Mutation Testing

Observation: a mutant cannot be killed if the test set does not cover the statement where the syntactic change is injected (reachability condition)

- **Heuristic 1**: we run only the tests that reach (cover) the target mutant
- **Heuristic 2**: if a mutant is already killed by one test case, there is no need to execute the remaining test
- Other heurists?

Speeding-up Mutation Testing

Observation 2: mutants generated by the same operator and injected in the same location (instruction) are likely coupled to the same type of real faults

Speeding-up Mutation Testing

Observation 2: mutants generated by the same operator and injected in the same location (instruction) are likely coupled to the same type of real faults

- Heuristic 3: considering/executing only a subset of the generated mutants ("do fewer"). The most straightforward way is random sampling
- Random sampling is the most simple, yet effective strategy to reduce the cost of mutation testing

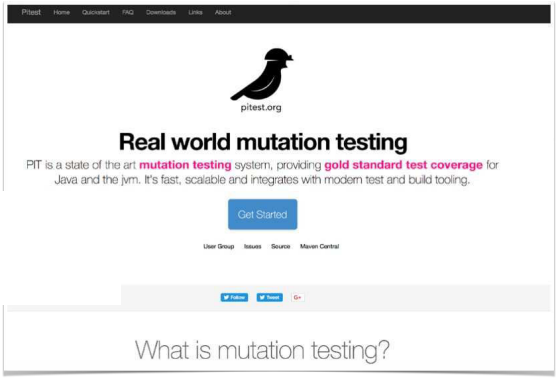
Mutation Testing Tool

- For Java:
  - PIT
  - MuJava
  - Bacterio
  - Javalanche
  - Major
  - Descardes
- For PHP:
  - Humbug
  - Infection PHP
- For JavaScript:
  - Stryker
- For C#:
  - Nester
  - VisualMutator
- For C/C++
  - Dextool Mutate
  - Mutate.py

PIT Testing Tool

http://pitest.org

```
java -cp <jar and dependencies> \
   org.pitest.mutationtest.commandline.MutationCoverageReport\
   --reportDir <outputdir> \
   --targetClasses com.your.package.tobemutated* \
   --targetTests com.your.package.*
   --sourceDirs <pathtosource>
```



Mutation Testing vs. Mutation-Based Testing

Mutation Testing vs. Mutation-Based Testing

- Common: mutation
- Purpose:
  - Mutation testing: to assess the quality of tests
  - Mutation-based testing: to create more tests
- Difference:
  - mutation testing: mutation on programs
  - syntax-based testing: mutation on program inputs

Then why do we study NC, EC, PPC, etc

Then why do we study NC, EC, PPC, etc

- necessary condition
  - no coverage → very bad
  - with coverage → unsure (e.g., no assertions)
- My Opinion: coverage is a size metric, w.r.t. a specific project
  - Is a set of 1000 tests large?
    - for “hello world!”, yes: 100% coverage
    - for Linux Kernel, no: tiny coverage level

Beliefs, Bug Finding and Coverity

Chengnian Su
University of Waterloo
cnsun@uwaterloo.ca

Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's.

Finding Bugs Requires Specifications

To determine whether a piece of code has a bug, you need to have a specification/rule/intention/criterion to check whether the code can produce expected results

- common types of bugs: generally available pre-defined specifications
  - `int a[4]; a[4] = 4;` // buffer overflow: access should be within the array abound
  - `int *p = nullptr; *p = 1;` // null dereferencing: nullptr should not be dereferenced.
  - `int *p = malloc(...); int *copy = p; free(p); *copy = 1;` // dangling pointers
  - `int a = 1 / 0;` // division by zero.

Finding Bugs Requires Specifications

To determine whether a piece of code has a bug, you need to have a specification/rule/intention/criterion to check whether the code can produce expected results

- functionality bugs (w.r.t. user requirements):
  - domain-specific, specifications usually unavailable
  - There is a function `static int reset_hardware(...) {...}` in the Linux kernel
  - implicit specification: callers of this function must acquire the lock.

```
// linux/driver/scsi/in2000.c
static int in2000_bus_reset(...) {
  .....
  reset_hardware(...);
  ...
}
```


Finding Bugs Requires Specifications

To determine whether a piece of code has a bug, you need to have a specification/rule/intention/criterion to check whether the code can produce expected results

- functionality bugs (w.r.t. user requirements):
  - domain-specific, specifications usually unavailable
  - There is a function `static int reset_hardware(...) {...}` in the Linux kernel
  - implicit specification: callers of this function must acquire the lock.

```
// linux/driver/scsi/in2000.c
static int in2000_bus_reset(...) {
  .....
  // no lock acquisition ➔ A bug!
  reset_hardware(...);
  ...
}
```

Finding Bugs Requires Specifications

- functionality bugs (w.r.t. user requirements):
  - domain-specific, specifications usually unavailable
  - There is a function `static int reset_hardware(...) {...}` in the Linux kernel
  - implicit specification: callers of this function must acquire the lock.
- other examples
  - `length < 100`
  - `x = 16*y+4*z+3`
  - `java.util.HashMap` can take `null` keys and `null` values, whereas `com.google.common.collect.ImmutableMap` does not accept `null` keys or `null` values
  - `close()` is always called after a stream is created.
  - callers of `reset_hardware()` must acquire the lock.

```
final FileInputStream stream = new FileInputStream("file.txt");
stream.read();
stream.close();
```


How to Get Specifications?

- programming languages come with some specifications
  - common types of bugs, e.g., buffer overflow, division by zero
  - all the patterns defined in Error Prone
  - streams need to be closed when they are not needed.
- developers write specification
  - specifications can be easily get outdated.

```
import java.util.Set;
import java.util.HashSet;

public class ShortSet {
  public static void main (String[] args) {
    Set<Short> s = new HashSet<>();
    for (short i = 0; i < 100; i++) {
      s.add(i);
      s.remove(i - 1);
    }
    System.out.println(s.size());
  }
}
```

```
$ bazel build :hello
ERROR: example/myproject/BUILD:29:1: Java compilation in rule '//example/myproject:hello'
ShortSet.java:6: error: [CollectionIncompatibleType] Argument 'i - 1' should not be passed to this method;
its type int is not compatible with its collection's type argument Short
      s.remove(i - 1);
               ^
    (see http://errorprone.info/bugpattern/CollectionIncompatibleType)
1 error
```


How to Get Specifications?

- programming languages come with some specifications
  - common types of bugs, e.g., buffer overflow, division by zero
  - all the patterns defined in Error Prone
  - streams need to be closed when they are not needed.
- developers write specification
  - specifications can be easily get outdated.
- use automated tools
  - static analysis: coverity, inferring specifications from source code
  - dynamic analysis: daikon, inferring from execution traces
    - https://plse.cs.washington.edu/daikon/

Bugs as Deviant Behaviors

-- A general approach to inferring errors in systems code

Dawson Engler, DavidYu Chen, Seth Hallem, Andy Chou, Benjamin Chelf: SOSP'01

Impact

- Coverity (http://coverity.com/)
  - able to find many bugs in large programs (millions lines of code)
  - a leading company in building bug detection tools
  - 900+ customers including BlackBerry, Yahoo, Mozilla (not accurate any more)
  - Acquired by Synopsys (https://scan.coverity.com/)

"No, your tool is broken: that's not a bug"

- "No, the loop will go through once!"

```
for(i=1; i < 0; i++) {
  …deadcode…
}
```

- "No, there is no malloc call in between"

```
free(foo);
foo->base = …;
```

- "No, I meant to do that; they are next to each other"
- "No, ANSI lets you write 1 past end of the array!"

```
int a[2], b;
memset(a, 0, 12);
```

```
unsigned p[4];   p[4] = 1;
```


Goal: Find as many serious bugs as possible

- Intuition: how to find errors without knowing truth?
  - Contradiction. To find lies: cross-examine. Any contradiction is an error
  - Deviance. To infer correct behaviors: If one person does X, might be right or a coincidence. If 1000s do X and one does Y, probably an error
- Crucial: we know contradiction is an error without knowing the correct belief.

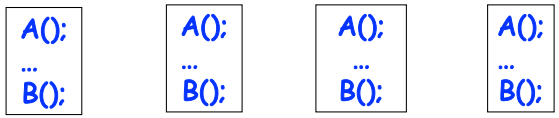
Cross-checking program belief systems

- Must beliefs:
  - Inferred from acts that imply beliefs code "must" have.
  - Check using internal consistency: infer beliefs at different locations, then cross-check for contradiction.

```
x = *p / z; // MUST belief: p not null
            // MUST: z != 0
unlock(l);   // MUST: l acquired
x++;         // MUST: x not protected by l
```

Cross-checking program belief systems

- May beliefs:
  - Could be coincidental.
  - Inferred from acts that imply beliefs code may have



// MAY: A() and B() must be paired

Trivial consistency: NULL pointers

- *p implies MUST belief: p is not null
- A check (p == null) implies two MUST beliefs:
  - POST: p is null on the true branch, and not null on the false branch
  - PRE: p was unknown before the check
- Cross-check these for three different error types.
  - Check-then-use (79 errors, 26 false positives)

```
        /* 2.4.1: drivers/isdn/svmb1/capidrv.c */
if (!card)
    printk(KERN_ERR, "capidrv-%d: …", card->contrnr…);
```


Trivial consistency: NULL pointers

- Cross-check these for three different error types.
  - Check-then-use (79 errors, 26 false positives)
  - Use-then-check: 102 bugs, 4 false positives

```
/* 2.4.7: drivers/char/mxser.c */
struct mxser_struct *info = tty->driver_data;
unsigned flags;
if(!tty || !info->xmit_buf)
      return 0;
```

  - Contradiction/redundant checks: 24 bugs, 10 false positives

```
/* 2.4.7/drivers/video/tdfxfb.c */
fb_info.regbase_virt = ioremap_nocache(...);
if(!fb_info.regbase_virt)
   return -ENXIO;
fb_info.bufbase_virt = ioremap_nocache(...);
/* [META: meant fb_info.bufbase_virt!] */
if(!fb_info.regbase_virt) {
   iounmap(fb_info.regbase_virt);
```


Redundancy checking

- Assume: code supposed to be useful
  - useless actions == conceptual confusion
- Identity operations: “x=x”, “1*y”, “x&x”, “x|x”

```
/* 2.4.5-ac8/net/appletalk/aarp.c */
da.s_node = sa.s_node;
da.s_net = da.s_net;
```

- Assignments that are never read

```
for(entry=priv->lec_arp_tables[i];entry != NULL; entry=next){
   next = entry->next;
   if (…)
      lec_arp_remove(priv->lec_arp_tables, entry);
   lec_arp_unlock(priv);
   return 0;
}
```

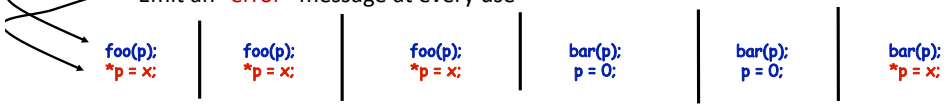
Note the return here, the loop only runs once at most.

Handling May beliefs

- MUST beliefs: only need a single contradiction
- MAY beliefs: need many examples to separate fact from coincidence
- Conceptionally
  - Assume MAY beliefs are MUST beliefs
  - Record every successful check with a “check” message
  - Every unsuccessful check with an “error” message
  - Rank errors based on the ratio of checks (n) to errors (err)
    - The most likely errors are those where n is large, and err is small

Statistical: Deriving deallocation routines

- Use-after free errors are horrible
  - Problem: lots of undocumented sub-system free functions
  - Solution: derive behaviorally: pointer “p” not used after call “foo(p)” implies MAY belief that “foo” is a free function
- Conceptually: Assume all functions free all arguments
  - (in reality: filter functions that have suggestive names)
  - Emit a “check” message at every call site
  - Emit an “error” message at every use



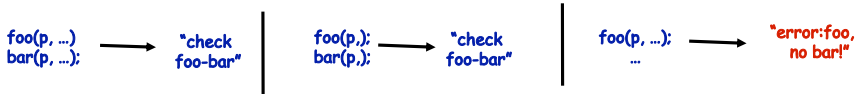
- Rank bar()’s error first

Statistical: deriving routines that can fail

- Traditional: use global analysis to track which routines return NULL
  - problem: false positives when pre-conditions hold, difficult to tell statically (“return p->next”)
- Instead: see how often program checks
  - Rank errors based on number of checks to non-checks
- Algorithm: Assume “all” functions can return NULL
  - If pointer checked before use, emit “check” message
  - If pointer used before check, emit “error”
  - Sort errors based on ratio of checks to errors
- Result: 152 bugs, 16 false

Deriving “A() must be followed by B()”

- “a();…b();” implies MAY belief that a() is followed by b()
  - Programmer may believe a-b paired, or might be a coincidence.
- Algorithm:
  - Assume every a-b is valid pair (reality: prefilter functions that seem to be plausibly paired)
  - Emit “check” for each path that has a() then b()
  - Emit “error” for each path that has a() and no b()
- Results: 23 errors, 11 false positives



Project Related

- "A() and B() must be paired": either A() then B(), or B() then A()
  - Intra-procedural
  - View a function body as a set
    - no order of function calls
    - function calls do not need to be continuous
- Support is the number of times a pair of functions appears together
  - support({A, B}) = 3
- Confidence({A, B}, {A}) = support({A, B}) / support({A}) = 3/4

```
void scope1() {
   A(); B(); C(); D();
}
void scope2() {
   A(); C(); D();
}
void scope3() {
   A(); B();
}
void scope4() {
   B(); D(); scope1();
}
void scope5() {
   B(); D(); A();
}
void scope6() {
   B(); D();
}
```


Project Related

- The sample output with the support threshold 3 and confidence threshold 65% is (intra-procedural analysis):
  - bug: A in scope2, pair: (A B), support: 3, confidence: 75%
  - bug: A in scope3, pair: (A D), support: 3, confidence: 75%
  - bug: B in scope3, pair: (B D), support: 3, confidence: 80%
  - bug: D in scope 2, pair: (B D), support: 4, confidence: 80%


```
void scope1() {
   A(); B(); C(); D();
}
void scope2() {
   A(); C(); D();
}
void scope3() {
   A(); B();
}
void scope4() {
   B(); D(); scope1();
}
void scope5() {
   B(); D(); A();
}
void scope6() {
   B(); D();
}
```


Checking derived lock functions

```
/* 2.4.0:drivers/sound/cmpci.c:cm_midi_release: */
lock_kernel();
if (file->f_mode & FMODE_WRITE) {
      add_wait_queue(&s->midi.owait, &wait);
         ...
         if (file->f_flags & O_NONBLOCK) {
             remove_wait_queue(&s->midi.owait, &wait);
             set_current_state(TASK_RUNNING);
             return -EBUSY;
… unlock_kernel();
```


Summary

- Key ideas:
  - check code beliefs: find errors without knowing truth
  - beliefs code MUST have: Contradictions = errors
  - beliefs code MAY have: check as MUST beliefs and rank errors by belief confidence
- Assumptions for MAY beliefs: Majority of the code is correct.

Additional Reading

- Bugs as Deviant Behaviours:A general approach to inferring errors in systems code
  - Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, Benjamin Chelf Stanford University, SOSP'01
- Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions (Best Paper) (OSDI'00)
  - http://www.stanford.edu/~engler/mc-osdi.pdf
- eXplode: a Lightweight, General System for Finding Serious Storage System Errors (OSDI'06)
  - http://www.stanford.edu/~engler/explode-osdi06.pdf

Graph Coverage for Specification (Model-Based Testing)

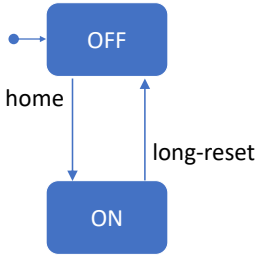
Chengnian Sun
cnsun@uwaterloo.ca

*Slides adapted from sttp.site

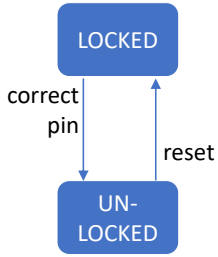
Finite State Machines (FSMs)

- A model that describes the software system by describing its states.
- A system often has multiple states and various transitions between these states.
- The state machine model uses these states and transitions to illustrate the system's behavior.

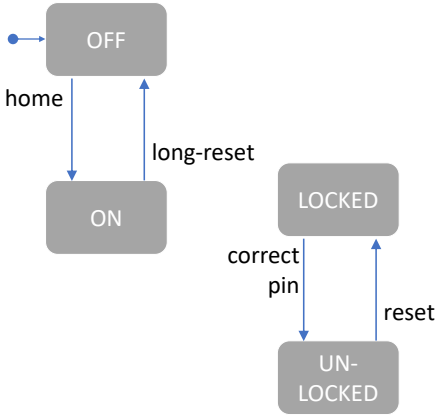
Example – On vs. Off



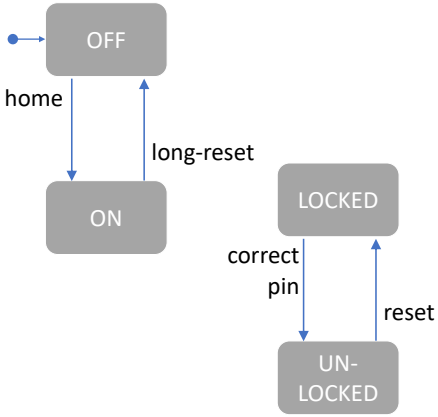
Example – Unlocking

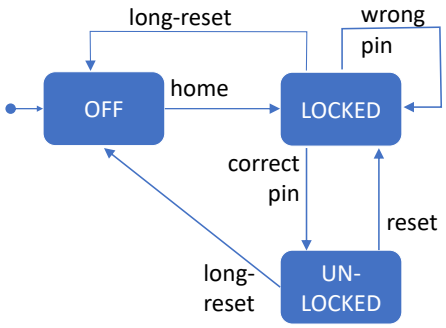


On/Off + Unlocking



On/Off + Unlocking

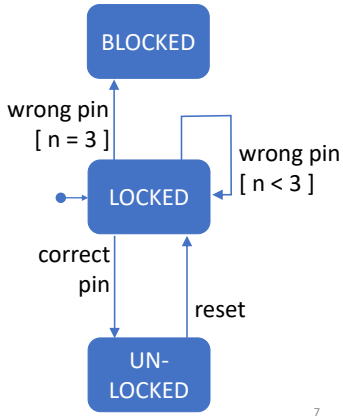




Conditional Transitions (precondition)

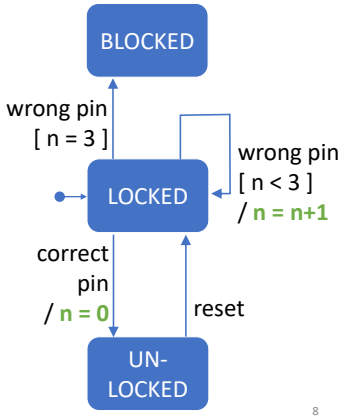
The condition is the precondition. The transition can be enabled if the condition is true.

"n" is a state variable, recording the number of unsuccessful trials.



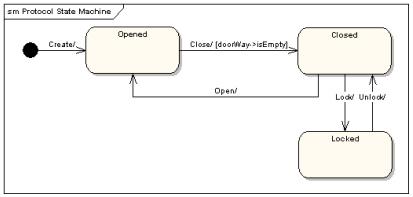
Actions / Effects

The action is executed if the corresponding transition is taken.



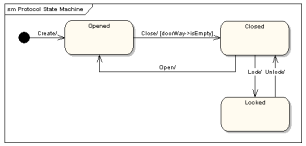
UML State Machine Diagrams

- A state machine diagram models the behavior of a single object, component, program or a system, specifying the sequence of events that an object/component/program/system goes through during its lifetime in response to events.
- Imposes constraints on ordering of events
- The figure is an example shows the states that a door goes through its lifetime.

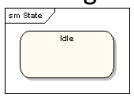


https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html

UML State Machine Diagrams

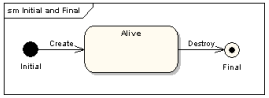


- States: a state is denoted by a round-cornered rectangle with the name of the state written inside it.
  - reflects as much of a system's history
  - as needed to determine current allowed behavior



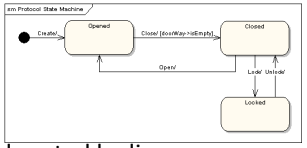
STATE

- Initial and final states: the initial state is denoted by a filled circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name. (A state machine diagram must have an initial state, but not necessarily a final state)

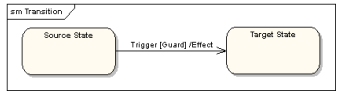


https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html

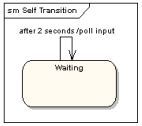
UML State Machine Diagrams



- Transitions: Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger/event, a guard/condition and an effect/action.



  - Trigger: the cause of the transition, which could be a signal, an event, a changed in some condition, or the passage of time.
  - Guard/Condition: a condition which must be true in order for the trigger to cause the transition.
  - Effect: an action which will be invoked directly on the object that owns the state machine as a result of the transition
- Self-Transition: a state can have a transition that returns to itself.

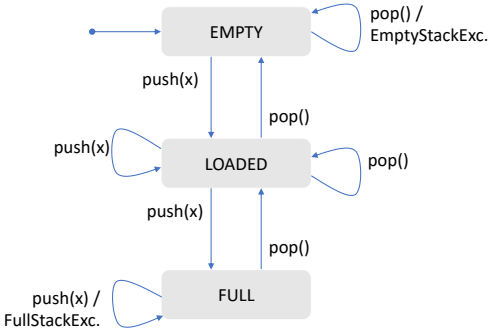


https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html

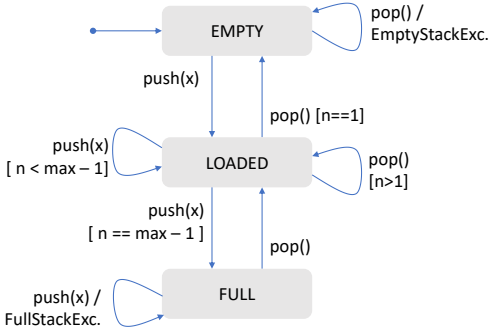
Exercise – Draw a FSM for a Bounded Stack (bound>1)

- throw EmptyStackExc if you call pop() on an empty stack.
- throw FullStackExc if you call push(x) on a full stack.
- pop() removes one element from the stack
- push(x) adds one element to the stack

Exercise – Draw a FSM for a Bounded Stack (bound>1)

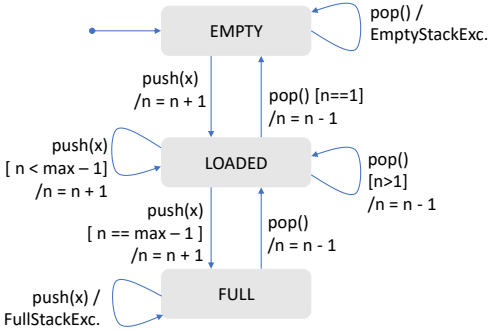


Exercise – Draw a FSM for a Bounded Stack (bound>1)

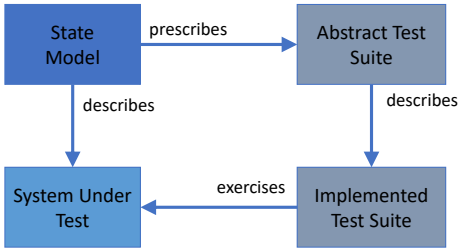


Exercise – Draw a FSM for a Bounded Stack (bound>1)

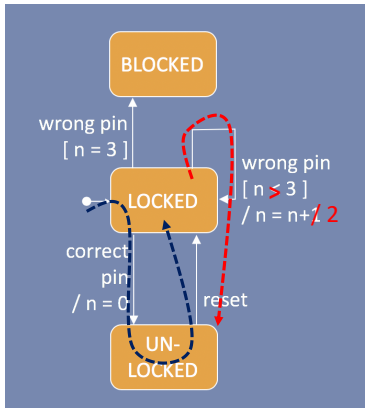
With Actions.



Model-Based Testing



Fault Model

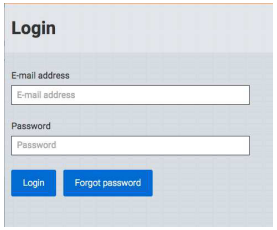


Coverage Criteria

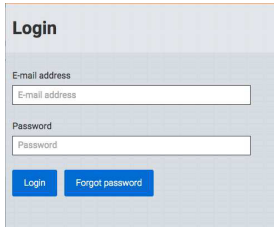
- State Coverage
  - reach every state
- Transition Coverage
  - exercise every transition
- Path Coverage
  - exercise sequence of transitions

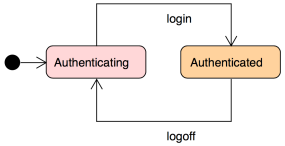
- Graph Coverage?

Example – Web App Authentication

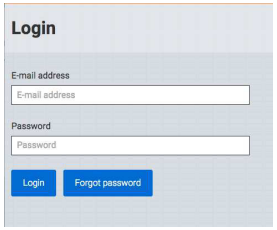


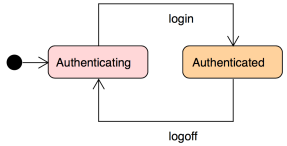
Example – Web App Authentication





Example – Web App Authentication





Testing Authentication

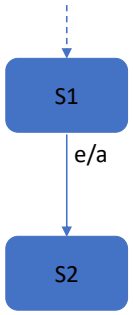
1. Startup
2. Check you are in "Authenticating"
3. Enter valid login
4. Check you are in "Authenticated"
5. Hit logoff
6. Check you are in "Authenticating"

Testing One Transition

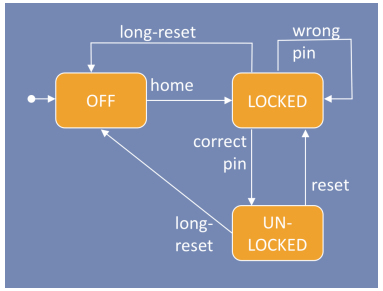
- Bring program in state *S1*
- check you are indeed in state *S1*

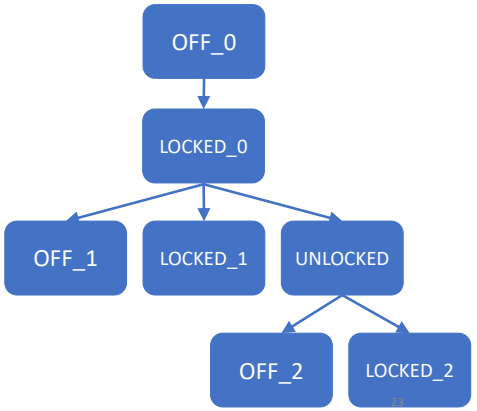
- trigger event *e*

- check effect of action *a* on *e*
- check you have arrived in state *S2*

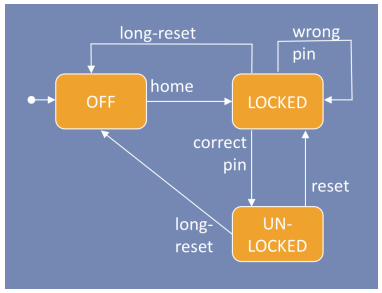


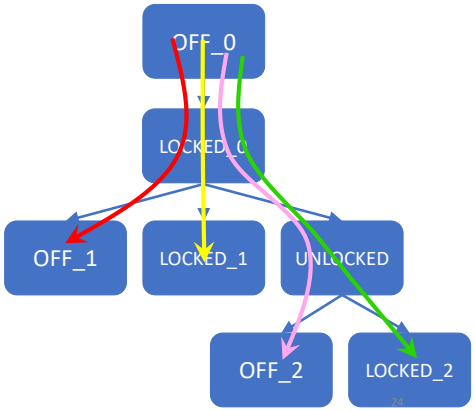
Transition Tree



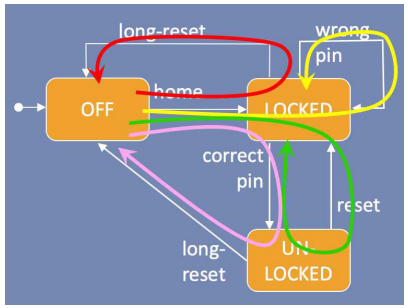


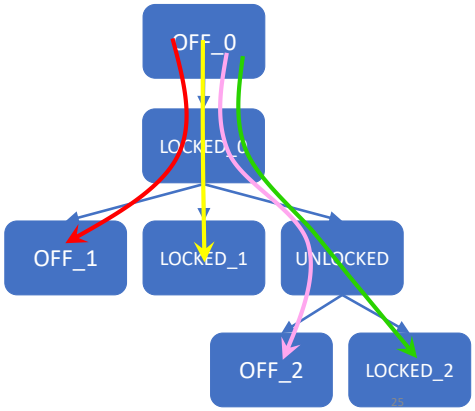
Tests from the Transition Tree





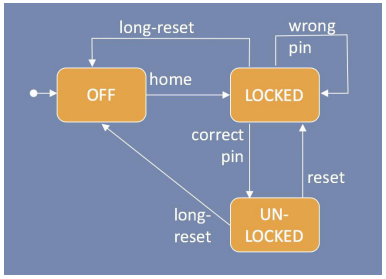
Tests from the Transition Tree





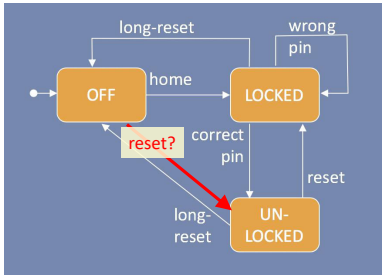
Sneak Path Testing

- Model describes expected behaviors.
- What about behaviors not in the model?



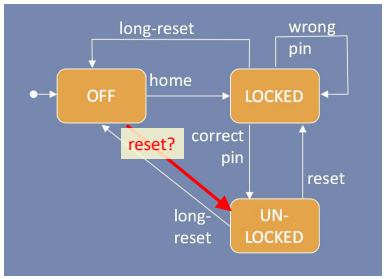
Sneak Path Testing

- Model describes expected behaviors.
- What about behaviors not in the model?



Transition Table

STATES	events				
	home	long-reset	reset	wrong-pin	correct-pin
OFF	LOCKED				
LOCKED		OFF		LOCKED	UNLOCKED
UNLOCKED		OFF	LOCKED		



Sneak Path Testing

STATES	events				
	home	long-reset	reset	wrong-pin	correct-pin
OFF	LOCKED				
LOCKED		OFF		LOCKED	UNLOCKED
UNLOCKED		OFF	LOCKED		

- Create transition table
- Determine effects of empty cells
  - Default: do nothing
- Test each empty cell:
  - Default action
  - No state change

- Verifies that illegal transitions cannot occur

Implementing State-Based Tests

- Class as a state machine

- Inspection methods
  - See current state

- Trigger methods
  - Transition to next state

- Test scenario
  - sequence of triggers and inspects

Implementing State-Based Tests

- Class as a state machine

- Inspection methods
  - See current state

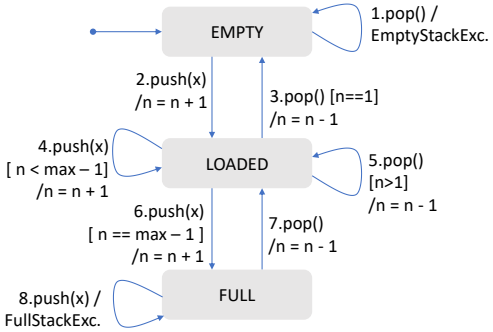
- Trigger methods
  - Transition to next state

- Test scenario
  - sequence of triggers and inspects

BoundedStack
int size()
int capacity()
void push(int v) throws FullStackException
int pop() throws EmptyStackException

Exercise – Bounded Stack

- (a) Draw the transition table



Exercise – Bounded Stack – Transition Table

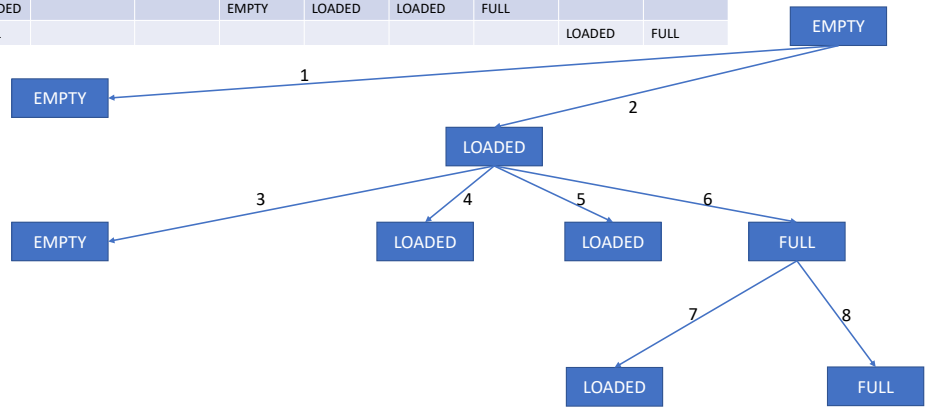
	1	2	3	4	5	6	7	8
	pop() /EmptyStackExc.	push(x) /n=n+1	pop() [n==1] /n=n-1	push(x) [n<max-1] /n=n+1	pop() [n>1] /n=n-1	push(x) [n==max-1] /n=n+1	pop() /n=n-1	push(x) /FullStackExc.
EMPTY	EMPTY	LOADED						
LOADED			EMPTY	LOADED	LOADED	FULL		
FULL							LOADED	FULL

Exercise – Bounded Stack – Transition Tree

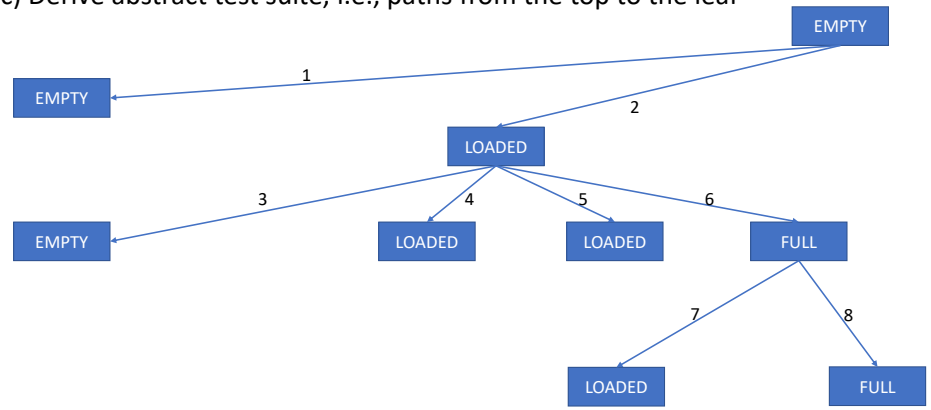
- (a) Draw the transition table
- (b) Draw the transition tree

	1	2	3	4	5	6	7	8
	pop() /EmptyStackExc.	push(x) /n=n+1	pop() [n==1] /n=n-1	push(x) [n<max-1] /n=n+1	pop() [n>1] /n=n-1	push(x) [n==max-1] /n=n+1	pop() /n=n-1	push(x) /FullStackExc.
EMPTY	EMPTY	LOADED						
LOADED			EMPTY	LOADED	LOADED	FULL		
FULL							LOADED	FULL

	1	2	3	4	5	6	7	8
	pop() /EmptyStackExc.	push(x) /n=n+1	pop() [n==1] /n=n-1	push(x) [n<max-1] /n=n+1	pop() [n>1] /n=n-1	push(x) [n==max-1] /n=n+1	pop() /n=n-1	push(x) /FullStackExc.
EMPTY	EMPTY	LOADED						
LOADED			EMPTY	LOADED	LOADED	FULL		
FULL							LOADED	FULL

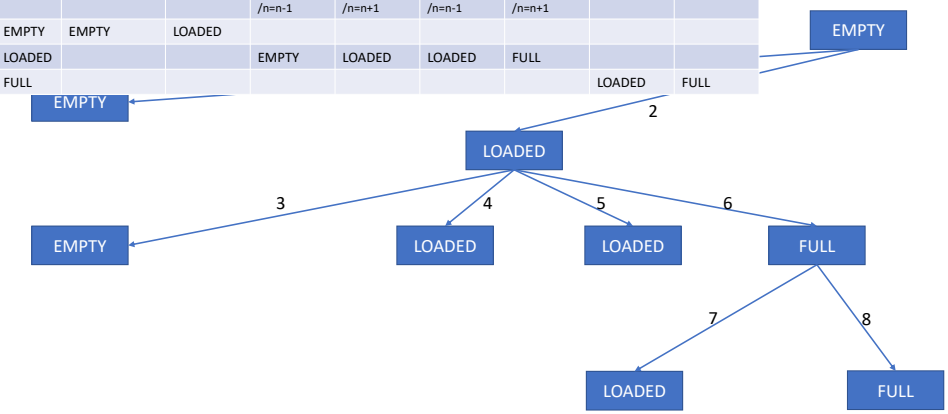


- (a) Draw the transition table
- (b) Draw the transition tree
- (c) Derive abstract test suite, i.e., paths from the top to the leaf



- (d) Implement abstract test suite.

	1	2	3	4	5	6	7	8
	pop() /EmptyStackExc.	push(x) /n=n+1	pop() [n==1] /n=n-1	push(x) [n<max-1] /n=n+1	pop() [n>1] /n=n-1	push(x) [n==max-1] /n=n+1	pop() /n=n-1	push(x) /FullStackExc.
EMPTY	EMPTY	LOADED						
LOADED			EMPTY	LOADED	LOADED	FULL		
FULL							LOADED	FULL



EMPTY –(1)-> EMPTY

```
@Test
public void test() {
  BoundedStack s = new BoundedStack(1);
  assertEquals(s.size(), 0); // EMPTY State
  try {
    s.pop();
    Assert.fail("Not reachable.");
  } catch (EmptyStackException e) {
    // Observe the effect of the action.
  }
  assertEquals(s.size(), 0); // EMPTY State
}
```


EMPTY –(2)-> LOADED—(3)->EMPTY

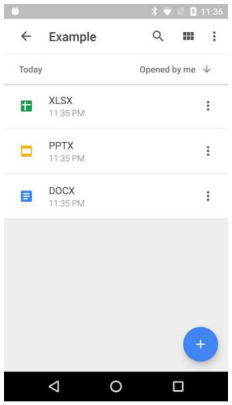
```
@Test
public void test() {
  BoundedStack s = new BoundedStack(2);
  assertEquals(s.size(), 0); // EMPTY State
  s.push(1);
  assertEquals(s.size(), 1); // LOADED State
  int v = s.pop();
  assertEquals(v, 1);
  assertEquals(s.size(), 0); // EMPTY State
}
```

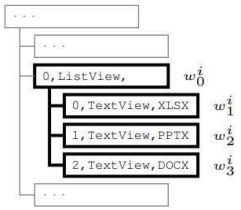
APE: Automated Model-Based GUI Testing of Android Apps

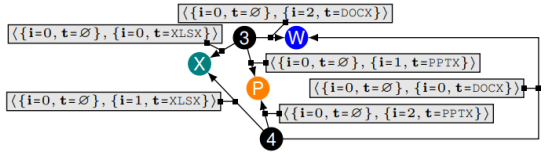
- Dynamically build a model of an Android app
- Based on the model, generate user events (clicks, swipes, drags) to test the app.
- Refine the model during testing as more knowledge is gained.

Practical GUI Testing of Android Applications via Model Abstraction and Refinement

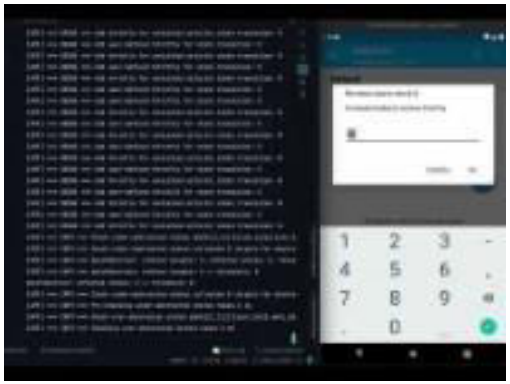
APE: Automated Model-Based GUI Testing of Android Apps







APE: Automated Model-Based GUI Testing of Android Apps



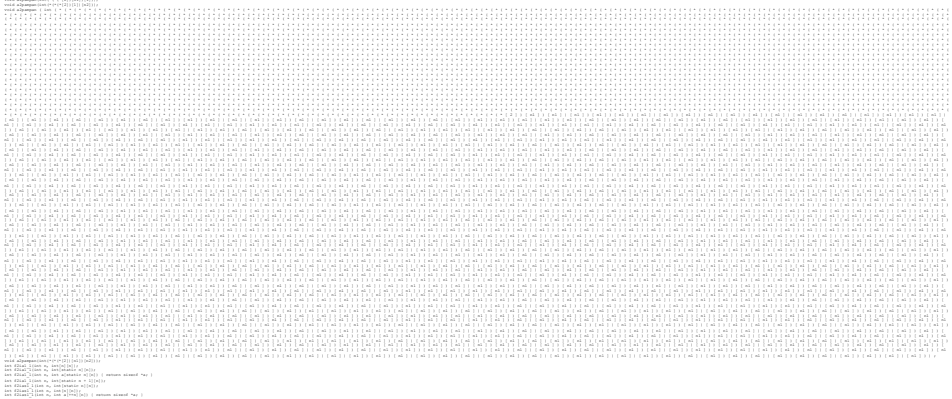
https://youtu.be/clJBXnNGseA

Test Input Reduction

Chengnian Sun
cnsun@uwaterloo.ca

a C program

13,003 tokens



a C program

13,003 tokens

```
$ gcc -v
gcc version 12.0.0 20210514 (experimental)

$ gcc t.c
internal compiler error: in build_attr_access_from_parms, at
c-family/c-attribs.c:5038
```


Can we report this bug directly with this program?

```
$ gcc -v
gcc version 12.0.0 20210514 (experimental)

$ gcc t.c
internal compiler error: in build_attr_access_from_parms, at
c-family/c-attribs.c:5038
```


Can we report this bug directly with this program?

- This program is large
  - contains bug-irrelevant code fragments → complicates debugging

Can we report this bug directly with this program?

NO

- This program is large
  - contains bug-irrelevant code fragments → complicates debugging

GCC Bugs - GNU Project - Fre

gcc.gnu.org/bugs/#detailed

Incognito

Detailed bug reporting instructions

Please refer to the next section when reporting bugs in GNAT, the Ada compiler, or to the one after that when reporting bugs that appear when using a precompiled header.

In general, all the information we need can be obtained by collecting the command line below, as well as its output and the preprocessed file it generates.

```
gcc -v -save-temps all-your-options source-file
```

The preprocessed source is the *basic* requirement to fix a bug. However, providing a minimal testcase increases the chances of getting your bug fixed.

a C program

13,003 tokens

a C program

13,003 tokens

382 tokens

```
n1; a2pampan (int
((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((([][n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n
1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1
])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])
[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1])[n1]);
```

reported: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100619

A Rust Program -- https://github.com/rust-lang/rust/issues/78651

```rust
#![allow(unused_must_use)]
fn main() {
    if false {
        test();
    }
}
fn test() {
    let rx = Err::<Vec<usize>, u32>(1).into_future();
    rx.map(|l: Vec<usize>| stream::iter(l.into_iter().map(|i| Ok(i))))
        .flatten_stream()
        .chunks(50)
        .buffer_unordered(5);
}
use future::{Future, IntoFuture};
mod future {
    use std::result;
    use {stream, Stream};
    pub trait Future {
        type Item;
        type Error;
        fn map<F, U>(self, _: F) -> Map<Self, F>
        where
            F: FnOnce(Self::Item) -> U,
            Self: Sized,
        {
            panic!()
        }
        fn flatten_stream(self) -> FlattenStream<Self>
        where
            <Self as Future>::Item: stream::Stream<Error = Self::Error>,
            Self: Sized,
        {
            panic!()
        }
    }
    pub trait IntoFuture {
        type Future: Future<Item = Self::Item, Error = Self::Error>;
        type Item;
        type Error;
        fn into_future(self) -> Self::Future;
    }
    impl<F: Future> IntoFuture for F {
        type Future = F;
        type Item = F::Item;
        type Error = F::Error;
        fn into_future(self) -> F {
            panic!()
        }
    }
    impl<T, E> IntoFuture for result::Result<T, E> {
        type Future = FutureResult<T, E>;
        type Item = T;
        type Error = E;
        fn into_future(self) -> FutureResult<Err, E> {
            panic!()
        }
    }
    pub struct Map<A, F> {
        _a: (A, F),
    }
```

A Rust Program -- https://github.com/rust-lang/rust/issues/78651

958 tokens

```
$ rustc t.rs

thread 'rustc' panicked at 'assertion failed: !self.substitutions.is_empty()',
compiler/rustc_errors/src/lib.rs:173:9 note: run with `RUST_BACKTRACE=1` environment
variable to display a backtrace error:

internal compiler error: unexpected panic note: the compiler unexpectedly panicked.
this is a bug.

note: we would appreciate a bug report: https://github.com/rust-
lang/rust/issues/new?labels=C-bug%2C+I-ICE%2C+T-compiler&template=ice.md

note: rustc 1.49.0-nightly (4f7612ac1 2020-10-31) running on x86_64-unknown-linux-gnu
_attr_access_from_parms, at c-family/c-attribs.c:5038
```


Issue: Internal Compiler Error

Create a report for an internal compiler error in rustc. If this doesn't look right, choose a different type.

Title

Write Preview

```
<!--
Thank you for finding an Internal Compiler Error! If possible, try to provide
a minimal verifiable example. You can read "Rust Bug Minimization Patterns" for
how to create smaller examples.

http://blog.pnkfx.org/blog/2019/11/18/rust-bug-minimization-patterns/

-->

### Code

```Rust
<code>
```
```

A Rust Program -- https://github.com/rust-lang/rust/issues/78651

958 tokens → 17 tokens

```
use std::result;
impl result {
    fn into_future() -> Err {}
}
```


Test Input Simplification

- A test input triggers a bug in a program, then we need to simplify it,
  - by removing parts irrelevant to the bug
  - to facilitate understanding and debugging
- Motivation
  - Less debugging burden
    - Irrelevant information ≡ less noise/distraction
    - Smaller test input → smaller program states and shorter execution paths
  - De-duplication
    - Bug reports with similar simplified test inputs tend to be duplicates

Expected Behavior:

- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Solutions:

a) `for (int i = x.length - 1; i >= 0; --i)`
b) x=null, y=_
c) x=[] or y appears in x at a non-zero index
   - x=[], y=_
     i>0 and i>=0 are both false
   - x=[2, 3, 5], y=3
     i>0 and i>=0 are both true
d) x.length > 0 and y is not in x
   - no failure as the result is always -1
   - but does not execute `if (x[0] == y)`

Questions:

a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Expected Behavior:

- Returns the index of the last element in ‘x’ that equals ‘y’.
- Returns -1 if no such element exists
- Throws an exception if ‘x’ is null

```
public int lastIndexOf(int[] x, int y) {
  for (int i = x.length - 1; i > 0; --i) {
    if (x[i] == y) {
      return i;
    }
  }
  return -1;
}
```

Solutions:

a) `for (int i = x.length - 1; i >= 0; --i)`
b) x=null, y=_
c) x=[] or y appears in x at a non-zero index
   - x=[], y=_
     i>0 and i>=0 are both false
   - x=[2, 3, 5], y=3
     i>0 and i>=0 are both true
d) x.length > 0 and y is not in x
   - no failure as the result is always -1
   - but does not execute `if (x[0] == y)`

Questions:

a) identify the fault, and fix it
b) identify a test case that does not execute the fault
c) identify a test case that executes the fault, but does not result in an error state
d) identify a test case that results in an error, but not a failure
e) identify the first error state for the following test case. Be sure to describe the complete program state.
   - `assert lastIndexOf(new int[]{2, 3, 5}, 2) == 0`

Difficulties debugging the following inputs:

1. x=[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], y=0
2. x=[0], y=0
3. X=[0, 1], y=0

A Real Story in 1999

- The Mozilla browser listed 370+ open bugs
  - Not simplified, containing irrelevant information
  - Overwhelming work for Mozilla engineers
  - Possible duplicates
- Mozilla BugAThon: a call out for volunteers to simplify them

> When you’ve cut away much HTML, CSS, and JavaScript as you can, and cutting away any more causes the bug to disappear, you’re done.
>
> -- Mozilla BugAThon call

A Real Story in 1999 – An Example Report

```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION
VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

File → Printing → Segmentation Fault in Mozilla

A Real Story in 1999 – An Example Report

<SELECT>

File → Printing → Segmentation Fault in Mozilla

How to Simplify the Large Input to <SELECT>

- How?

How to Simplify the Large Input to <SELECT>

- How?
- Delta Debugging
  - A technique based on **binary search**

Delta Debugging – A Simplified Example

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

Delta Debugging – Iteration 1

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

- Iteration 1, partition the input into halves

Delta Debugging – Iteration 1

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

- Iteration 1, partition the input into halves
  - `1.PASS: <SELECT NAME="priori`
  - `2.PASS:                    ty" MULTIPLE SIZE=7>`

Delta Debugging – Iteration 2

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

- Iteration 2, partition the input into quarters

Delta Debugging – Iteration 2

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

- Iteration 2, partition the input into quarters
  - `1.PASS: <SELECT NA`

Delta Debugging – Iteration 2

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>

Delta Debugging – Iteration 2

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>
  - 3.PASS: ME="priori

Delta Debugging – Iteration 2

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>
  - 3.PASS: ME="priori
  - **4.FAIL**: <SELECT NA ty" MULTIPLE SIZE=7>

Delta Debugging – Iteration 2

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>
  - 3.PASS: ME="priori
  - **4.FAIL**: <SELECT NA ty" MULTIPLE SIZE=7>
  - **5.FAIL**: <SELECT NA. LE SIZE=7>

Delta Debugging – Iteration 2

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>
  - 3.PASS: ME="priori
  - **4.FAIL**: <SELECT NA ty" MULTIPLE SIZE=7>
  - **5.FAIL**: <SELECT NA LE SIZE=7>
  - 6.PASS: <SELECT NA

Delta Debugging – Iteration 2

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>
  - 3.PASS: ME="priori
  - **4.FAIL**: <SELECT NA ty" MULTIPLE SIZE=7>
  - **5.FAIL**: <SELECT NA LE SIZE=7>
  - 6.PASS: <SELECT NA
  - 7.PASS: LE SIZE=7>

Delta Debugging – Iteration 2

<SELECT NA LE SIZE=7>

- Iteration 2, partition the input into quarters
  - 1.PASS: <SELECT NA
  - 2.PASS: ME="priority" MULTIPLE SIZE=7>
  - 3.PASS: ME="priori
  - **4.FAIL**: <SELECT NA ty" MULTIPLE SIZE=7>
  - **5.FAIL**: <SELECT NA LE SIZE=7>
  - 6.PASS: <SELECT NA
  - 7.PASS: LE SIZE=7>

Delta Debugging – Iteration 3

<SELECT NALE SIZE=7>

- Iteration 3, partition the input into halves
  - 1.PASS: <SELECT NA
  - 2.PASS: LE SIZE=7>

Delta Debugging – Iteration 4

<SELECT NALE SIZE=7>

- Iteration 3, partition the input into quarters
  - 1.PASS: CT NALE SIZE=7>
  - 2.PASS: <SELE LE SIZE=7>
  - **3.FAIL**: <SELECT NA ZE=7>
  - 4.PASS: <SELECT NA
  - 5.PASS: CT NA ZE=7>
  - 6.PASS: <SELE ZE=7>

Delta Debugging – Eventually

- Repeat iterations until no further progress.

<SELECT>

A Simplified Algorithm

(1) Divide a string S equally into $\Delta_1, \Delta_2, \ldots \Delta_n$ and the respective complements are $\nabla_1, \nabla_2, \ldots, \nabla_n$.
(2) Test each $\Delta_1, \Delta_2, \ldots \Delta_n$ and $\nabla_1, \nabla_2, \ldots, \nabla_n$.
 if (all pass) {
  n=2n;
  if (n>|s|) return the most recent failure inducing substring.
  else goto (1)
} else if (Δ_t fails) {
  n=2; s= Δ_t
  if (|s|==1) return s
  else goto (1)
} else { /* ∇_t fails */
  s= ∇_t ; n=n-1; goto (1);
}

Test all candidates in order. Check whether a candidate triggers the bug immediately before checking the remaining ones.

Input: A B C D E *

Property of interest: A,*

(1) Divide a string S equally into $\Delta_1, \Delta_2, \ldots \Delta_n$ and the respective complements are $\nabla_1, \nabla_2, \ldots, \nabla_n$.
(2) Test each $\Delta_1, \Delta_2, \ldots \Delta_n$ and $\nabla_1, \nabla_2, \ldots, \nabla_n$.
 if (all pass) {
  n=2n;
  if (n>|s|) return the most recent failure inducing substring.
  else goto (1)
} else if (Δ_t fails) {
  n=2; s= Δ_t
  if (|s|==1) return s
  else goto (1)
} else { /* ∇_t fails */
  s= ∇_t ; n=n-1; goto (1);
}

```
if (n < |s| && 2*n > |s|) {n = |s|;}
else {n = 2 * n;}
```

Global Minimality

- A test case $c \subseteq c_F$ is called the global minimum of c_F if
  - $\forall c' \subseteq c_F, |c'| < |c| \Rightarrow test(c') \neq F$
- The smallest set of changes to trigger the program to fail
- Searching for global minimal is exponential
- NP-complete (black-box optimization problem).
  - Can be reduced to a hitting set problem

Local Minimality

- A test case $c \subseteq c_F$ is called the local minimum of c_F if
  - $\forall c' \subseteq c, test(c') \neq F$
- A test case $c \subseteq c_F$ is n-minimal if
  - $\forall c' \subseteq c, |c| - |c'| \leq n \Rightarrow test(c') \neq F$
- The delta debugging algorithm finds 1-minimal test case
  - Example: AAAABBBBCCCC, program fails when |A|=|B|=|C|>0

Monotonicity

- The string *s* induces a failure.
- The super string of *s* always induces the failure.
- Delta debugging is not effective for cases without monotonicity.

A Real Story in 1999 – An Example Report

<SELECT>

File → Printing → Segmentation Fault in Mozilla

Case Studies

- The following C program causes GCC to crash (755 characters)

```
#define SIZE 20
double mult(double z[], int n) {
 int i , j ;
 i = 0;
 for (j = 0; j < n; j++) {
 i = i + j + 1;
 z[i] = z[i] *(z[0]+1.0);
 return z[n];
}
void copy(double to[], double from[], int count) {
 int n = count + 7) / 8;
 switch(count % 8) do {
 case 0: *to++ = *from++;
 case 7: *to++ = *from++;
 case 6: *to++ = *from++;
 case 5: *to++ = *from++;
 case 4: *to++ = *from++;
 case 3: *to++ = *from++;
 case 2: *to++ = *from++;
 case 1: *to++ = *from++;
 } while ( --n > 0);
 return mult(to, 2);
}
int main(int argc, char *argv[]) {
 double x[SIZE], y[SIZE];
 double *px = x;
 while (px < x + SIZE)
 *px++ = (px - x) * (SIZE + 1.0);
 return copy(y, x, SIZE);
}
```


Case Studies

- The following C program causes GCC to crash (755 characters)

```
#define SIZE 20
double mult(double z[], int n) {
 int i , j ;
 i = 0;
 for (j = 0; j < n; j++) {
 i = i + j + 1;
 z[i] = z[i] *(z[0]+1.0);
 return z[n];
}
void copy(double to[], double from[], int count) {
 int n = count + 7) / 8;
 switch(count % 8) do {
 case 0: *to++ = *from++;
 case 7: *to++ = *from++;
 case 6: *to++ = *from++;
 case 5: *to++ = *from++;
 case 4: *to++ = *from++;
 case 3: *to++ = *from++;
 case 2: *to++ = *from++;
 case 1: *to++ = *from++;
 } while ( --n > 0);
 return mult(to, 2);
}
int main(int argc, char *argv[]) {
 double x[SIZE], y[SIZE];
 double *px = x;
 while (px < x + SIZE)
 *px++ = (px - x) * (SIZE + 1.0);
 return copy(y, x, SIZE);
}
```

Delta debugging minimizes it to 77 characters.

```
t(double z[],int n){
 int i,j;
 for(;;){
  i=i+j+1;z[i]=z[i]* (z[0]+0);
 }
 return[n];
}
```

If a single character is removed from the reduced input, the failure disappears.

Problems with Delta Debugging?

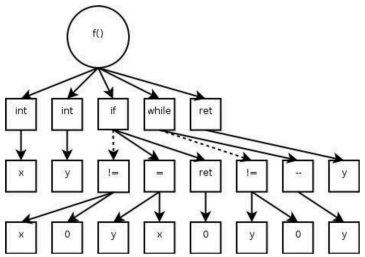
- Speed: slow
  - Generate too many syntactically invalid inputs.
- Granularity:
  - Input → a list of characters
  - Input → a list of tokens
  - Input → a list of lines
- Redundant property tests
  - the same variant may be generated multiple times
- Size of minimized input: maybe still large
  - Simple transformations: only deletion
  - Example: ((a+b)), suppose `a+b` is failure-relevant

Hierarchical Delta Debugging

```
void f()
{
    int x;  int y;
    if (x!=0) { y= x; } else { return 0; }
    while (y!=0) { y--; }
    return y;
}
```

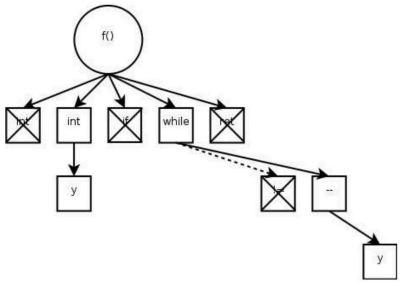
Hierarchical Delta Debugging

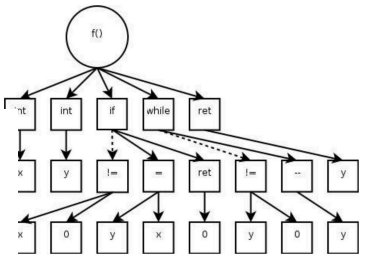
```
void f()
{
    int x;  int y;
    if (x!=0) { y= x; } else { return 0; }
    while (y!=0) { y--; }
    return y;
}
```



Hierarchical Delta Debugging

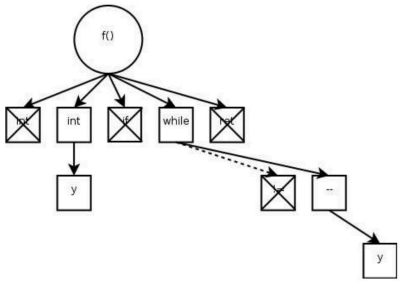
```
void f()
{
    int x;  int y;
    if (x!=0) { y= x; } else { return 0; }
    while (y!=0) { y--; }
    returr ⋯
}
```

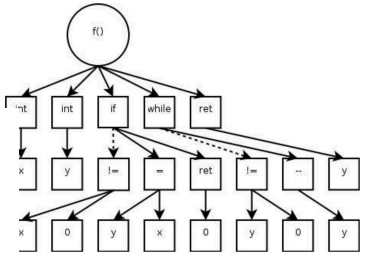




Hierarchical Delta Debugging

```
void f()
{
    int x;  int y;
    if (x!=0) { y= x; } else { return 0; }
    while (y!=0) { y--; }
    returr ⋯
}
```





1-tree minimality

HDD*: run HDD indefinitely until fixpoint

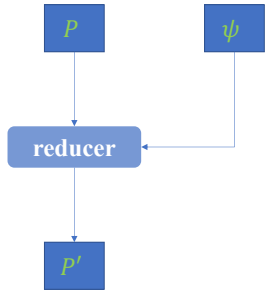
File	size (tokens)	bug report (id)	ddmin tests (# of tests)	HDD tests (# of tests)	HDD* tests (# of tests)	ddmin size (tokens)	HDD size (tokens)	HDD* size (tokens)
bug.c	277	unknown [20]	680	86	164	53	51	51
boom7.c	420	663	3727	144	304	102	57	19
cache.c	25011	1060	1743	191	327	62	61	58
cache-min.c	145	1060	1074	114	182	71	59	59

Syntax-Guided Language-Agnostic Program Reduction

Chengnian Sun
cnsun@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

program reduction

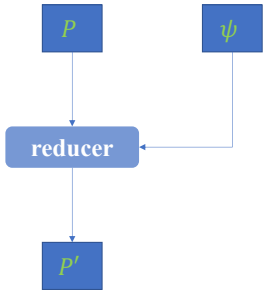
Input:
- P: a program
- ψ: a property, and $\psi(P)$



program reduction

Input:
- P: a program
- ψ: a property, and $\psi(P)$

Goal: remove ψ-irrelevant elements from P

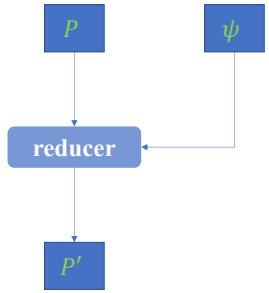


program reduction

Input:
- P: a program
- ψ: a property, and $\psi(P)$

Goal: remove ψ-irrelevant elements from P

Output:
- P': a minimized program from P, *s.t.* $\psi(P')$



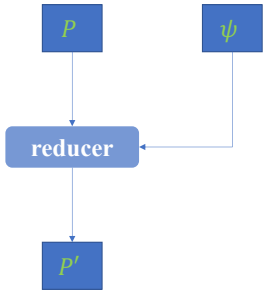
program reduction

specialized test input **minimization**

Input:
- P: a program
- ψ: a property, and $\psi(P)$

Goal: remove ψ-irrelevant elements from P

Output:
- P': a minimized program from P, *s.t.* $\psi(P')$



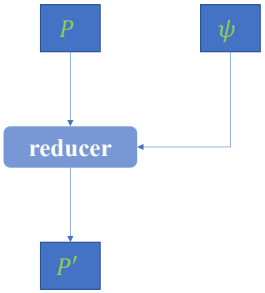
program reduction – an **important** problem

specialized test input **minimization**

Input:
- P: a program
- ψ: a property, and $\psi(P)$

Goal: remove ψ-irrelevant elements from P

ψ can be a bug in:
- static analyses
  - ❖ Frama-C, Soot, Wala



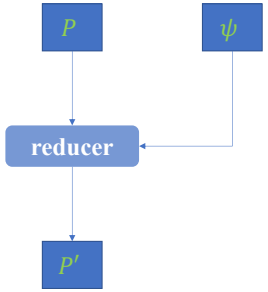
program reduction – an **important** problem

specialized test input **minimization**

Input:
- P: a program
- ψ: a property, and $\psi(P)$

Goal: remove ψ-irrelevant elements from P

ψ can be a bug in:
- static analyses
  - ❖ Frama-C, Soot, Wala
- refactoring engines
  - ❖ Eclipse, IntelliJ, Netbeans



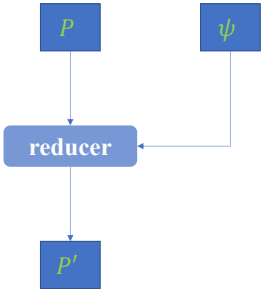
program reduction – an **important** problem

specialized test input **minimization**

Input:
- P: a program
- ψ: a property, and $\psi(P)$

Goal: remove ψ-irrelevant elements from P

ψ can be a bug in:
- static analyses
  - ❖ Frama-C, Soot, Wala
- refactoring engines
  - ❖ Eclipse, IntelliJ, Netbeans
- compilers
  - ❖ **GCC** (100k+ bugs), **LLVM** (50k+ bugs), **JVM, V8**

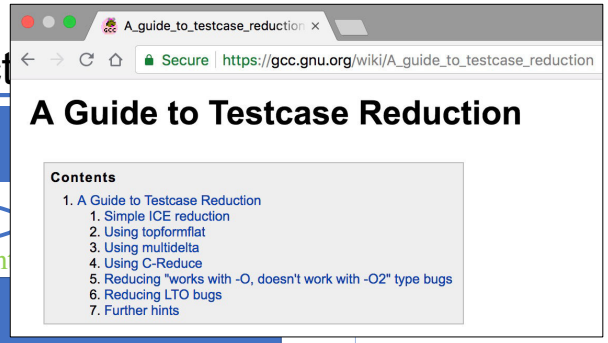


program reduct

Input:

- P: a program
- ψ: a property, and $\psi(P)$

Goal: re... ψ-irrelevan...



P'

ψ can be a bug in:

- static analyses
  - ❖ Frama-C, Soot, Wala
- refactoring engines
  - ❖ Eclipse, IntelliJ, Netbeans
- compilers
  - ❖ **GCC** (100k+ bugs), **LLVM** (50k+ bugs), **JVM, V8**

an example

P

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

```
$ gcc t.c ; ./a.out
1
Hello world!
End
```


an example

P

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

```
$ gcc t.c ; ./a.out
1
Hello world!
End
```

property: ψ

print "**Hello world!**"

an example

P

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

```
$ gcc t.c ; ./a.out
1
Hello world!
End
```

property: ψ

print "**Hello world!**"

ideal result

```
int main() {
 printf("Hello ");
 printf("world!\n");
 return 0;
}
```

```
$ gcc t.c ; ./a.out
Hello world!
```

state-of-the-art

- DD: Delta Debugging
  - binary search (delete lines each time and check ψ)
  - generate many **syntactically invalid** variants

ideal result

```
int main() {
 printf("Hello ");
 printf("world!\n");
 return 0;
}
```

DD & HDD

```
int main() {
 int a = 1;
 if (a) {
   printf("Hello ");
   printf("world!\n");
 }
 return 0;
}
```


an example

P

```
 1: int main() {
 2:  int a = 1;
 3:  if (a) {
 4:   printf("%d\n", a);
 5:   printf("Hello ");
 6:   printf("world!\n");
 7:   printf("End\n");
 8:  }
 9:  return 0;
10: }
```

```
$ gcc t.c ; ./a.out
1
Hello world!
End
```

property: ψ

print "**Hello world!**"

DD & HDD

```
int main() {
 int a = 1;
 if (a) {
   printf("Hello ");
   printf("world!\n");
 }
 return 0;
}
```


state-of-the-art

- DD: Delta Debugging
  - binary search (delete lines each time and check ψ)
  - generate many **syntactically invalid** variants
- HDD: Hierarchical Delta Debugging
  - parse a program into a tree
  - breadth-first search and apply DD on each level
  - better at program reduction
  - generate many **syntactically invalid** variants

ideal result

```
int main() {
 printf("Hello ");
 printf("world!\n");
 return 0;
}
```

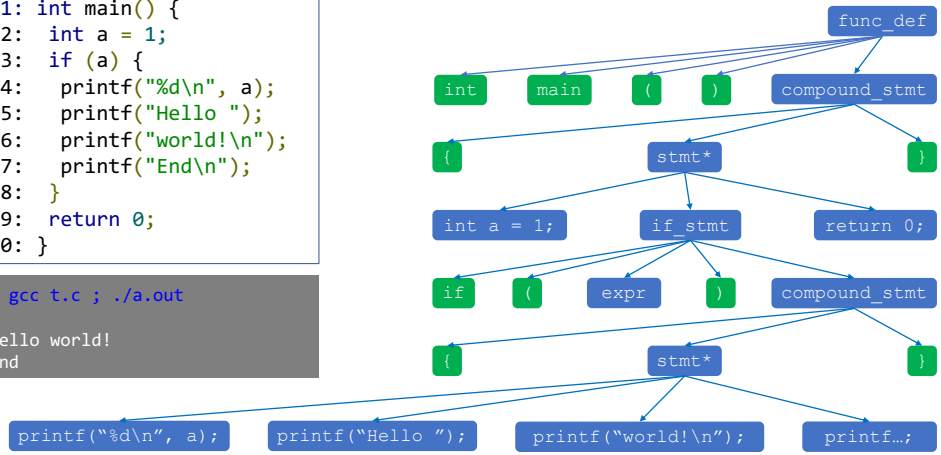
DD & HDD

```
int main() {
 int a = 1;
 if (a) {
   printf("Hello ");
   printf("world!\n");
 }
 return 0;
}
```

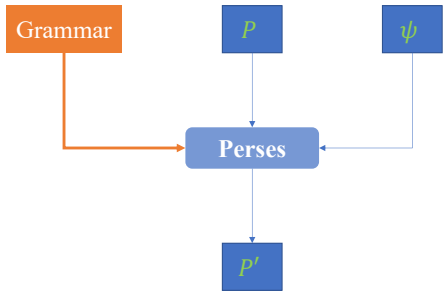

P

```
 1: int main() {
 2:  int a = 1;
 3:  if (a) {
 4:   printf("%d\n", a);
 5:   printf("Hello ");
 6:   printf("world!\n");
 7:   printf("End\n");
 8:  }
 9:  return 0;
10: }
```

```
$ gcc t.c ; ./a.out
1
Hello world!
End
```



Perses: fully syntax-guided



Perses: fully syntax-guided

- Avoid generating **syntactically invalid** variants

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

```
<func_def>      ::= <type> <identifier> '(' ')' <compound_stmt>
<compound_stmt> ::= '{' <stmt_star> '}'
<stmt_star>     ::= <stmt>*      // A list of zero or more statements
<stmt>          ::= <expr_stmt>
                  | <decl_stmt>
                  | <if_stmt>
                  | <compound_stmt>
<if_stmt>  ::= 'if' '(' <expr> ')' <stmt>
```


Perses: fully syntax-guided

- Avoid generating **syntactically invalid** variants
  - Most symbols are NOT removable
    - e.g., all symbols in `<func_def>`

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

```
<func_def>      ::= <type> <identifier> '(' ')' <compound_stmt>
<compound_stmt> ::= '{' <stmt_star> '}'
<stmt_star>     ::= <stmt>*      // A list of zero or more statements
<stmt>          ::= <expr_stmt>
                  | <decl_stmt>
                  | <if_stmt>
                  | <compound_stmt>
<if_stmt>  ::= 'if' '(' <expr> ')' <stmt>
```


Perses: fully syntax-guided

- Avoid generating **syntactically invalid** variants
  - Most symbols are NOT removable
    - e.g., all symbols in `<func_def>`
  - Except symbols that are quantified by *, + and ?
    - e.g., `printf` statements in body of `if` are removable

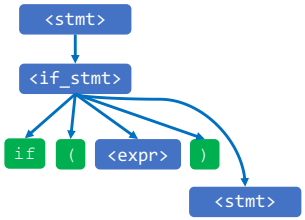
```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

```
<func_def>      ::= <type> <identifier> '(' ')' <compound_stmt>
<compound_stmt> ::= '{' <stmt_star> '}'
<stmt_star>     ::= <stmt>*      // A list of zero or more statements
<stmt>          ::= <expr_stmt>
                  | <decl_stmt>
                  | <if_stmt>
                  | <compound_stmt>
<if_stmt>  ::= 'if' '(' <expr> ')' <stmt>
```

Perses: fully syntax-guided

- Avoid generating **syntactically invalid** variants
- Enable more program transformations
  - e.g., replace with a **syntax-compatible** descendant

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```



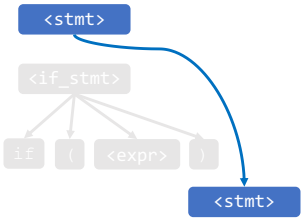

```
<func_def>      ::= <type> <identifier> '(' ')' <compound_stmt>
<compound_stmt> ::= '{' <stmt_star> '}'
<stmt_star>     ::= <stmt>*      // A list of zero or more statements
<stmt>          ::= <expr_stmt>
                  | <decl_stmt>
                  | <if_stmt>
                  | <compound_stmt>
<if_stmt>  ::= 'if' '(' <expr> ')' <stmt>
```

21

Perses: fully syntax-guided

- Avoid generating **syntactically invalid** variants
- Enable more program transformations
  - e.g., replace with a **syntax-compatible** descendant

```
int main() {
 int a = 1;
 if (a) {
   printf("%d\n", a);
   printf("Hello ");
   printf("world!\n");
   printf("End\n");
 }
 return 0;
}
```

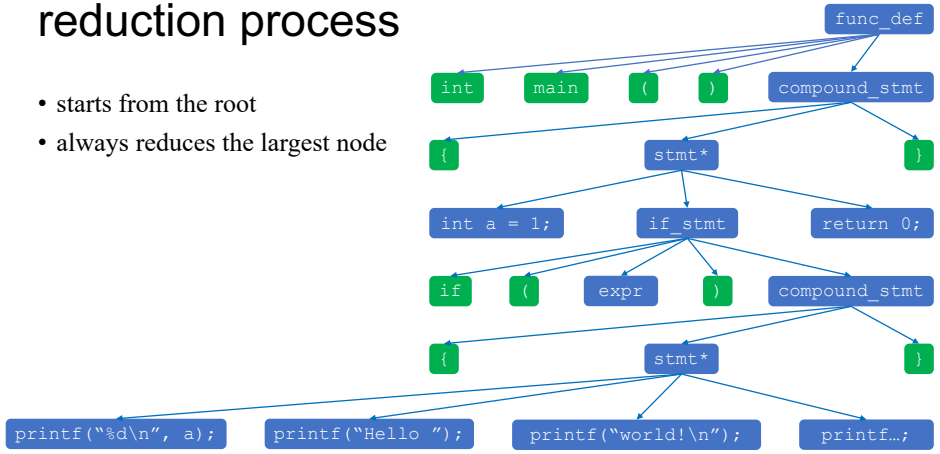



```
<func_def>      ::= <type> <identifier> '(' ')' <compound_stmt>
<compound_stmt> ::= '{' <stmt_star> '}'
<stmt_star>     ::= <stmt>*      // A list of zero or more statements
<stmt>          ::= <expr_stmt>
                  | <decl_stmt>
                  | <if_stmt>
                  | <compound_stmt>
<if_stmt>  ::= 'if' '(' <expr> ')' <stmt>
```

22

reduction process

- starts from the root
- always reduces the largest node



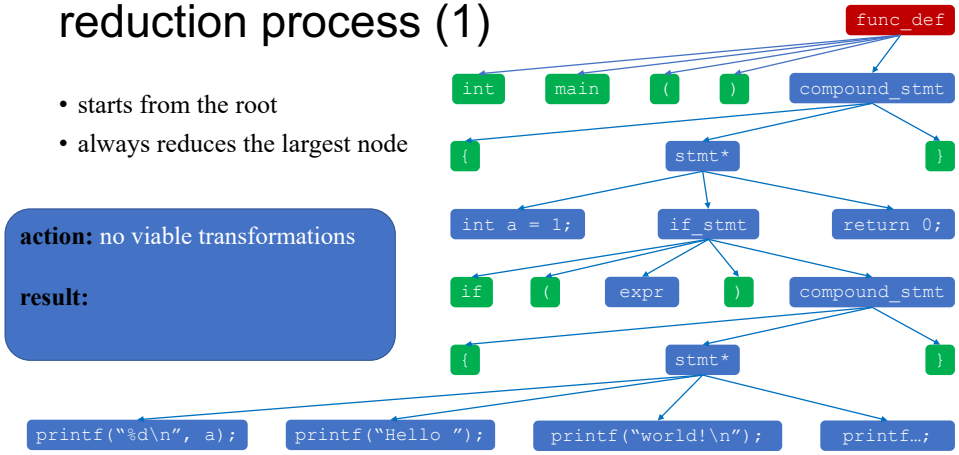
23

reduction process (1)

- starts from the root
- always reduces the largest node

action: no viable transformations

result:



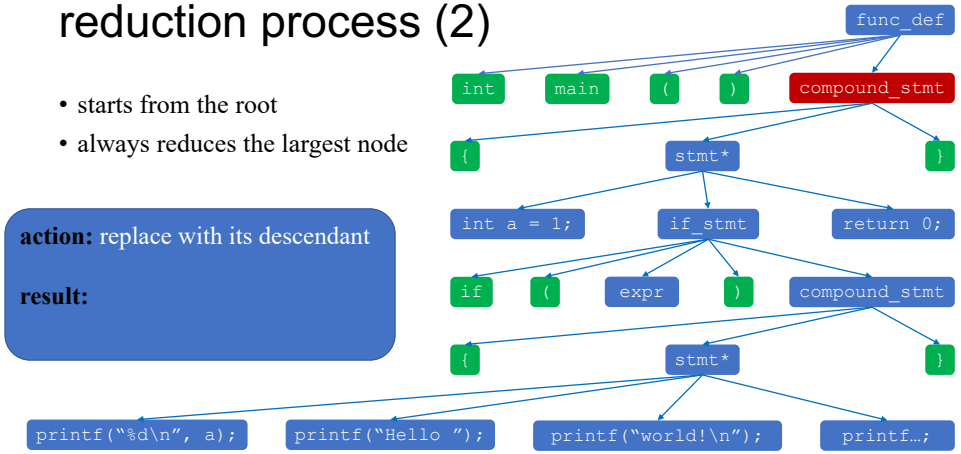
24

reduction process (2)

- starts from the root
- always reduces the largest node

action: replace with its descendant

result:

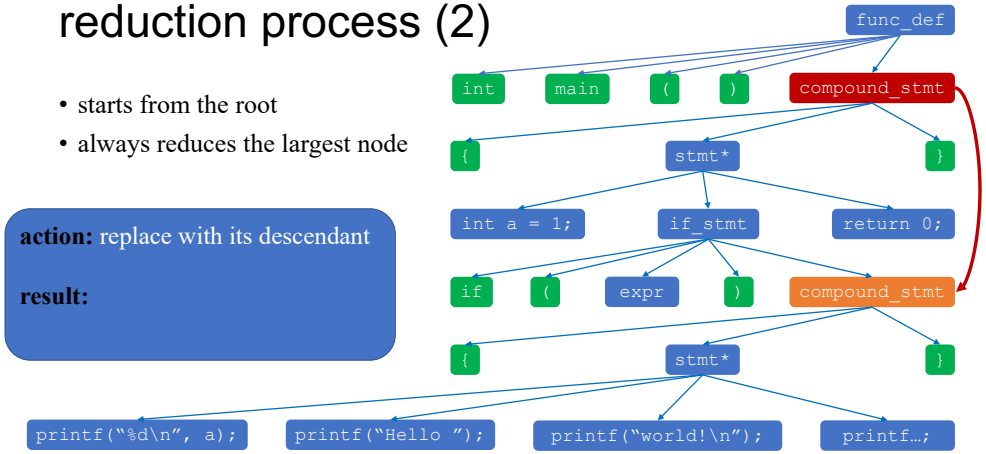


reduction process (2)

- starts from the root
- always reduces the largest node

action: replace with its descendant

result:

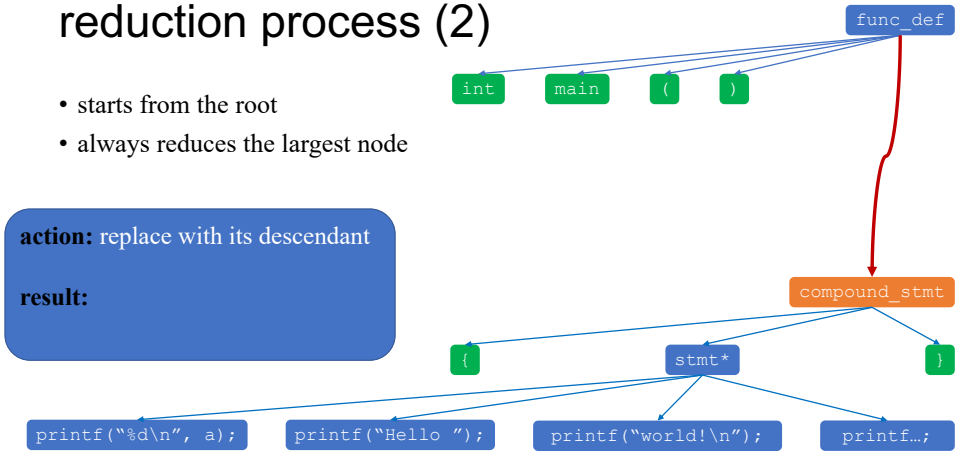


reduction process (2)

- starts from the root
- always reduces the largest node

action: replace with its descendant

result:

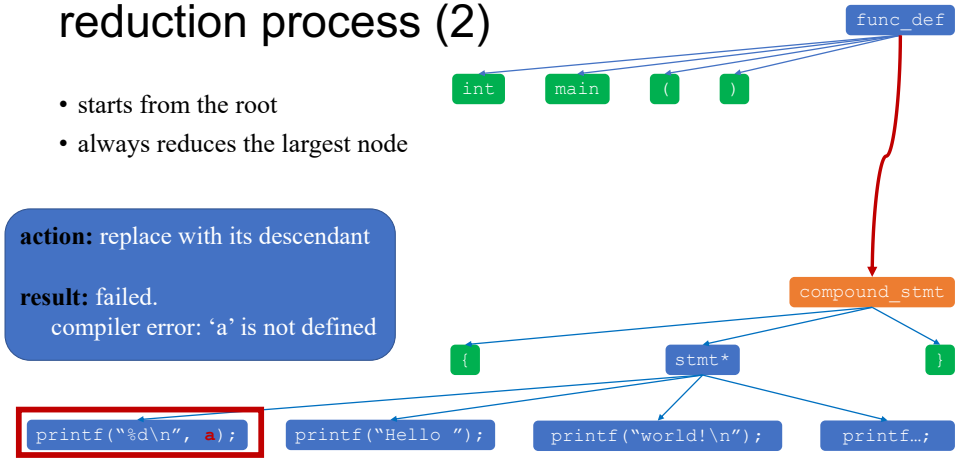


reduction process (2)

- starts from the root
- always reduces the largest node

action: replace with its descendant

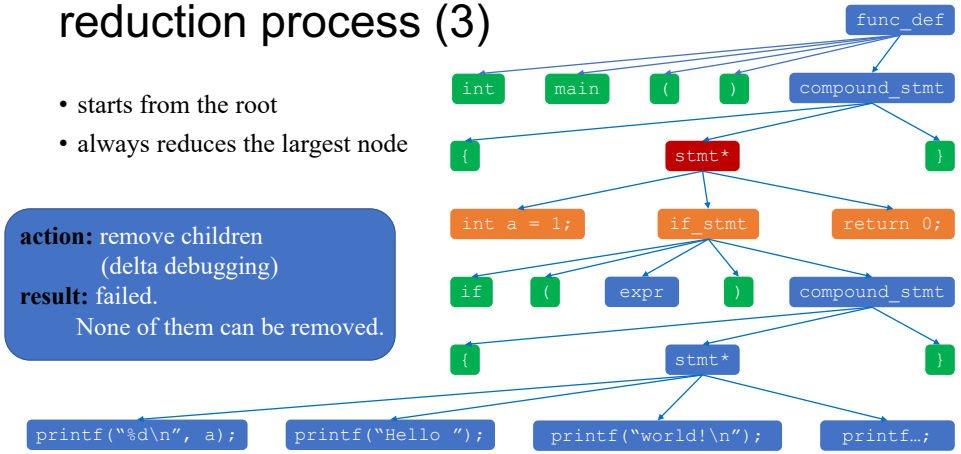
result: failed.
compiler error: 'a' is not defined



reduction process (3)

- starts from the root
- always reduces the largest node

action: remove children
(delta debugging)
result: failed.
None of them can be removed.

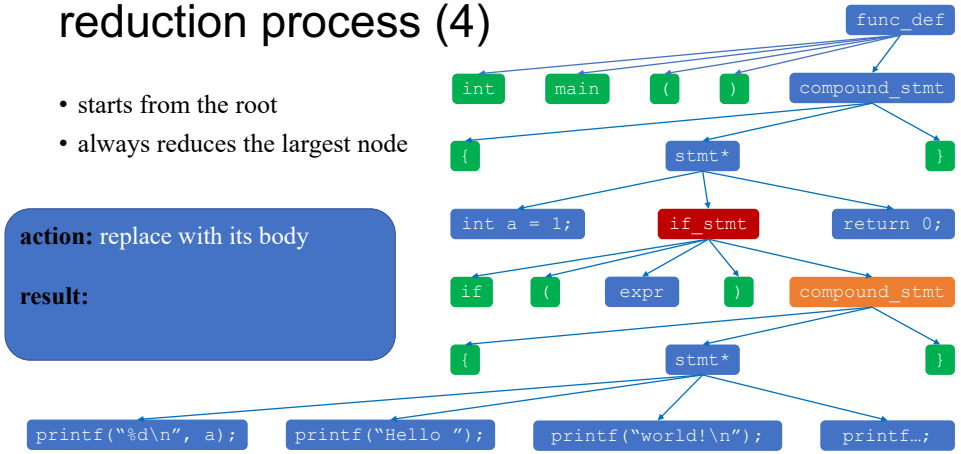


reduction process (4)

- starts from the root
- always reduces the largest node

action: replace with its body

result:

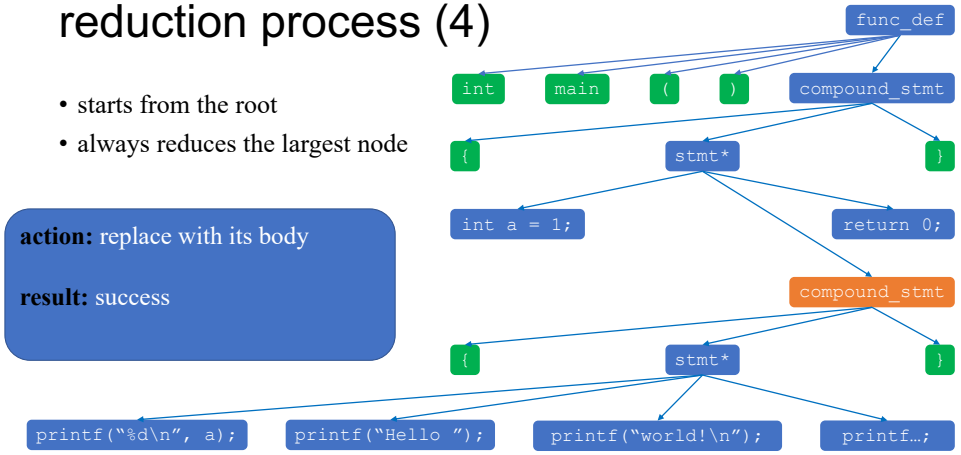


reduction process (4)

- starts from the root
- always reduces the largest node

action: replace with its body

result: success

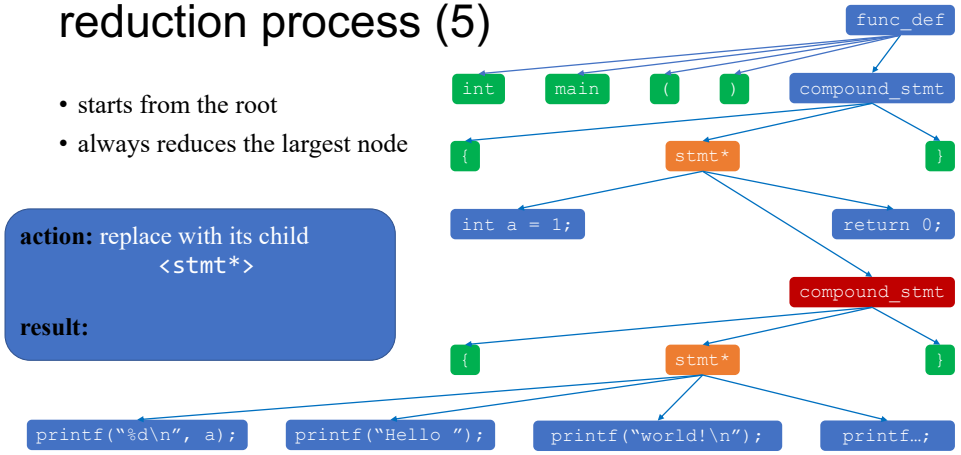


reduction process (5)

- starts from the root
- always reduces the largest node

action: replace with its child
<stmt*>

result:

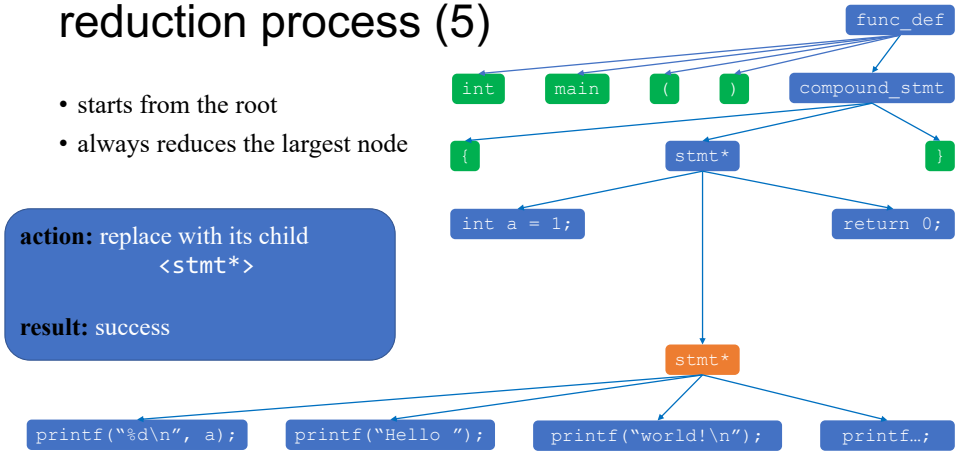


reduction process (5)

- starts from the root
- always reduces the largest node

action: replace with its child
`<stmt*>`

result: success

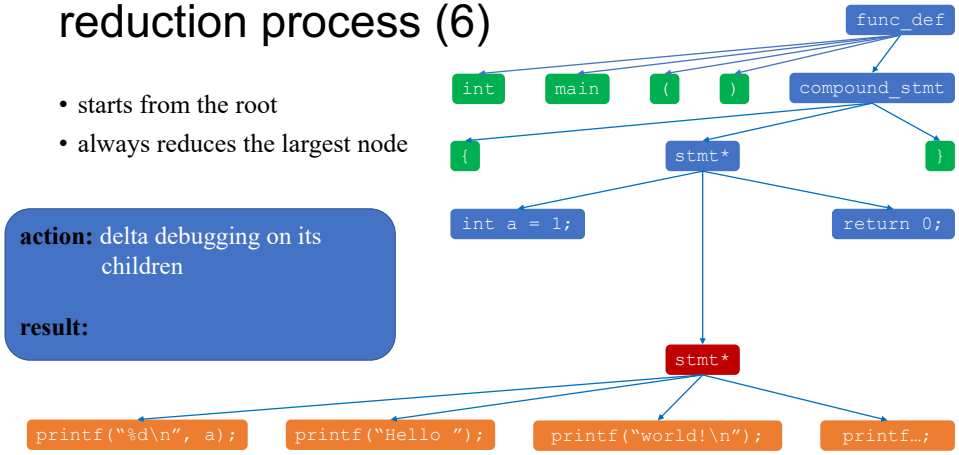


reduction process (6)

- starts from the root
- always reduces the largest node

action: delta debugging on its children

result:

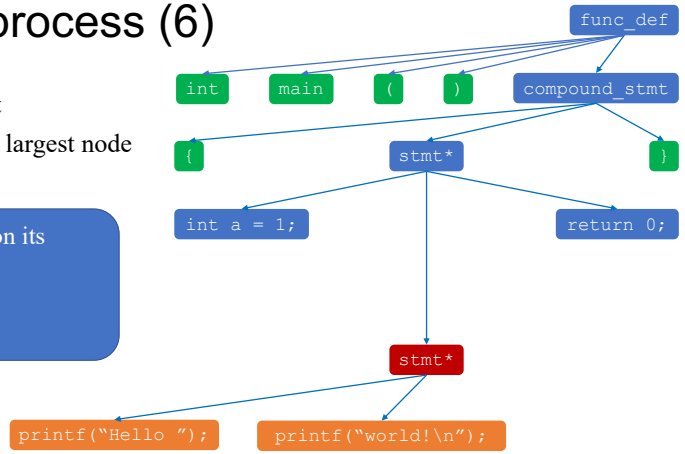


reduction process (6)

- starts from the root
- always reduces the largest node

action: delta debugging on its children

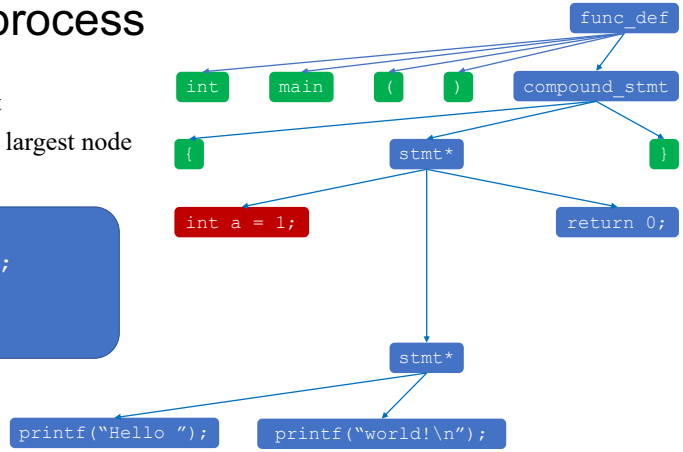
result: success



reduction process

- starts from the root
- always reduces the largest node

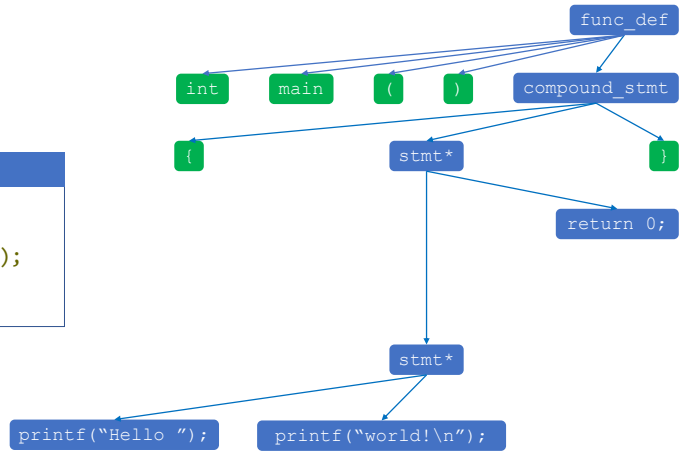
later
`int a = 1;`
will be deleted.



final result

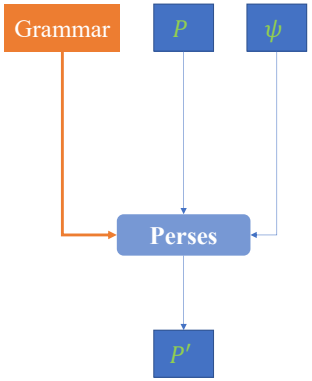
ideal result

```
int main() {
 printf("Hello ");
 printf("world!\n");
 return 0;
}
```



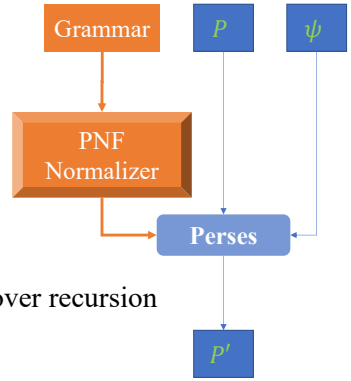
Perses normal form

- Perses relies on quantifiers
  - `*`, `+`, `?`
- A grammar can be written recursively
  - `<stmt_list> ::= <stmt> <stmt_list>`
    `| ϵ`



Perses normal form

- Perses relies on quantifiers
  - `*`, `+`, `?`
- A grammar can be written recursively
  - `<stmt_list> ::= <stmt> <stmt_list>`
    `| ϵ`
- Perses Normal Form: in favor of `*`, `+`, `?` over recursion
  - `<stmt_list> ::= <stmt>*`
- Propose an automatic conversion algorithm (section 4.1 in icse'18)



evaluation

- benchmarks
  - 20 C programs: each triggered a bug in a stable compiler release
    - GCC, Clang
  - size: 6K ~ 212K tokens
- comparison
  - DD: delta debugging
  - HDD: hierarchical delta debugging
  - C-Reduce:
    - specialized reducer for C/C++
    - based on Clang front-end

evaluation

- effectiveness
  - number of tokens in the reduced result
  - number of characters in the reduced result (not meaningful for programs)
- efficiency
  - time (wall time of the reduction process)
  - number of queries/tests/variants
  - reduction speed (#tokens deleted per second)

evaluation – effectiveness

size of reduced programs (mean)					
	original	DD	HDD	C-Reduce	**Perses**
size (#)	94,486	4,573	575	90	**257**
ratio (Perses/other)		6%	45%	285%	**100%**

evaluation – efficiency (1)

A property check ψ takes constant time
→ then number of ψ checks becomes a good measure of efficiency.
→ more checks mean more reduction time

number of property checks				
	DD	HDD	C-Reduce	**Perses**
#	78,305	16,886	27,359	**5,095**
ratio (Perses/other)	7%	30%	19%	**100%**

evaluation – efficiency (2)

reduction time				
	DD	HDD	C-Reduce	**Perses**
seconds	9,710	4,658	3,675	**2,198**
ratio (Perses/other)	23%	47%	60%	**100%**

reduction speed				
	DD	HDD	C-Reduce	**Perses**
tokens/seconds	20	37	33	**63**
ratio (Perses/other)	3.2x	1.7x	1.9x	**100%**

conclusion

- general, syntax-guided program reduction
- outperform DD, HDD, complement C-Reduce on C/C++
  - smaller reduction results in less time
- language-agnostic and fully automated support for any? new language

a rustc bug to be reduced – the program

```
#![feature(member_constraints)]
#![feature(type_alias_impl_trait)]
#[derive(Clone)]
struct CopyIfEq<T, U>(T, U);
impl<T: Copy> Copy for CopyIfEq<T, T> {}
type E<'a, 'b> = impl Sized;
fn foo<'a: 'b, 'b, 'c>(x: &'static i32, mut y: &'a i32) -> E<'b, 'c> {
    let v = CopyIfEq::<*mut _, *mut _>(&mut { x }, &mut y);
    let a_closure = || {
        let x: Option<u32> = None;
        x?;
        22
    };
    let _: *mut &'a i32 = u.1;
    unsafe {
        let _: &'b i32 = *u.0;
    }
    u.0
}
fn main() {}
```


a rustc bug to be reduced – the property

```
#![feature(member_constraints)]
#![feature(type_alias_impl_trait)]
#[derive(Clone)]
struct CopyIfEq<T, U>(T, U);
impl<T: Copy> Copy for CopyIfEq<T, T> {}
type E<'a, 'b> = impl Sized;
fn foo<'a: 'b, 'b, 'c>(x: &'static i32, mut y: &'a i32
    let v = CopyIfEq::<*mut _, *mut _>(&mut { x }, &mu
    let a_closure = || {
        let x: Option<u32> = None;
        x?;
        22
    };
    let _: *mut &'a i32 = u.1;
    unsafe {
        let _: &'b i32 = *u.0;
    }
    u.0
}
fn main() {}
```

```
#!/usr/bin/env bash
set -o nounset
set -o pipefail
set -o xtrace

readonly OUTPUT="temp_compilation_output.tmp.txt"
readonly CRASH_EXIT_CODE=101

readonly RUSTC_AT_ECEUBUNTU="/opt-src/rust-1.58.1/bin/rustc"
if [[ -f "${RUSTC_AT_ECEUBUNTU}" ]] ; then
  readonly RUSTC="${RUSTC_AT_ECEUBUNTU}"
else
  readonly RUSTC="rustc"
fi

readonly RUSTC_VERSION=$(${RUSTC} --version)
if [[ "${RUSTC_VERSION}" != *"1.58.1"* ]] ; then
  echo "the version of rustc is not 1.58.1"
  exit 1
fi
timeout -s 9 30 "${RUSTC}" t.rs &> "${OUTPUT}"

[[ "$?" == "${CRASH_EXIT_CODE}" ]] || exit 1


if ! grep --quiet --fixed-strings "error: internal compiler error: compiler/rustc_borrowck/src/universal_regions.rs" "${OUTPUT}" ; then
  cat "${OUTPUT}"
  exit 1
fi

if ! grep --quiet --fixed-strings "thread 'rustc' panicked at 'Box<dyn Any>'" "${OUTPUT}" ; then
  cat "${OUTPUT}"
  exit 1
fi

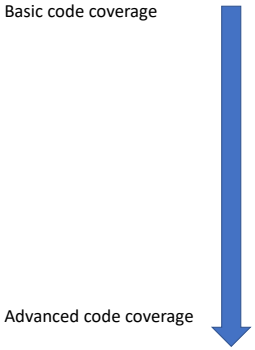
if ! grep --quiet --fixed-strings "note: the compiler unexpectedly panicked. this is a bug." "${OUTPUT}" ; then
  cat "${OUTPUT}"
  exit 1
fi
exit 0
```


Logic Coverage

Chengnian Sun
cnsun@

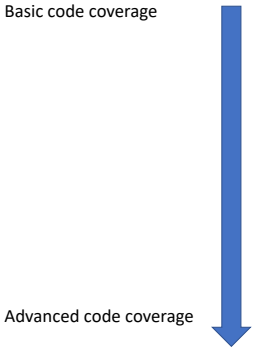
*Slides adapted from Prof. Lin Tan's slides.

Code Coverage



- Line Coverage
- Statement
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- Path coverage
- Loop coverage
- Mutation coverage

Code Coverage



- Line Coverage
- Statement
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- Path coverage
- Loop coverage
- Mutation coverage

Why Logic Coverage?

- MC/DC (Modified condition/decision coverage)

- MC/DC is required by the FAA for safety critical software
- It is used in the standard Do-178B
- Software considerations in Airborne Systems and Equipment Certification.
- A practical tutorial on Modified Condition/Decision Coverage – by NASA

http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf

Decisions/Predicates

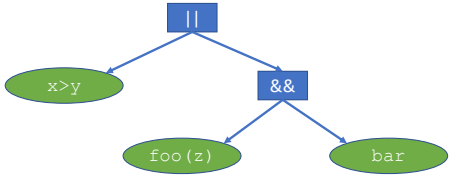
- A logical expression in a program is called a decision/predicate.
  - an expression that evaluates to a Boolean value.
    - e.g., $a \wedge b \leftrightarrow c$
  - can come from code:
    - e.g., `b=(visited && x > y || foo(z))`

- The leaves of a decision are called conditions/clauses.
- Clauses are connected by logical operators to make a decision.

Conditions/Clauses

- Conditions are decisions/predicates without logical operators
  - also called *atomic predicates*

- For example, this predicate contains three conditions/clauses:
  - `(x>y)||(foo(z) && bar)`



Logical Operators

- Here are the logical operators we allow, in order of **precedence** (high to low)
  - ¬: negation
  - ∧: and (not short-circuit)
  - ∨: or (not short-circuit)
  - →: implication
  - ⊕: exclusive or
  - ↔: equivalence

Notations

- P is a set of decisions (predicates) of the program
- For a predicate p, C_p is the set of conditions making up p
- C is the set of all clauses in all the decisions in P

- Formally,

$$C_p = \{c | c \in p\} \qquad C = \bigcup_{p \in P} C_p$$

Decision Coverage

- Decision Coverage (DC). For each $p \in P$, TR contains two requirements:
  - p evaluates to true; and
  - p evaluates to false
- DC is analogous to edge coverage on a CFG. (Let P be the decisions associated with branches.)
  - Example: $P = \{(x + 1 == y) \wedge z\}$

Decision Coverage:
- $x = 4, y = 5, z = true \rightarrow P = T$
- $x = 3, y = 5, z = false \rightarrow P = F$

- DC is a very coarse-grained.
  - Break up decisions into clauses.

Condition Coverage (CC)

- For each $c \in C$, TR contains two requirements:
  - c evaluates to true; and
  - c evaluates to false.
- Example: $p = a \vee b$
- Question: Are there subsumption relationships between CC and DC?

Condition Coverage (CC)

- For each $c \in C$, TR contains two requirements:
  - c evaluates to true; and
  - c evaluates to false.
- Example: $p = a \vee b$
- Question: Are there subsumption relationships between CC and DC?
  - No subsumption relations between CC and DC.
  - List the TR or CC and DC
  - Try to find a test set satisfying CC but not DC
  - Try to find a test set satisfying DC but not CC

Combinatorial Coverage (CoC)

- For each $p \in P$, TR has test requirements for the conditions in C_p to evaluate to each possible combination of truth values.
- This is also known as multiple condition coverage.
- Unfortunately, the number of test requirements, while finite, grows _____ and is hence unscalable.

Active Conditions

- Focus on **each** condition and make sure it affects the decision
  - Test each condition while it is active

$$p = x \wedge y \vee (z \wedge w)$$

- For example, focus on y – called the major condition.
  - Minor conditions: *x*, *z*, *w*.

Determine $p = x \wedge y \vee (z \wedge w)$

- y determines p with certain minor condition values:
  - if we set y to true, then p evaluates to some value X;
  - if we set y to false, then p must evaluate to $\neg X$
- Question: What truth assignment for x, z, w will make y determine p; in particular, y true makes p true and y false make p false
  - x = ?
  - z = ?
  - w = ?

Determine

- $p = a \vee b$
  - b does not determine p when:
- We do not require the major clause has the same truth value as p, which leads to trouble.
  - Example?

Determine

- $p = a \vee b$
  - b does not determine p when: *a=true*
- We do not require the major clause has the same truth value as p, which leads to trouble.
  - Example?

Determine

- $p = a \vee b$
  - b does not determine p when: *a=true*
- We do not require the major clause has the same truth value as p, which leads to trouble.
  - Example? $p = T \oplus y$

Symbolically Making a Condition Determine a Decision

For predicate p and condition c,

- $p_{c=true}$: p with c replaced by true; and
- $p_{c=false}$: p with c replaced by false.

Construct

$$p_c = p_{c=true} \oplus p_{c=false}$$

Property of p_c

p_c describes conditions under which c determines p.

Namely,

- if p_c evaluates to true, then c determines p
- if p_c evaluates to false, then c does not determine p

p_c -- Example

$p = a \vee b$

$$\begin{aligned} p_a = p_{a=\text{true}} \oplus p_{a=\text{false}} &= (\text{true} \vee b) \oplus (\text{false} \vee b) \\ &= \text{true} \oplus b \\ &= \neg b \end{aligned}$$

p_c -- Another Example

$p = a \wedge b$

$$\begin{aligned} p_a = p_{a=\text{true}} \oplus p_{a=\text{false}} &= (\text{true} \wedge b) \oplus (\text{false} \wedge b) \\ &= b \oplus \text{false} \\ &= b \end{aligned}$$

p_c -- Exercise

$p = a \wedge (b \vee c)$

$$p_b = ?$$

$$p = a \vee \neg a$$

$$p_a = p_{a=true} \oplus p_{a=false} = (true \vee false) \oplus (false \vee true) = false$$

What if p_c is false

- If *c* can never determine *p*, then *c* is redundant with respect to *p*.
  - For tester, this means that we could write *p* without using *c*, so something is probably wrong somewhere.

A Tabular Shortcut for Determination

$\boldsymbol{p = a \wedge (b \vee c)}$, check whether $\boldsymbol{a}$ determines $\boldsymbol{p}$

	a	b	c	$\boldsymbol{p = a \wedge (b \vee c)}$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- $\boldsymbol{b}$ and $\boldsymbol{c}$ are minor clauses
- $\boldsymbol{a}$ is the major clause

For each truth assignment of b and c, check the value of a and p

A Tabular Shortcut for Determination

$p = a \wedge (b \vee c)$, check whether a determines p

	a	b	c	$p = a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- b and c are minor clauses
- a is the major clause

For each truth assignment of b and c, check the value of a and p

b	c	a determines p
T	T	
T	F	
F	T	
F	F	

A Tabular Shortcut for Determination

$p = a \wedge (b \vee c)$, check whether a determines p

	a	b	c	$p = a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- b and c are minor clauses
- a is the major clause

For each truth assignment of b and c, check the value of a and p

b	c	a determines p
T	T	Yes
T	F	
F	T	
F	F	

A Tabular Shortcut for Determination

$p = a \wedge (b \vee c)$, check whether a determines p

	a	b	c	$p = a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- b and c are minor clauses
- a is the major clause

For each truth assignment of b and c, check the value of a and p

b	c	a determines p
T	T	Yes
T	F	Yes
F	T	
F	F	

A Tabular Shortcut for Determination

$p = a \wedge (b \vee c)$, check whether a determines p

	a	b	c	$p = a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- b and c are minor clauses
- a is the major clause

For each truth assignment of b and c, check the value of a and p

b	c	a determines p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	

A Tabular Shortcut for Determination

$p = a \wedge (b \vee c)$, check whether a determines p

	a	b	c	$p = a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- b and c are minor clauses
- a is the major clause

For each truth assignment of b and c, check the value of a and p

b	c	a determines p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	No

Active Clause/Condition Coverage (ACC)

- For each p ∈ P and making each clause ci ∈ Cp major, choose assignments for minor clauses cj , j ≠ i such that ci determines p. TR has two requirements for each ci: ci evaluates to true and ci evaluates to false.
- This is a form of MC/DC, which is required by the FAA for safety critical software.

ACC Example

- For *p* = *a* v *b*, make *a* major. We need *b* to be *false* for *a* to determine *p*.
  - This leads to the TRs: {(a=T, b=F), (a=F, b=F)}
- Similarly for b to determine p we need TRs:
  - {(a=F, b=T), (a=F, b=F)}
- Note the overlap between test requirements; it will always exist, meaning that our set of TRs for active clause coverage are:
  - {(a=T, b=F), (a=F, b=T), (a=F, b=F)}

Ambiguity

ACC is almost identical to the way early papers described another technique called MCDC. It turns out that this criterion has some ambiguity, which has led to a fair amount of confusion about how to interpret MCDC over the years. The most important question is whether the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false. Resolving this ambiguity leads to three distinct and interesting flavors of Active Clause Coverage.

Ambiguity

ACC is almost identical to the way early papers described another technique called MCDC. It turns out that this criterion has some ambiguity, which has led to a fair amount of confusion about how to interpret MCDC over the years. The most important question is whether the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false. Resolving this ambiguity leads to three distinct and interesting flavors of Active Clause Coverage.

- Consider predicate $p = a \wedge (b \vee c)$.

and let a be the major clause. Then does the following test suite,

$\langle a : \text{true}, b : \text{false}, c : \text{true} \rangle$ $\langle a : \text{false}, b : \text{true}, c : \text{true} \rangle$

satisfy ACC? Only on a (but you need more cases for b and c).

Look at "a" only for now!

Ambiguity

ACC is almost identical to the way early papers described another technique called MCDC. It turns out that this criterion has some ambiguity, which has led to a fair amount of confusion about how to interpret MCDC over the years. The most important question is whether the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false. Resolving this ambiguity leads to three distinct and interesting flavors of Active Clause Coverage.

- Consider predicate $p = a \wedge (b \vee c)$.

and let a be the major clause. Then does the following test suite,

$\langle a : \text{true}, b : \text{false}, c : \text{true} \rangle$ $\langle a : \text{false}, b : \text{true}, c : \text{true} \rangle$

satisfy ACC? Only on a (but you need more cases for b and c).

Look at "a" only for now!

Resolving the ambiguity → Three flavors of ACC: General, Correlated and Restricted

General Active Clause Coverage (GACC)

- For each p ∈ P and letting each clause ci ∈ Cp be a major clause, choose minor clause values cj such that ci determines p. TR contains two test requirements for each ci: ci evaluates to true and ci evaluates to false. Note that the cjs may be different when ci evaluates to true and ci evaluates to false. There are no restrictions on the cj.

- Unfortunately, GACC does not subsume DC; it is more like CC (and obviously subsumes it).
  - e.g., p = a ⊕ b

Correlated Active Clause Coverage (CACC)

- For each p ∈ P and letting each clause ci be major, choose minor clause values for cj such that ci determines p. TR contains two requirements for each ci: ci evaluates to true and ci evaluates to false. Furthermore, the values chosen for cj must cause p to be true for one value of ci and false for the other.

- To satisfy CACC, you have to satisfy GACC, plus, for each clause, you must make p true and false.

Restricted Active Clause Coverage (RACC)

- For each p ∈ P and letting each clause ci be major, choose minor clause values for cj such that ci determines p. TR contains two requirements for each ci: ci evaluates to true and ci evaluates to false. The values for each cj must be the same when ci is true and when ci is false.

- Key point: must have the same minor clause values in the test set for each major clause. It is therefore harder to satisfy RACC than CACC.

Summary of ACC

- General Active Clause Coverage (GACC) The cjs may be different when ci evaluates to true and ci evaluates to false. There are no restrictions on the cj.
- Correlated Active Clause Coverage (CACC) Values chosen for cj must cause p to be true for one value of ci and false for the other.
- Restricted Active Clause Coverage (RACC) The values for each cj must be the same when ci is true and when ci is false.

- Unfortunately, GACC does not subsume PC; it is more like CC (and obviously subsumes it).

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p.

(b) Compute and simplify the conditions under which each of the clauses determines predicate p. The condition must only use ∨, ∧, and ¬.

(c) Write the complete truth table for all clauses. Label your rows starting from 1. Row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.

(d) Identify all pairs of rows from your table that satisfy GACC with respect to each clause.

(e) Identify all pairs of rows from your table that satisfy CACC with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy RACC with respect to each clause.

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of the clauses determines predicate p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T, p_b = c, p_c = b$

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of the clauses determines predicate p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T, p_b = c, p_c = b$

(c) Write the complete truth table for all clauses. Label your rows starting from 1. Row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.

(d) Identify all pairs of rows from your table that satisfy GACC with respect to each clause.

(e) Identify all pairs of rows from your table that satisfy CACC with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy RACC with respect to each clause.

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		
2	T	T	F	F			
3	T	F	T	F			
4	T	F	F	F			
5	F	T	T	F	T		
6	F	T	F	T			
7	F	F	T	T			
8	F	F	F	T			

b	c	p_a: a determins p
T	T	Yes

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		
2	T	T	F	F	T		
3	T	F	T	F			
4	T	F	F	F			
5	F	T	T	F	T		
6	F	T	F	T	T		
7	F	F	T	T			
8	F	F	F	T			

b	c	p_a: a determins p
T	T	Yes
T	F	Yes

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		
2	T	T	F	F	T		
3	T	F	T	F	T		
4	T	F	F	F			
5	F	T	T	F	T		
6	F	T	F	T	T		
7	F	F	T	T	T		
8	F	F	F	T			

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		
2	T	T	F	F	T		
3	T	F	T	F	T		
4	T	F	F	F	T		
5	F	T	T	F	T		
6	F	T	F	T	T		
7	F	F	T	T	T		
8	F	F	F	T	T		

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	Yes

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		T
2	T	T	F	F	T		T
3	T	F	T	F	T		
4	T	F	F	F	T		
5	F	T	T	F	T		
6	F	T	F	T	T		
7	F	F	T	T	T		
8	F	F	F	T	T		

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	Yes

a	b	p_c: c determins p
T	T	Yes
T	F	
F	T	
F	F	

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		T
2	T	T	F	F	T		T
3	T	F	T	F	T		F
4	T	F	F	F	T		F
5	F	T	T	F	T		
6	F	T	F	T	T		
7	F	F	T	T	T		
8	F	F	F	T	T		

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	Yes

a	b	p_c: c determins p
T	T	Yes
T	F	No
F	T	
F	F	

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		T
2	T	T	F	F	T		T
3	T	F	T	F	T		F
4	T	F	F	F	T		F
5	F	T	T	F	T		T
6	F	T	F	T	T		T
7	F	F	T	T	T		
8	F	F	F	T	T		

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	Yes

a	b	p_c: c determins p
T	T	Yes
T	F	No
F	T	Yes
F	F	

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T		T
2	T	T	F	F	T		T
3	T	F	T	F	T		F
4	T	F	F	F	T		F
5	F	T	T	F	T		T
6	F	T	F	T	T		T
7	F	F	T	T	T		F
8	F	F	F	T	T		F

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	Yes

a	b	p_c: c determins p
T	T	Yes
T	F	No
F	T	Yes
F	F	No

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

b	c	p_a: a determins p
T	T	Yes
T	F	Yes
F	T	Yes
F	F	Yes

a	b	p_c: c determins p
T	T	Yes
T	F	No
F	T	Yes
F	F	No

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of the clauses determines predicate p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T, p_b = c, p_c = b$

(c) Write the complete truth table for all clauses. Label your rows starting from 1. Row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.

(d) Identify all pairs of rows from your table that satisfy GACC with respect to each clause.

(e) Identify all pairs of rows from your table that satisfy CACC

(f) Identify all pairs of rows from your table that satisfy RACC

Answer:
a: {1,2,3,4} x {5,6,7,8}

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of the clauses determines predicate p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T, p_b = c, p_c = b$

(c) Write the complete truth table for all clauses. Label your rows starting from 1. Row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.

(d) Identify all pairs of rows from your table that satisfy GACC with respect to each clause.

(e) Identify all pairs of rows from your table that satisfy CACC

(f) Identify all pairs of rows from your table that satisfy RACC

Answer:
a: {1,2,3,4} x {5,6,7,8}
b: {1,5} x {3,7}

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of the clauses determines predicate p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T, p_b = c, p_c = b$

(c) Write the complete truth table for all clauses. Label your rows starting from 1. Row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.

(d) Identify all pairs of rows from your table that satisfy GACC with respect to each clause.

(e) Identify all pairs of rows from your table that satisfy CACC

(f) Identify all pairs of rows from your table that satisfy RACC

Answer:
a: {1,2,3,4} x {5,6,7,8}
b: {1,5} x {3,7}
c: {1,5} x {2,6}
one full test set: (1,7) + (1,6) = (1, 6, 7)

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of th
p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T,$

(c) Write the complete truth table for all clauses. Label your row
be all clauses true. You should include columns for the condi
determines the predicate, and also a column for the predicat

(d) Identify all pairs of rows from your table that satisfy GACC w

(e) Identify all pairs of rows from your table that satisfy CACC with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy RACC with respect to each clause.

Answer:
a: (1, 5) or {2,3,4} x {6,7,8}

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	T	T	F	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

Homework: $p = a \leftrightarrow (b \wedge c)$

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of th
p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T,$

(c) Write the complete truth table for all clauses. Label your row
be all clauses true. You should include columns for the condi
determines the predicate, and also a column for the predicat

(d) Identify all pairs of rows from your table that satisfy GACC w

(e) Identify all pairs of rows from your table that satisfy CACC with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy RACC with respect to each clause.

Answer:
a: (1, 5) or {2,3,4} x {6,7,8}
b: (1,3) or (5,7)

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

Homework: $p = a \leftrightarrow (b \wedge c)$

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of th
p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T,$

(c) Write the complete truth table for all clauses. Label your row
be all clauses true. You should include columns for the condi
determines the predicate, and also a column for the predicat

(d) Identify all pairs of rows from your table that satisfy GACC w

(e) Identify all pairs of rows from your table that satisfy CACC with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy RACC with respect to each clause.

Answer:
a: (1, 5) or {2,3,4} x {6,7,8}
b: (1,3) or (5,7)
c: (1,2) or (5,6)
one full test set: (1,5) + (1,3) + (1,2) = (1,2,3,5)

Homework: $p = a \leftrightarrow (b \wedge c)$

	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T	T	T	T
2	T	T	F	F	T	F	T
3	T	F	T	F	T	T	F
4	T	F	F	F	T	F	F
5	F	T	T	F	T	T	T
6	F	T	F	T	T	F	T
7	F	F	T	T	T	T	F
8	F	F	F	T	T	F	F

(a) Identify all the clauses in predicate p. Answer: a, b, c

(b) Compute and simplify the conditions under which each of th
p. The condition must only use ∨, ∧, and ¬. Answer: $p_a = T,$

(c) Write the complete truth table for all clauses. Label your row
be all clauses true. You should include columns for the condi
determines the predicate, and also a column for the predicat

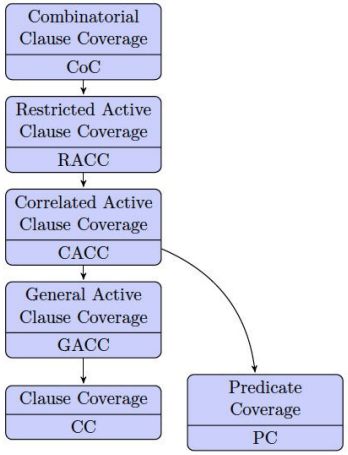
(d) Identify all pairs of rows from your table that satisfy GACC w

(e) Identify all pairs of rows from your table that satisfy CACC with respect to each clause.

(f) Identify all pairs of rows from your table that satisfy RACC with respect to each clause.

Answer:
a: (1,5), (2,6), (3,7) or (4,8)
b: (1,3) or (5,7)
c: (1,2) or (5,6)
one test set: (1,5) + (1,3) + (1,2) = (1,2,3,5)

Subsumption



Infeasibility

- A logic coverage test requirement ***t*** is infeasible if no test case can satisfy test requirement ***t***.
- Clearly, no array c can have negative length in the following example, so the predicate can't be true.

```
void m(char[] c ) {
  if (c.length < 0) {
    print ("infeasible");
  }
}
```

Workaround

- If some test requirements are not feasible:
  - ***Workaround I***. Satisfy feasible test requirements, drop infeasible test requirements.
  - ***Workaround II***. If you can't satisfy a particular coverage criterion, use a looser criterion. For instance, if you can't satisfy RACC, settle for CACC.
- Workaround II is analogous to best-effort touring in the graph coverage case.

Causes of Infeasibility

- Several ways for a coverage test requirement to be infeasible:
  - The predicate is unreachable in its context (as in graph reachability); or
  - The method containing the predicate never assigns appropriate values to its variables to cause the desired clause values (as in the example above); or
  - A clause never determines a predicate and you are trying to achieve an active clause coverage criterion.
- These ways reflect the steps you need to follow to satisfy test requirements.

To Achieve Logic Coverage

1) identify the predicates p ∈ P in the program fragment under test;
2) figure out how to reach each of the predicates;
3) make c determine p (for the active clause criteria); and
4) find values for (program) variables to meet various criteria.


```
isPat == false
    && iSub + pattern.length - 1 < subject.length
    || subject[iSub] == pattern[0]
```

4) Finding values for Variables

- An example predicate from the textbook

4) Finding values for Variables

```
isPat == false
    && iSub + pattern.length - 1 < subject.length
    || subject[iSub] == pattern[0]
```

- An example predicate from the textbook
- We can assign names to these clauses, giving the symbolic predicate
  - p = a ∧ b ∨ c

```
a = isPat == false;
b = iSub + pattern.length - 1 < subject.length
c = subject[iSub] == pattern[0]
p = a ∧ b ∨ c
```


4) Finding values for Variables

```
isPat == false
    && iSub + pattern.length - 1 < subject.length
    || subject[iSub] == pattern[0]
```

- An example predicate from the textbook
- We can assign names to these clauses, giving the symbolic predicate
  - p = a ∧ b ∨ c
  - pa = b ∧ ¬ c
  - pb = a ∧ ¬ c
  - pc = ¬ (a ∧ b)

```
a = isPat == false;
b = iSub + pattern.length - 1 < subject.length
c = subject[iSub] == pattern[0]
p = a ∧ b ∨ c
```


4) Finding values for Variables

```
isPat == false
    && iSub + pattern.length - 1 < subject.length
    || subject[iSub] == pattern[0]
```

- An example predicate from the textbook
- We can assign names to these clauses, giving the symbolic predicate
  - p = a ∧ b ∨ c
  - pa = b ∧ ¬ c
  - pb = a ∧ ¬ c
  - pc = ¬ (a ∧ b)
- Example Assignment:
  - a = T: `isPat=false`
  - b = T: `iSub=0, pattern=new char[1], subject=new char[5]`
  - c = T: `iSub=0, pattern[0]=subject[0]='a'`

```
a = isPat == false;
b = iSub + pattern.length - 1 < subject.length
c = subject[iSub] == pattern[0]
p = a ∧ b ∨ c
```


Coverage Criteria

```
a = isPat == false;
b = iSub + pattern.length - 1 < subject.length
c = subject[iSub] == pattern[0]
p = a ∧ b ∨ c
```

- TR_{PC} = {p=true, p = false}
  - Predicate coverage is decision coverage in this course
- Test set T = {t1, t2} satisfies TR_{PC} for p.
  - t1
    - a=T: `isPat==false`
    - b=T: `iSub=0, pattern=new char[1], subject=new char[5]`
    - c=T: `pattern[0]=subject[0]='a'`
  - t2
    - a=T: `isPat==false`
    - b=F: `iSub=0, pattern=new char[10], subject=new char[5]`
    - c=F: `pattern[0]='b',subject[0]='a'`

Coverage Criteria

```
a = isPat == false;
b = iSub + pattern.length - 1 < subject.length
c = subject[iSub] == pattern[0]
p = a ∧ b ∨ c
```

- Homework:
  - list the TR for CC, GACC, CACC, RACC
  - list test sets to satisfy each coverage criterion

Input Space Partitioning

Chengnian Sun
cnsun@

*Slides adapted from Prof. Lin Tan's and Prof. Patrick Lam's

Input Space Partitioning

- We cannot feed all inputs to the program, so we only test a **representative** set of inputs.
- Input space partitioning makes this idea more "formal"
  - test one input from each partition.
- Two properties for partitions:
  - **Completeness**: covers the entire domain
  - **Disjointness**: must not overlap

Input Space Partitioning

- The underlying assumption of partition coverage is that any test in a block is as good as any other for testing. Several partitions are sometimes considered together, which, if not done carefully, leads to a combinatorial explosion of test cases.
- A common way to apply input space partitioning is to start by considering the domain of each parameter separately, partitioning each domain's possible values into blocks, and then combining the blocks for each parameter. Sometimes the parameters are considered completely independently, and sometimes they are considered in conjunction with each other, usually by taking the semantics of the program into account.

Characteristics of Input

- Partitions are usually based on characteristics of the input (or environment)
  - value,
    - *foo(int a)*: *negative*, *positive*, *zero*
  - length,
    - foo(@NonNull int[] a): zero, one, and more
  - nullness,
    - foo(@Nullable Object a): null, non-null
  - emptiness
    - foo(@NonNull int[] a): empty, non-empty
    - BoundedStack: empty, loaded, full
  - …

Definitions

- Given an input parameter *p*
  - *D*: the input domain of *p*, the set of all possible values of *p*
  - *c*: characteristic of *p*
  - *q:* a partition defined by *c* over *D,*
  - B_q: the set of equivalence classes created by *q* over *D*
  - *Block*: an equivalence class in this course
- Each input value belongs to one and only one block per characteristic.
- Completeness: $\bigcup_{b \in B_q} b = D$
- Disjointness: $b_m \cap b_n = \emptyset, m \neq n; b_m \in B_q; b_n \in B_q$

Definitions

void foo(int a):
D: int
characteristic: relation with zero
blocks: {negative int}, {0}, {positive int}

- Given an input parameter *p*
  - *D*: the input domain of *p*, the set of all possible values of *p*
  - *c*: characteristic of *p*
  - *q:* a partition defined by *c* over *D,*
  - B_q: the set of equivalence classes created by *q* over *D*
  - *Block*: an equivalence class in this course
- Each input value belongs to one and only one block per characteristic.
- Completeness: $\bigcup_{b \in B_q} b = D$
- Disjointness: $b_m \cap b_n = \emptyset, m \neq n; b_m \in B_q; b_n \in B_q$

Disjoint Partitions – Example: *foo(File[] a)*

- A bad partition on file ordering:
  - Block 1: ascending
  - Block 2: descending
  - Block 3: arbitrary
- What is the problem?

Disjoint Partitions – Example: *foo(File[] a)*

- A bad partition on file ordering:
  - Block 1: ascending
  - Block 2: descending
  - Block 3: arbitrary
- What is the problem?
- Instead,
  - use the characteristics separately:
    - partition based on the *ascending* characteristic: true or false
    - partition based on the *descending* characteristic: true or false

Input Space Modeling

- Three Steps
  - Find units/functions to test
  - Identify input parameters of each unit
  - Produce the input space model

Step 1: What Units Should We Test?

- For classes in general, could test all public methods
  - (grouping them together as needed)
  - For Java, you can also test package-private, and protected methods

Use cases can also give you hints about how to group both methods and inputs, or about which methods might be most important.

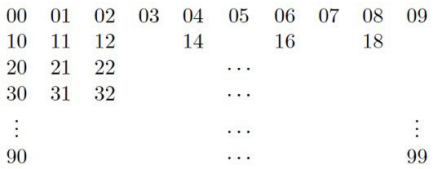
Step 2: What are the Parameters?

- Method parameters and other program state
- Fields and data structures
- Files
- Command-line arguments

- `void insert(Comparable obj)`
  - insert `obj` into a binary search tree
- `int find(String str)`
  - find `str` in a file

Step 3: Create the Model

- An input domain model (IDM) represents the input space of the system under test in an abstract way.
- Input domain of two-digit numbers



- Possible characteristics and partitions:
  - first digit: [0-9]
  - odd, even
  - diagonal

Input Domain Models (IDMs)

- Coming up with IDMs requires *creativity* and *analysis*.
  - (Instructor's note: I don't like creativity, because it is hard to assess objectively)
- Two general approaches:
  - Interface-based
    - using the input space directly, considering each parameter separately.
  - Functionality-based
    - using a functional or behavioral view of the program

Example

```
public boolean containsElement(List list, Object element)
```

- If `list` or element is `null`, throw a `NullPointerException`.
- Otherwise, return
  - true if `list` contains element and
  - false if `list` does not contain element.

public boolean containsElement(List list, Object element)

Interface-Based Input Domain Modeling

- Possible interface-based characteristics:
  - list is null
    - block 1: true, block 2: false
  - list is empty
    - block 1: true, block 2: false
  - element is null
    - block 1: true, block 2: false
- (+) Easy to identify characteristics
- (+) Easy to translate to test cases
- (-) Less effective, because it does not use domain knowledge, e.g., relationship between parameters

public boolean containsElement(List list, Object element)

Functionality-Based Input Domain Modeling

- Possible functionality-based characteristics:
  - Number of occurrences of element in list: 0, 1, > 1
  - Element occurs first in list: true or false
  - Element occurs last in list: true or false
- (+) Might yield better test cases due to domain knowledge
- (+) Can create these models from specifications, do not need implementations
- (-) May be hard to identify values and characteristics
- (-) Harder to generate tests from such an IDM (but worth doing it)

Identifying Characteristics

- Sources of characteristics for functionality-based IDMs:
  - Preconditions and postconditions, e.g.,
    - Object.wait() precondition: must hold lock
      - characteristic: lock held or not.
  - Relationships between variables, e.g., parameter aliasing
    - public static boolean Objects.equals(Object a, Object b)
      - characteristic: a == b?
  - Specifications contain concentrated domain knowledge => easier to extract characteristics from them.
    - leap year, year/4, year/100, year/400 (can be described by a Venn diagram)
- Fewer blocks gives a more tractable partition, and it is easier to ensure disjointness and completeness.

Choosing Blocks and Values

- This part is most creative in input space testing.
- Some tips for finding good values to test with:
  - Be sure to include
    - Valid values and invalid values,
    - Boundary values and non-boundary values, and
    - Special values (0, null, empty set, Integer.MAX_VALUE).
  - If you need more partitions, you can divide existing partitions, especially those containing valid values
  - Check that partitions are disjoint and complete.

Misc.

- Alternative approach. Instead of multiple blocks, use T/F characteristics (e.g., isEquilateral, etc)
  - Then disjointness and completeness are guaranteed, but you get more characteristics
- Verifying IDMs. Some tips:
  - Are we missing anything we could include?
  - Check completeness and disjointness (if using multiple IDMs, only the sum of the IDMs needs to be complete)

Triangle Type Example

- Standard triangle classification example, which takes three side lengths as input
- *private static int Triang (int side1, int side2, int side3) {...}*
  - Returns 1 if triangle is scalene
  - Returns 2 if triangle is isosceles
  - Returns 3 if triangle is equilateral
  - Returns 4 if not a triangle

Interface-Based Tries

- Interface-based (3 blocks)
  - side 1: >0, <0, =0
  - side 2: >0, <0, =0
  - side 3: >0, <0, =0

Partition	*b1*	*b2*	*b3*
q_1 = "Relation of Side 1 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_2 = "Relation of Side 2 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_3 = "Relation of Side 3 to 0"	*greater than 0*	*equal to 0*	*less than 0*

- We might refine some partitions, assuming that we take integers as parameters (4 blocks)
  - side 1: >1, =1, =0, <0
  - side 2: >1, =1, =0, <0
  - side 3: >1, =1, =0, <0

Partition	b_1	b_2	b_3	b_4
q_1 = "Length of Side 1"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_2 = "Length of Side 2"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_3 = "Length of Side 3"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*

Functionality-Based Try – based on triangle type

scalene	e.g., (3,4,5)
isosceles	e.g., (3,3,4)
equilateral	e.g., (3,3,3)
Invalid	e.g., (3,4,8)

- What is wrong with this partition?

Functionality-Based Try – based on triangle type

scalene	(3,4,5)
isosceles, not equilateral	(3,3,4)
equilateral	(3,3,3)
invalid	(3,4,8)

- What is wrong with this partition?
- Equilateral triangles are also isosceles.

Combining Characteristics

- How to test multiple inputs/characteristics systematically, each with their own partitions?
- Consider three inputs, with the following representative values from each block of a partition:
  - p1: [A, B]
    - p1 is the first input, and it has two blocks based on a characteristic
    - A is from the first block, and B is from the second
  - p2: [1, 2, 3]
    - p2 is the second input, and has three blocks based on a characteristic
    - 1 is from the first block, 2 is from the second, and 3 is from the third
  - p3: [x, y]
    - p3 is the third input, and has three blocks based on a characteristic
    - x is from the first block, and y is from the other block.

All Combinations Coverage (ACoC)

[A, B]
[1, 2, 3]
[x, y]

- TR contains all combinations of blocks from all characteristics.
- TR_{ACoC} = {(A, 1, x), (A, 1, y), (A, 2, x), (A, 2, y), (A, 3, x), (A, 3, y), (B, 1, x), (B, 1, y), (B, 2, x), (B, 2, y), (B, 3, x), (B, 3, y)}
- The number of test requirements is the product

$$\prod_{Q} B_i$$

where B_i is the number of blocks for partition i, and Q is the total number of partitions.

Each Choice Coverage (ECC)

[A, B]
[1, 2, 3]
[x, y]

- TR contains the requirements to include one value from each block for each characteristic in some test case.
- TR_{ECC} = {A, B, 1, 2, 3, x, y}
- A test set T = {(A ,1, x), (B, 2, y), (A, 3, x)} satisfies ECC

Pair-Wise Coverage (PWC)

[A, B]
[1, 2, 3]
[x, y]

- TR contains the requirements to combine
  - a value for each block for each characteristic
  - with some value from every block for each other characteristic.
- 16 test requirements = 2x3+2x3+2x2
  - TR_{PWC}= {(A, 1), (A, 2), (A, 3), (A, x), (A, y), (B, 1), (B, 2), (B, 3), (B, x), (B, y), (1, x), (1, y), (2, x), (2, y) (3, x), (3, y)}
- 6 test cases
  - {(A, 1, x), (A, 2, **y**), (A, 3, x), (B, 1, y), (B, 2, **x**), (B, 3, y)}

Treat Different Blocks Differently

[A, B]
[1, 2, 3]
[x, y]

- We've treated all blocks as equivalent so far. We'll now pick one block as the most important block and call it the base choice.
- Examples: simplest, smallest, first (with respect to some ordering), most likely.

Base Choice Coverage (BCC)

[A, B]
[1, 2, 3]
[x, y]

- Choose a base choice block for each characteristic, and a base test by combining the base choices for all characteristics. For each characteristic c, TR contains the requirements for a test which varies the base test by using all other blocks for characteristic c.
- In our example, suppose the base choice blocks are A, 1 and x. Then the base choice test is (A, 1, x).
- Other tests would be (**B**,1, x), (A, **2**, x), (A, **3**, x), (A,1, **y**).
- Number of tests: $1 + \sum^{Q}(B_i - 1)$

Base Choice Coverage (BCC)

side 1: >1, =1, =0, <0
side 2: >1, =1, =0, <0
side 3: >1, =1, =0, <0

- For triangle classification (with 4 blocks), we have 1+3+3+3 = 10 tests.
- Say that the block "> 1" is the base choice.
  - A base choice could be (2, 2, 2)
  - The non-base choices are:
    - (2, 2, 1), (2, 2, 0), (2, 2,-1), (2, 1, 2), (2, 0, 2), (2,-1, 2), (1, 2, 2), (0, 2, 2), (-1, 2, 2)

side 1: >1, =1, =0, <0
side 2: >1, =1, =0, <0
side 3: >1, =1, =0, <0

Constraints among Partitions

- Choices can lead to infeasible test requirements.

public boolean containsElement(List list, Object element)

Characteristics	Block 1	Block 2	Block 3	Block 4
A: length and contents	no element	one element	more than one sorted	more than one unsorted
B: match	element not found	element found once	element found more than once	--

Invalid combinations: (A1, B2), (A2, B3), etc.

Constraints among Partitions

- Choices can lead to infeasible test requirements.

public boolean containsElement(List list, Object element)

Characteristics	Block 1	Block 2	Block 3	Block 4
A: length and contents	no element	one element	more than one sorted	more than one unsorted
B: match	element not found	element found once	element found more than once	--

- Handling infeasible TRs.
  - Drop infeasible TRs for ACoC and PWC.
  - For BCC, we can also change the base case.

Reporting Bugs

Chengnian Sun
cnsun@

*Slides adapted from Prof. Patrick Lam's and Prof. Lin Tan's notes.

Goals of Reporting Bugs

- Get the bugs fixed.
- Get the bugs fixed faster
- For better software.

Properties of Important Bugs

- Bug is sufficiently general to affect many users (easily reproducible).
- Bug has severe consequences (crashes, data loss).
- Bug is new to most recent version.
- Bug has security implications.
- Examples: heartbleed, log4shell

Properties of Important Bugs

- Bug is sufficiently general to affect many users (easily reproducible).
- Bug has severe consequences (crashes, data loss).
- Bug is new to most recent version.
- Bug has security implications.
- Examples: heartbleed, log4shell

Recent Bugs – Heartbleed 2014

- OpenSSL is a widely-used security protocol library
- No bound check for memcpy(bp, pl, payload);
- Affected many websites, e.g., Yahoo, Stack Overflow, DuckDuckGo, …

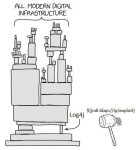
Canada Revenue Agency (CRA)
- **The website was hacked due to this software bug, and had to be shut down,**
- **Deadline for filing individual tax returns was postponed.**

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
https://en.wikipedia.org/wiki/Heartbleed#Websites_and_other_online_services

Properties of Important Bugs

- Bug is sufficiently general to affect many users (easily reproducible).
- Bug has severe consequences (crashes, data loss).
- Bug is new to most recent version.
- Bug has security implications.
- Examples: heartbleed, log4shell

Recent Bugs – log4shell 2021

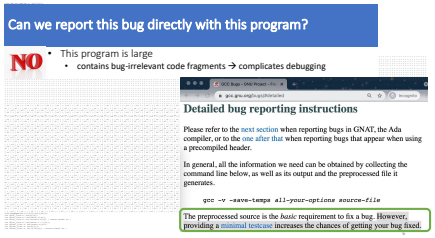


- Log4j is a logging library for Java, widely used
- allows remote code execution
  - affected Amazon Web Services, iCloud, Minecraft, etc
  - affected many websites in Canada
    - CRA, Employment and Social Development…
- "Experts described Log4Shell as the largest vulnerability ever;[36] LunaSec characterized it as "a dawg proportions",[36] Tenable said the exploit was "the single biggest, most critical vulnerability ever",[14] most severe vulnerability ever"[140] and The Washington Post said that descriptions by security professionals "border on the apocalyptic".[36]

https://www.cve.org/CVERecord?id=CVE-2021-44228
https://en.wikipedia.org/wiki/Log4Shell

Reproducibility

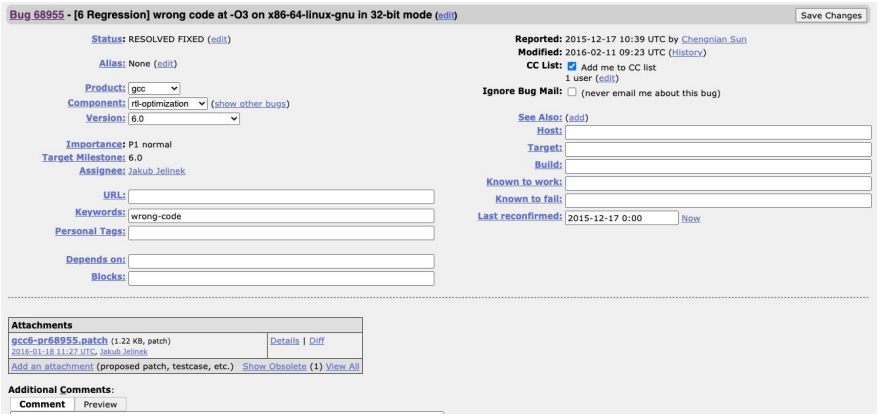
- Developers cannot fix problems they cannot observe.
  - Be maximally specific in describing steps to reproduce.
  - Write the steps down as soon as possible, before you forget.
  - Try to find a *minimal* testcase that demonstrates the problem.



Anatomy of a Bug Report

- Bug id: automatically generated number
- Reporter: usually an email address
- Product, Version, Component: helps direct the bug to the right developers
- Platform, OS: e.g. PC/Linux, or Mac/Mac OS X 10.5.
- Severity: based on consequences; examples: blocker, critical, normal, trivial, enhancement
- Assign to: person responsible for bug
- CC: list of people who track bug changes
- Keywords: e.g. crash, intl, patch, security
- Depends on/blocks: relationships between bugs
- URL: (especially applicable for browsers)
- Attachments: e.g. test cases/input files which exhibit the bug
  - avoid using attachment if you can use description for the same purpose

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68955



Field: Summary

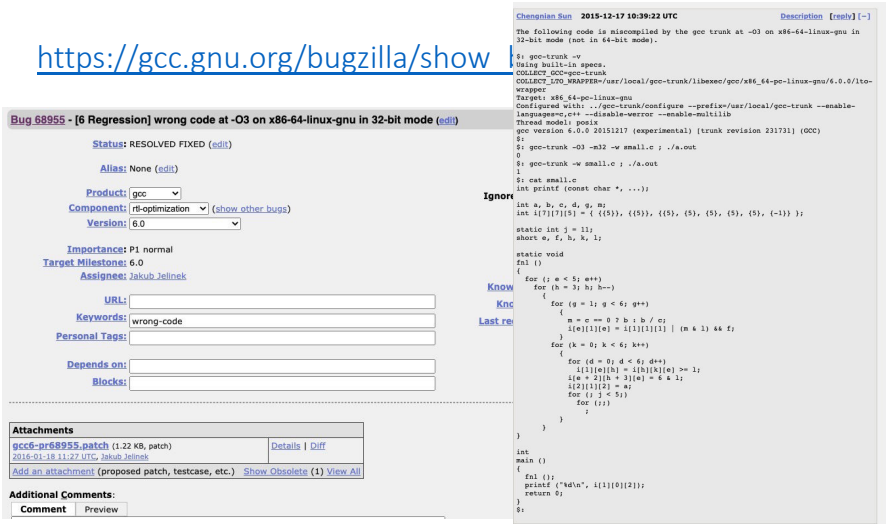
- A one-line recap of the bug.
- Enables searching for and judgement of the bug.
- Perhaps the most critical field
- Examples (some good, some bad):
  - **“RPM 4 installer crashes if launched on Red Hat 6.2 (RPM 3) system”**
  - “Back button does not work”
  - **“When memory cache disabled, no-store pages not displayed at all”**
  - “History and bookmarks completely inoperable”
  - “Can’t install”

Field: Description

Should be a complete description of the bug, including:

- Overview: expanded summary, e.g. “Drag-selecting any page crashes Mac builds in NSGet- Factory”.
- Steps to Reproduce (key!): minimized easy-to-follow steps to trigger the bug;
  - 1) try to reproduce the bug based on the steps you report;
  - 2) try to describe the steps to that anyone (like the developer) can reproduce the bug.
    - Be specific: instead of “save the document”, “File > Save, select foo from dropdown, ...”.
- Actual Results: what you see when you perform the steps to reproduce. e.g. “Application crashes. Stack trace included.”
- Expected Results: what you think is correct
- Build Date and Platform: on development software, helps find the bug; include additional builds and platforms the bug might apply to.

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68955



Lifecycle-Related Fields

Some fields summarize the current state of the bug.

- Comments: either by the assigned developer, original reporter, or interested bystanders. Often people give additional information (“also happens on latest build”) or help diagnose the problem.
- Status: e.g. UNCONFIRMED, NEW, REOPENED, VERIFIED
- Priority: for internal use by development team.
  - severity of the bug
    - priority is different from severity.
  - time

Properties of Good Bug Reports

- Reported in the database.
- Simple: one bug per report.
- Understandable, minimal, and generalizable.
- Reproducible.
- Non-judgemental. “The developers are all morons”.
- Not a duplicate (Should not be reported before.).

- Analysis of the cause of the bug (good to have)

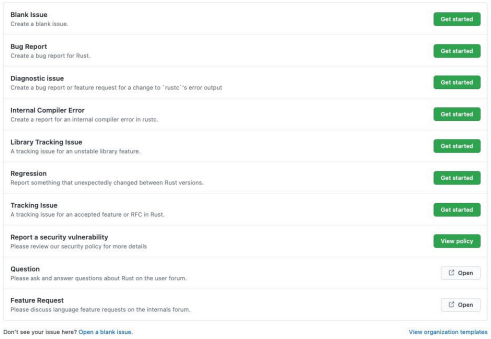
Research: What Makes a Good Bug Report?

- Betternburg et al performed a survey asking experienced developers to rate bug reports and to identify important information in them [BJS+08], published in Foundations of Software Engineering 2008.

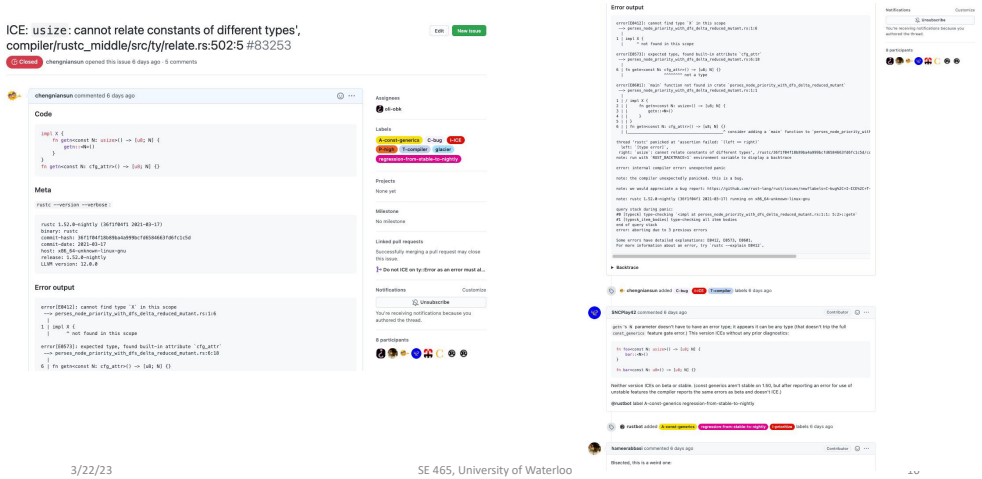
- I recommend consulting the full paper. But the executive summary is that developers rated steps to reproduce (83%), stack traces (57%) and test cases (51%) as most useful. Furthermore, by examining data from Apache projects, Eclipse, and Mozilla, they found that bug reports with stack traces get fixed sooner, and that bug reports that are easier to read have lower lifetimes. Code samples also help increase the chance that a bug report gets fixed.

[BJS+08] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In Proceedings of the 16th International Symposium on Foundations of Software Engineering, November 2008.

Some projects provide templates for various types of issues: https://github.com/rust-lang/rust/issues/new/choose

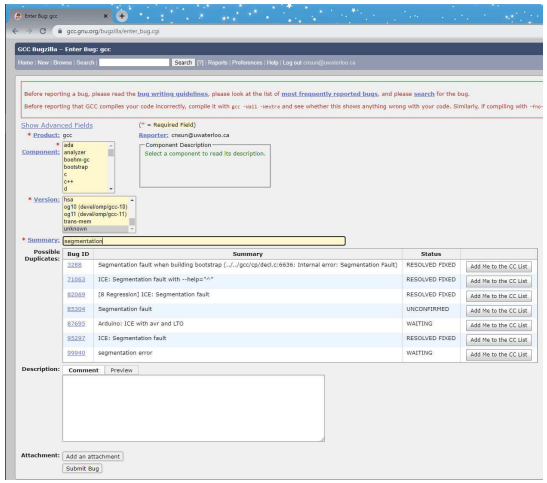


https://github.com/rust-lang/rust/issues/83253



Avoid Reporting Duplicate Bugs

- If a bug is already reported in the database, do not create a new report for the bug. Instead, report the bug as a comment of the bug report.
  - Duplicate bugs reports are not always bad; they might provide different bug-triggering test inputs.
- How to check whether a bug is a duplicate or not?
  - search the database with keywords
  - Bugzilla automatically searches the database with the summary



Avoid Reporting Duplicate Bugs

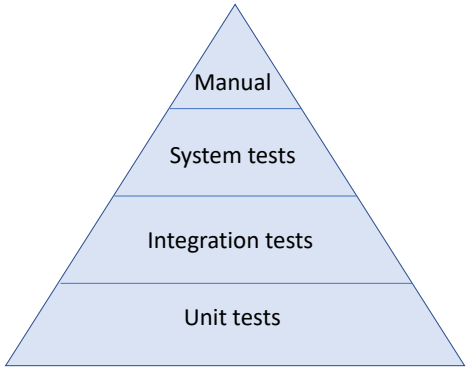
- If a bug is already reported in the database, do not create a new report for the bug. Instead, report the bug as a comment of the bug report.
- How to check whether a bug is a duplicate or not?
  - search the database with keywords
  - Bugzilla automatically searches the database with the summary
- How to detect duplicate bug reports automatically?
  - an information retrieval problem
  - can be improved with machine learning, e.g., LLM.

Testing Pyramid

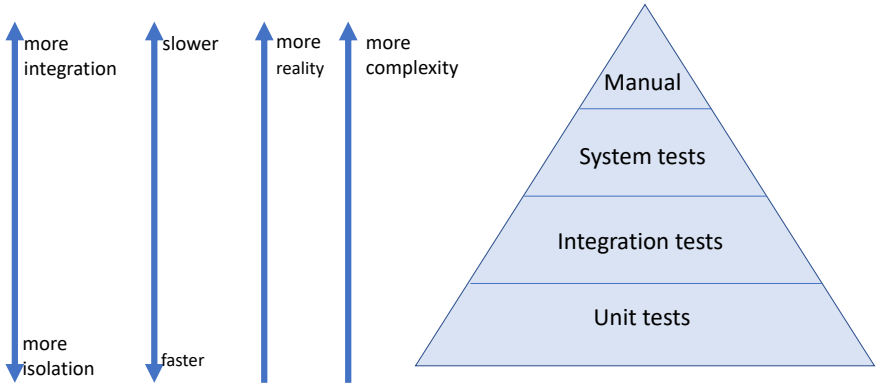
Chengnian Sun
cnsun@

*Slides adapted from sttp.site
Reference: https://martinfowler.com/articles/practical-test-pyramid.html

Testing Pyramid



Testing Pyramid



Unit Testing

A *unit of work* can be

- a single function,
- multiple functions or even
- multiple modules or components.

But it always has an entry point which we can trigger from the outside (via tests or other production code), and it always ends up doing something useful.

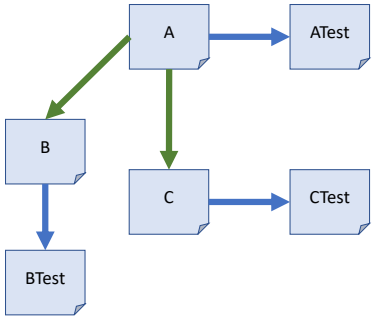
-- The Art of Unit Testing (3rd edition)

A unit test is a piece of code that invokes a unit of work and checks one specific exit point as an end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed. A unit test's scope can span as little as a function or as much as multiple modules or components depending on how many functions and modules are used between the entry point and the exit point.

https://livebook.manning.com/book/the-art-of-unit-testing-third-edition/chapter-1/v-2/36

Unit Testing – What to Test

- Rule of thumb
  - One test class per production class
  - test at least the public interface of the class
    - protected methods
    - package private methods
  - private methods are implementation details
- Test the interface not the implementation
  - if I enter values x and y, will the result be z?
  - if I enter x and y, will the method call class A first, then call class B and then return the result of class A plus the result of class B?
  - when testing a sorting function, tests whether it sorts the input array but not how it sorts



Unit Testing – Test Structure

- A good structure for all your tests
  - **Arrange**: Set up the test data
  - **Act**: Call your method under test
  - **Assert**: Assert that the expected results are returned
- Recall prefix, postfix, test case values in Lecture 4

A Junit test case for MoreFilesTest in Guava

```
public class MoreFilesTest {

  private Path tempDir;

  @Before
  public void setUp() throws Exception {
    tempDir = Files.createTempDirectory("MoreFilesTest");
  }

  @Test
  public void testCreateParentDirectories_oneParentNeeded() {
    Path path = tempDir.resolve("parent/nonexistent.file");
    Path parent = path.getParent();
    assertFalse(Files.exists(parent));
    MoreFiles.createParentDirectories(path);
    assertTrue(Files.exists(parent));
  }

  @After
  protected void tearDown() throws Exception {
    if (tempDir != null) {
      // delete tempDir and its contents
      ……
    }
  }
}
```

Prefix Value

Test case value: path
Expected Results:
- before: no parent
- after: parent create

'Files.exits(parent)' might be a postfix value?

Postfix Value

Unit Testing – Characteristics

- Very fast
  - A unit test usually takes just a couple of milliseconds to execute. Fast tests give us the ability to test huge portions of the system in a small amount of time.
- Easy to control
  - We have a high degree of control on how the unit tests exercise the system. A unit test tests the software by giving certain parameters to a method and then comparing the return value of a method to the expected result. The parameters and expected result values are very easy to be adapted/modified in the test.
- Easy to write
  - Junit
- Less real
  - Only touch a small part of the system
- Some bugs cannot be found with unit testing
  - integration testing, e.g., HTML + JavaScript + Web Server + Database

Unit Testing – Characteristics

- How many units to tests?
- What to do in each unit test?

```
public static void main(String[] argv) {
  M obj = new M();
  if (argv.length > 0) obj.m(argv[0], argv.length);
}

public void m(String arg, int i) {
  int q = 1;
  A o = null;
  Impossible nothing = new Impossible();
  if (i == 0)
    q = 4;
  q++;
  switch (arg.length()) {
    case 0:
      q /= 2;
      break;
    case 1:
      o = new A();
      new B();
      q = 25;
      break;
    case 2:
      o = new A();
      q = q * 100;
    default:
      o = new B();
      break;
  }
  if (arg.length() > 0) {
    o.m();
  } else {
    System.out.println("zero");
  }
  nothing.happened();
}
```

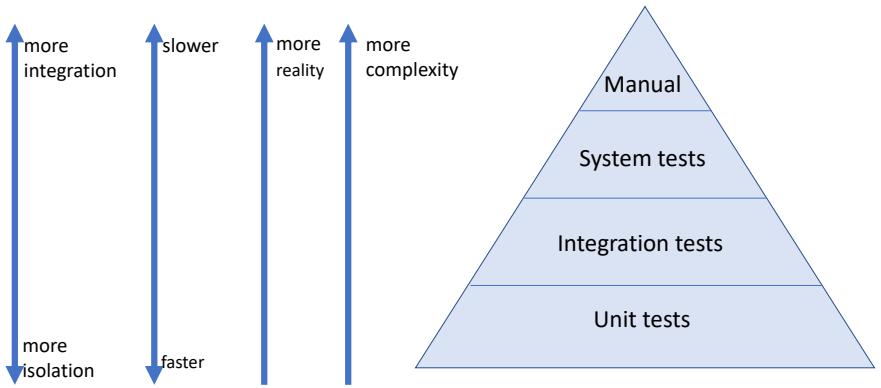

Unit Testing – Characteristics

- How many units to tests?
  - Two units: main(String[]) and m(String, int)
- What to do in each unit test?
  - When testing main(String[]), focus on the code in it
  - When testing m(String, int), focus on the code in it
  - Avoid writing duplicate assertions.
    - Do not *intensively* test m(String, int) when testing main(String[])

```
public static void main(String[] argv) {
  M obj = new M();
  if (argv.length > 0) obj.m(argv[0], argv.length);
}

public void m(String arg, int i) {
  int q = 1;
  A o = null;
  Impossible nothing = new Impossible();
  if (i == 0)
    q = 4;
  q++;
  switch (arg.length()) {
    case 0:
      q /= 2;
      break;
    case 1:
      o = new A();
      new B();
      q = 25;
      break;
    case 2:
      o = new A();
      q = q * 100;
    default:
      o = new B();
      break;
  }
  if (arg.length() > 0) {
    o.m();
  } else {
    System.out.println("zero");
  }
  nothing.happened();
}
```

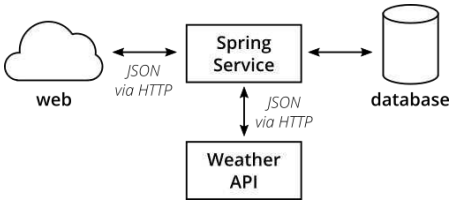

Testing Pyramid



Integration Testing

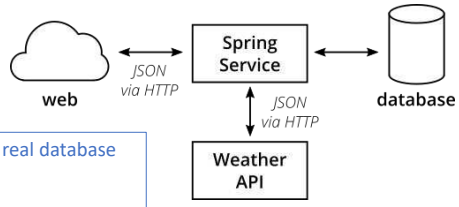
- Some non-trivial applications will integrate with some other parts
  - databases
  - file systems
  - networks calls to other applications
- Unit tests exclude these parts for isolation and fast tests.
- Integration tests test the integration of your application with all the parts that live outside of your application.

Integration Testing



https://martinfowler.com/articles/practical-test-pyramid.html

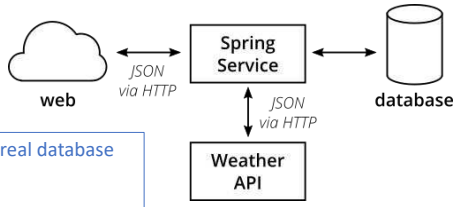
Integration Testing



A database integration test integrates your code with a real database
- start a database
- connect your application to the database
- trigger a function within your code that writes data to the database
- check that the expected data has been written to the database by reading the data from the database

https://martinfowler.com/articles/practical-test-pyramid.html

Integration Testing



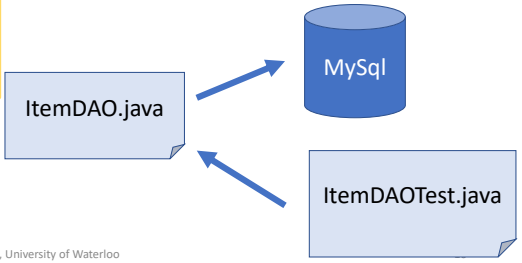
A database integration test integrates your code with a real database
- start a database
- connect your application to the database
- trigger a function within your code that writes data to the database
- check that the expected data has been written to the database by reading the data from the database

A integration test integrates your code with a web service
- start your application
- start an instance of the separate service (or a test double with the same interface, e.g., mock)
- trigger a function within your code that reads from the separate service's API
- check that your application can parse the response correctly

https://martinfowler.com/articles/practical-test-pyramid.html

Integration Testing – An Example

- We need a DB!
- Make sure the DB has the right schema
- Set up the database state (INSERTs, …)
- Make sure one test does not interfere in the other
  - Clean up everything after the test

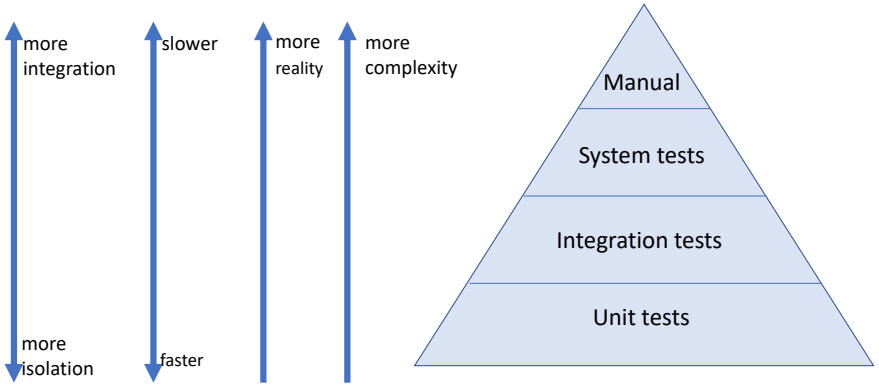


DAO is short for Data Access Object.

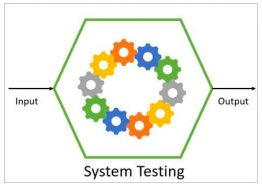
Integration Testing

- Slow
  - Need to spin up an external component (db, file system, services) as part of your tests
  - Non-in-memory communication with other components
- Flaky – tests run nondeterministically, sometimes pass sometimes fail
  - Network access
- Avoid integrating with real production system in your tests
  - production data contamination
  - unnecessary traffic to the production system
  - cost, e.g., commercial machine translation service

Testing Pyramid



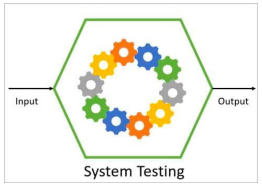
System Testing



- Test the entire system as a whole
  - End-to-end testing, E2E testing
  - Some people think that system testing is subtly different from E2E testing
- Advantages?

https://www.softwaretestinghelp.com/system-vs-end-to-end-testing/

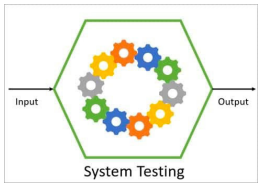
System Testing



- Test the entire system as a whole
- Advantages?
  - realistic, like how end users use the system
    - captures the user perspective
  - ensure that components work with each other

https://www.softwaretestinghelp.com/system-vs-end-to-end-testing/

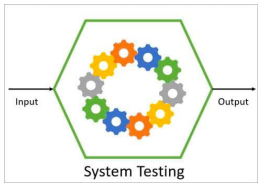
System Testing



- Test the entire system as a whole
- Advantages?
  - realistic, like how end users use the system
    - captures the user perspective
  - ensure that components work with each other
- Disadvantages?

https://www.softwaretestinghelp.com/system-vs-end-to-end-testing/

System Testing



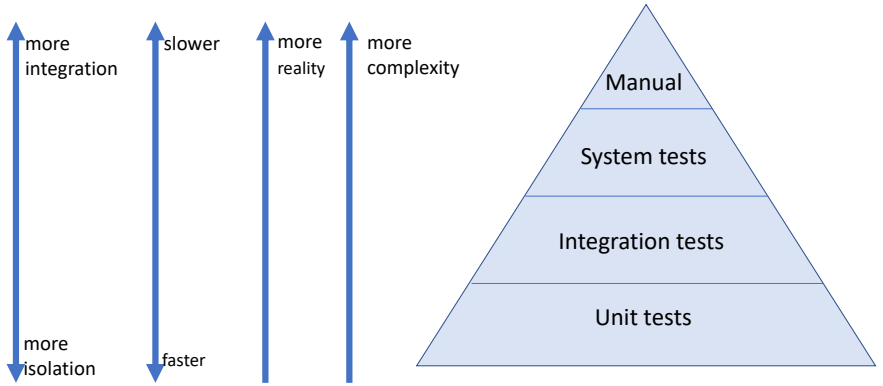
- Test the entire system as a whole
- Advantages?
  - realistic, like how end users use the system
    - captures the user perspective
  - ensure that components work with each other
- Disadvantages?
  - slow: e.g., time to start each component
  - difficult to write: automating starting each component, configuring them
  - flaky: network access
  - may not touch code deeply (compared to unit testing)
  - else?

https://www.softwaretestinghelp.com/system-vs-end-to-end-testing/

Question

- Can we replace system testing with integration testing?
- Can we replace integration testing with system testing?

Testing Pyramid

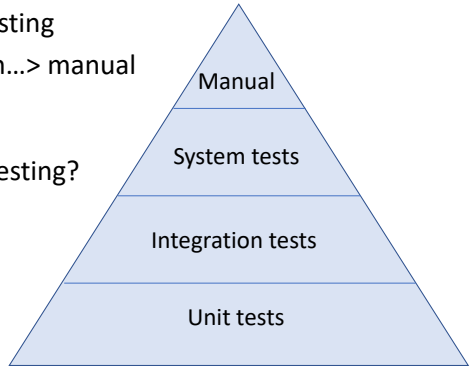


Manual Testing – Exploratory Testing

- occurs on a regular schedule
- have a destructive mindset
- come up with ways to provoke issues and errors in your application

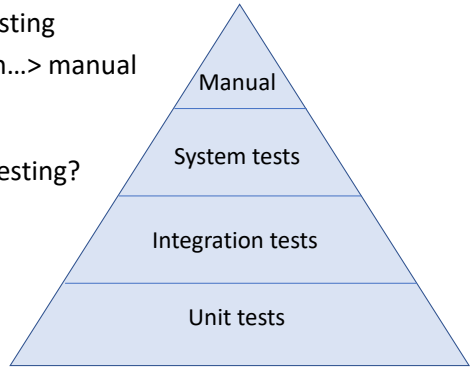
Testing Pyramid

- size: more unit testing, less other testing
- debuggability: unit test > integration…> manual
  - easy to localize faults
  - easy to use a debugger
- When spotting a fault in high level testing?



Testing Pyramid

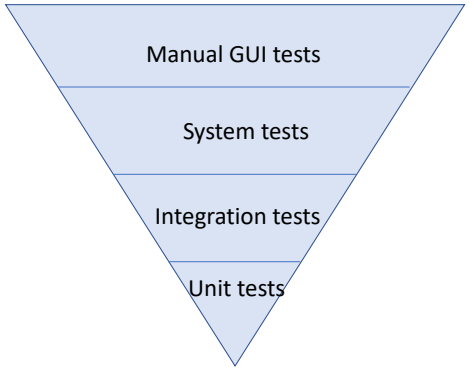
- size: more unit testing, less other testing
- debuggability: unit test > integration…> manual
  - easy to localize faults
  - easy to use a debugger
- When spotting a fault in high level testing?
  - try to trigger it with a unit test, and
  - add the unit test to the test suite.



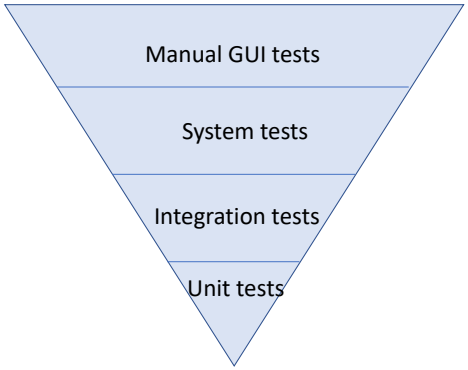
Testing Pyramid

- Unit testing
  - when you write an algorithm
- Integration testing
  - when you write code to interact with an external component
- System testing
  - when you have critical parts of the system, such as payment
- Manual testing
  - occasional

Ice Cream Cone Test



Ice Cream Cone Test – Avoid It



Unit Testing in Isolation

- When unit testing a piece of code, we want to test it in isolation.
  - Should not be in complete isolation from other classes
  - In isolation from some external dependency to run?
    - database
    - web service

Unit Testing in Isolation

- When unit testing a piece of code, we want to test it in isolation.
  - Not in complete isolation from other classes
  - In isolation from some external dependency to run?
    - database
    - web service
- What if code requires some external dependency to run?
- Mock object -- simulation of external dependency
  - a type of test doubles, (stunt doubles → test doubles)
  - benefits:
    - speed: in memory
    - control: return values, throw exceptions (db crashes)

Invoice Example

DAO: short for data access object

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

```
public List<Invoice> filter() {
    InvoiceDao dao = new InvoiceDao();
    List<Invoice> all= dao.all();
    List<> filtered = new ArrayList<>();

    for(Invoice inv : all) {
        if(inv.getValue() < 100.0)
            filtered.add(inv);
    }

    return filtered;
}
```

Invoice Example

DAO: short for data access object

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

```
public List<Invoice> filter() {
    InvoiceDao dao = new InvoiceDao();
    List<Invoice> all= dao.all();
    List<> filtered = new ArrayList<>();

    for(Invoice inv : all) {
        if(inv.getValue() < 100.0)
            filtered.add(inv);
    }

    return filtered;
}
```

DAO: short for data access object

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

```java
public List<Invoice> filter() {
    InvoiceDao dao = new InvoiceDao();
    List<Invoice> all= dao.all();
    List<> filtered = new ArrayList<>();

    for(Invoice inv : all) {
        if(inv.getValue() < 100.0)
            filtered.add(inv);
    }

    return filtered;
}
```

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

```java
@Test void filterInvoices() {
    InvoiceDao dao = new InvoiceDao();
    Invoice i1 = new Invoice("M", 20.0);
    Invoice i2 = new Invoice("A", 300.0);
    dao.save(i1); dao.save(i2);

    InvoiceFilter f = new InvoiceFilter();
    List<Invoice> result = f.filter();

    assertEquals(1, result.size());
    assertEquals(i1, result.get(0));
    dao.close();
}
```

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

```java
@Test void filterInvoices() {
    InvoiceDao dao = new InvoiceDao();
    Invoice i1 = new Invoice("M", 20.0);
    Invoice i2 = new Invoice("A", 300.0);
    dao.save(i1); dao.save(i2);

    InvoiceFilter f = new InvoiceFilter();
    List&lt;Invoice&gt; result = f.filter();

    assertEquals(1, result.size());
    assertEquals(i1, result.get(0));
    dao.close();
}
```

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

```java
@Test void filterInvoices() {
    InvoiceDao dao = new InvoiceDao();
    Invoice i1 = new Invoice("M", 20.0);
    Invoice i2 = new Invoice("A", 300.0);
    dao.save(i1); dao.save(i2);

    InvoiceFilter f = new InvoiceFilter();
    List&lt;Invoice&gt; result = f.filter();

    assertEquals(1, result.size());
    assertEquals(i1, result.get(0));
    dao.close();
}
```

I want to filter all the invoices where their value are smaller than 100.0. *Invoices come from the database.*

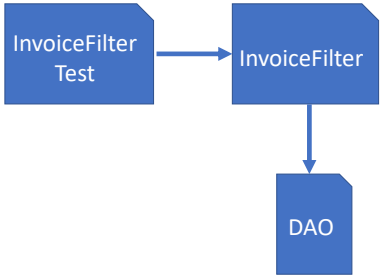
```
public List<Invoice> filter() {
    InvoiceDao dao = new InvoiceDao();
    List<Invoice> all= dao.all();
    List<> filtered = new ArrayList<>();

    for(Invoice inv : all) {
        if(inv.getValue() < 100.0)
            filtered.add(inv);
    }

    return filtered;
}
```

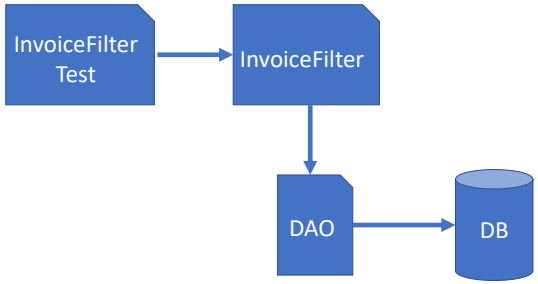
This is what we want to test!

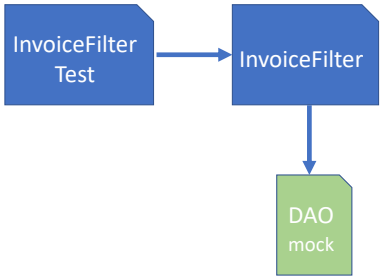
How can I get rid of the database for the sake of the test?

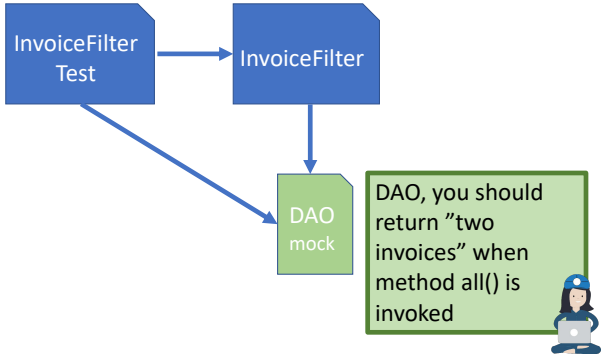
Mock Objects

- To simulate exceptions
- To simulate database access
- To simulate the interaction with any other infrastructure
- To avoid building complex objects
- To control third-party libraries that I do not own



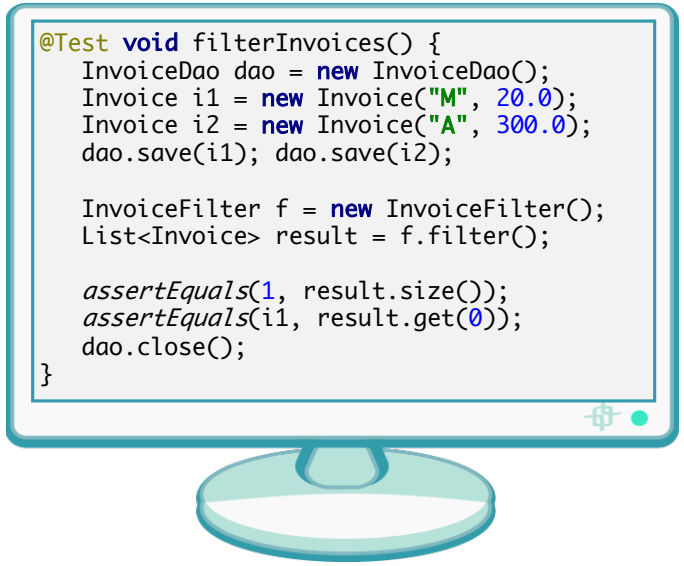

InvoiceFilter Test
InvoiceFilter
DAO
DB

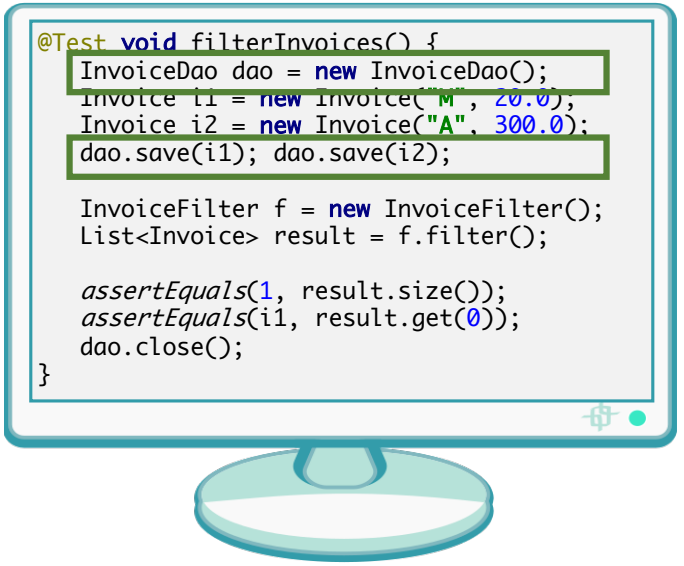

InvoiceFilter Test
InvoiceFilter
DAO mock

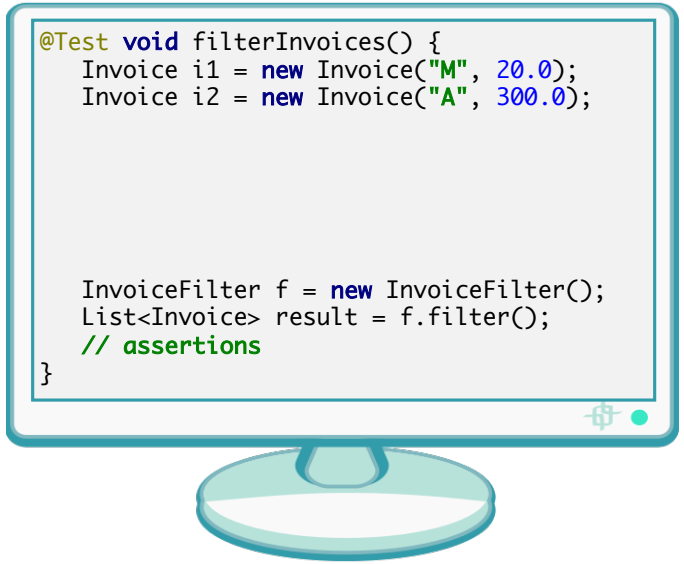

InvoiceFilter Test
InvoiceFilter
DAO mock
DAO, you should return ”two invoices” when method all() is invoked

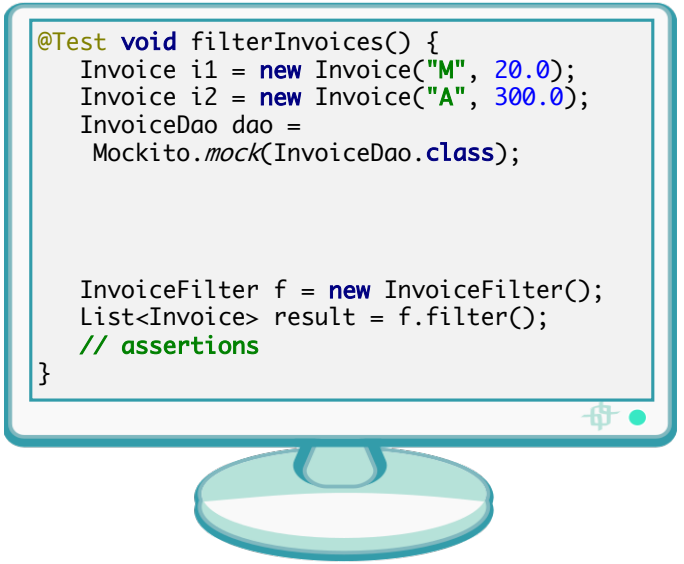

mockito


INTRO
WHY
HOW
MORE
WHO
LINKS
TRAINING
Tasty mocking framework for unit tests in Java
mockito
Current version is 2


@Test void filterInvoices() {
InvoiceDao dao = new InvoiceDao();
Invoice i1 = new Invoice("M", 20.0);
Invoice i2 = new Invoice("A", 300.0);
dao.save(i1); dao.save(i2);
InvoiceFilter f = new InvoiceFilter();
List<Invoice> result = f.filter();
assertEquals(1, result.size());
assertEquals(i1, result.get(0));
dao.close();
}


@Test void filterInvoices() {
InvoiceDao dao = new InvoiceDao();
Invoice i2 = new Invoice("A", 300.0);
dao.save(i1); dao.save(i2);
InvoiceFilter f = new InvoiceFilter();
List<Invoice> result = f.filter();
assertEquals(1, result.size());
assertEquals(i1, result.get(0));
dao.close();
}


@Test void filterInvoices() {
Invoice i1 = new Invoice("M", 20.0);
Invoice i2 = new Invoice("A", 300.0);
InvoiceFilter f = new InvoiceFilter();
List<Invoice> result = f.filter();
// assertions
}


@Test void filterInvoices() {
Invoice i1 = new Invoice("M", 20.0);
Invoice i2 = new Invoice("A", 300.0);
InvoiceDao dao =
Mockito.mock(InvoiceDao.class);
InvoiceFilter f = new InvoiceFilter();
List<Invoice> result = f.filter();
// assertions
}

```java
@Test void filterInvoices() {
   Invoice i1 = new Invoice("M", 20.0);
   Invoice i2 = new Invoice("A", 300.0);
   InvoiceDao dao =
    Mockito.mock(InvoiceDao.class);

  List<Invoice> list = Arrays.asList(i1,i2);
   Mockito.when(dao.all()).thenReturn(list);

   InvoiceFilter f = new InvoiceFilter();
   List<Invoice> result = f.filter();
   // assertions
}
```

What is the problem with the original method?

```java
public List<Invoice> filter() {
    InvoiceDao dao = new InvoiceDao();
    List<Invoice> all= dao.all();
    List<> filtered = new ArrayList<>();

    for(Invoice inv : all) {
        if(inv.getValue() < 100.0)
            filtered.add(inv);
    }

    return filtered;
}
```

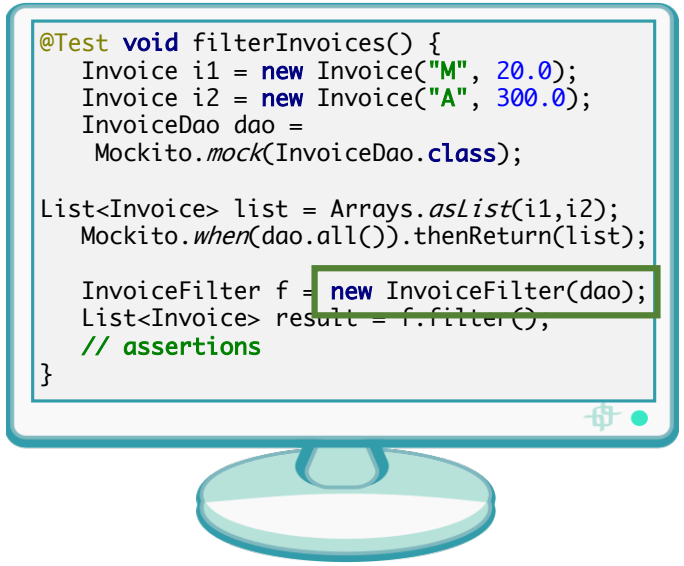
How can I let the filter use the mock DAO?

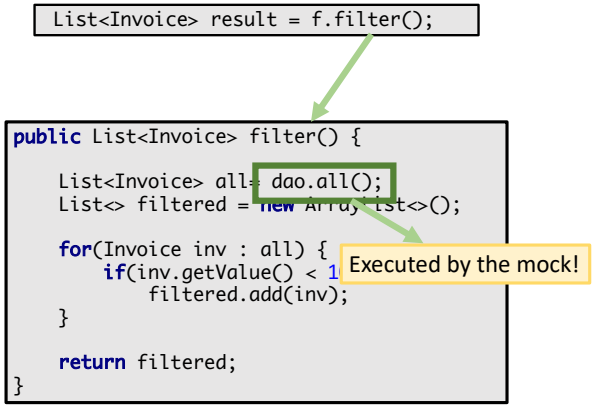
```java
InvoiceDao dao =
    Mockito.mock(InvoiceDao.class);
```

Lifting local variable “dao” to a field.

```java
public List<Invoice> filter() {

    List<Invoice> all= dao.all();
    List<> filtered = new ArrayList<>();

    for(Invoice inv : all) {
        if(inv.getValue() < 100.0)
            filtered.add(inv);
    }

    return filtered;
}
```

```java
public class InvoiceFilter {

   private InvoiceDao dao;

   public InvoiceFilter
    (InvoiceDao dao) {
      this.dao = dao;
   }

   public List<Invoice> filter() { … }
```





Don't Overuse Mocks

Overusing mocks can cause several problems

- tests can be harder to understand

```
public void testCreditCardIsCharged() {
  paymentProcessor = new PaymentProcessor(mockCreditCardServer);
  when(mockCreditCardServer.isServerAvailable()).thenReturn(true);
  when(mockCreditCardServer.beginTransaction()).thenReturn(mockTransactionManager);
  when(mockTransactionManager.getTransaction()).thenReturn(transaction);
  when(mockCreditCardServer.pay(transaction, creditCard, 500)).thenReturn(mockPayme
  when(mockPayment.isOverMaxBalance()).thenReturn(false);
  paymentProcessor.processPayment(creditCard, Money.dollars(500));
  verify(mockCreditCardServer).pay(transaction, creditCard, 500);
}
```

https://testing.googleblog.com/2013/05/testing-on-toilet-dont-overuse-mocks.html

Don't Overuse M

```
public void testCreditCardIsCharged() {
  paymentProcessor = new PaymentProcessor(creditCardServer);
  paymentProcessor.processPayment(creditCard, Money.dollars(500));
  assertEquals(500, creditCardServer.getMostRecentCharge(creditCard));
}
```

Overusing mocks can cause several problems

- tests can be harder to understand


```
public void testCreditCardIsCharged() {
  paymentProcessor = new PaymentProcessor(mockCreditCardServer);
  when(mockCreditCardServer.isServerAvailable()).thenReturn(true);
  when(mockCreditCardServer.beginTransaction()).thenReturn(mockTransactionManager);
  when(mockTransactionManager.getTransaction()).thenReturn(transaction);
  when(mockCreditCardServer.pay(transaction, creditCard, 500)).thenReturn(mockPayme
  when(mockPayment.isOverMaxBalance()).thenReturn(false);
  paymentProcessor.processPayment(creditCard, Money.dollars(500));
  verify(mockCreditCardServer).pay(transaction, creditCard, 500);
}
```

https://testing.googleblog.com/2013/05/testing-on-toilet-dont-overuse-mocks.html

Don’t Overuse Mocks

Overusing mocks can cause several problems

- tests can be harder to understand
- tests can be harder to maintain
  - leaking implementation details
- tests provide less assurance that your code is working properly
  - what you specified in the mock object is different from the real implementation.

https://testing.googleblog.com/2013/05/testing-on-toilet-dont-overuse-mocks.html

Don’t Overuse Mocks

Overusing mocks can cause several problems

- tests can be harder to understand
- tests can be harder to maintain
  - leaking implementation details
- tests provide less assurance that your code is working properly
  - what you specified in the mock object is different from the real implementation.

- Alternative: fake implementation, start a local server only for testing

https://testing.googleblog.com/2013/05/testing-on-toilet-dont-overuse-mocks.html

Unit Testing: The Myth of Complete Isolation

https://dzone.com/articles/unit-testing-the-myth-of-complete-isolation

Complete Isolation

```java
public class Order {
    private List<OrderItem> items;

    // c-tor

    public BigDecimal getTotalPrice() {
        return items.stream()
                .map(item -> item.getPrice().multiply(item.getQuantity()))
                .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}

class OrderItem {
    private BigDecimal price;
    private BigDecimal quantity;

    // c-tor, getters
}
```

```java
public class FragileOrderTest {

    @Test
    public void shouldSumItemPrices() throws Exception {
        OrderItem item1 = orderItem(2, 3);
        OrderItem item2 = orderItem(4, 5);
        Order order = new Order(asList(item1, item2));
        assertEquals(BigDecimal.valueOf(26), order.getTotalPrice());
    }

    private OrderItem orderItem(int price, int quantity) {
        OrderItem item = mock(OrderItem.class);
        when(item.getPrice()).thenReturn(BigDecimal.valueOf(price));
        when(item.getQuantity()).thenReturn(BigDecimal.valueOf(quantity));
        return item;
    }
}
```

https://dzone.com/articles/unit-testing-the-m

te Isolation

```java
public class Order {
    // same as before

    public BigDecimal getTotalPrice() {
        return items.stream()
                // it's OrderItem's responsibility to do the multiplication!
                .map(OrderItem::getTotalPrice)
                .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```

```java
public class FragileOrderTest {

    @Test
    public void shouldSumItemPrices() throws Exception {
        OrderItem item1 = orderItem(2, 3);
        OrderItem item2 = orderItem(4, 5);
        Order order = new Order(asList(item1, item2));
        assertEquals(BigDecimal.valueOf(26), order.getTotalPrice());
    }

    private OrderItem orderItem(int price, int quantity) {
        OrderItem item = mock(OrderItem.class);
        when(item.getPrice()).thenReturn(BigDecimal.valueOf(price));
        when(item.getQuantity()).thenReturn(BigDecimal.valueOf(quantity));
        return item;
    }
}
```

https://dzone.com/articles/unit-testing-the-m

te Isolation


```java
public class Order {
    // same as before

    public BigDecimal getTotalPrice() {
        return items.stream()
                // it's OrderItem's responsibility to do the multiplication!
                .map(OrderItem::getTotalPrice)
                .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```


```java
public class RobustOrderTest {
    // same as before:
    @Test
    public void shouldSumItemPrices() throws Exception {
        OrderItem item1 = orderItem(2, 3);
        OrderItem item2 = orderItem(4, 5);
        Order order = new Order(asList(item1, item2));
        assertEquals(BigDecimal.valueOf(26), order.getTotalPrice());
    }

    // returns real objects, not mocks! we should not care about the methods calls here!
    private OrderItem orderItem(int price, int quantity) {
        return new OrderItem(BigDecimal.valueOf(price), BigDecimal.valueOf(quantity));
    }
}
```

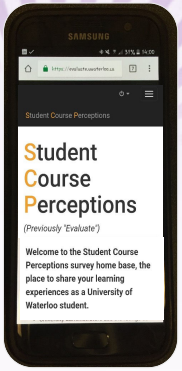
https://dzone.com/articles/unit-testing-the-myth-of-complete-isolation

https://perceptions.uwaterloo.ca

Hey, Engineering students!

Did you know that course evaluations results are used to improve your courses and your programs?

Help us improve your student experience.



Your feedback counts.
Complete your Winter 2023 course evaluations by Sunday, April 2!

Self Testing / Assertions

Chengnian Sun
cnsun@

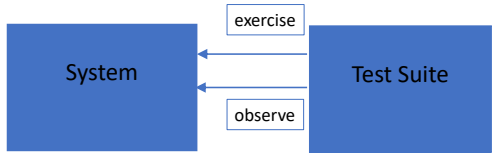
*Slides adapted from sttp.site

Self Testing

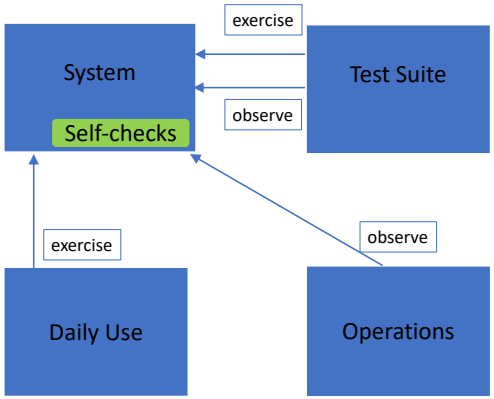
Can the production code test itself?

Can a program inspect the states of itself, and determine whether there is a bug?

Self Testing



Self Testing



The Java (C, C++, Python, …) assert Statement

```
"assert" boolean-expression [":" string ]
```

If *boolean-expression* is true, do nothing.

If it is false, raise an `AssertionError`, with the string as message

Assert by Example

```
public class MyStack() {

  public Element pop() {
    assert count() > 0;
    ....
    // real method body here
    ....
    assert count() == oldCount – 1;
  }
}
```

Can also be written as :

```
assert count() > 0: count();
assert count() == oldCount - 1 : "count=" + count() + ", oldCount=" + oldCount;
```


Assertion Checking can be Enabled or Disabled

- Assertions in Java are optional
  - java –enableassertions ……
  - java –ea ……
  - Eclipse: add jvm argument to run configuration
  - Maven/IntelliJ: enabled by default when executing tests
  - If assertions are disabled, the predicates are NOT evaluated at all.
- Pros?

Assertion Checking can be Enabled or Disabled

- Assertions in Java are optional
  - java –enableassertions ……
  - java –ea ……
  - Eclipse: add jvm argument to run configuration
  - Maven/IntelliJ: enabled by default when executing tests
  - If assertions are disabled, the predicates are NOT evaluated at all.
- Pros?
  - configurable, can be enabled and disabled without recompilation
  - can be disabled in production, without affecting runtime performance

Assertion Checking can be Enabled or Disabled

- Assertions in Java are optional
  - java –enableassertions ……
  - java –ea ……
  - Eclipse: add jvm argument to run configuration
  - Maven/IntelliJ: enabled by default when executing tests
  - If assertions are disabled, the predicates are NOT evaluated at all.
- Pros?
  - configurable, can be enabled and disabled without recompilation
  - can be disabled in production, without affecting runtime performance
- Cons?

Assertion Checking can be Enabled or Disabled

- Assertions in Java are optional
  - java –enableassertions ……
  - java –ea ……
  - Eclipse: add jvm argument to run configuration
  - Maven/IntelliJ: enabled by default when executing tests
  - If assertions are disabled, the predicates are NOT evaluated at all.
- Pros?
  - configurable, can be enabled and disabled without recompilation
  - can be disabled in production, without affecting runtime performance
- Cons?
  - can have side effects in the predicate of the assertion
  - can be disabled in production

Assertion Checking can be Enabled or Disabled

```
int increment(int i) {
  final int oldValue = i;
  assert ++i == oldValue + 1;
  return i;
}
```

- Assertions in Java are optional
  - java –enableassertions ……
  - java –ea ……
  - Eclipse: add jvm argument to run configuration
  - Maven/IntelliJ: enabled by default when executing tests
  - If assertions are disabled, the predicates are NOT evaluated at all.
- Pros?
  - configurable, can be enabled and disabled without recompilation
  - can be disabled in production, without affecting runtime performance
- Cons?
  - can have side effects in the predicate of the assertion
  - can be disabled in production

Assertion Checking can be Enabled or Disabled

```
int increment(int i) {
  final int oldValue = i;
  assert ++i == oldValue + 1;
  return i;
}
```

- Assertions in Java are optional
  - java –enableassertions ……
  - java –ea ……
  - Eclipse: add jvm argument to run configuration
  - Maven/IntelliJ: enabled by default when executing tests
  - If assertions are disabled, the predicates are NOT evaluated at all.
- Pros?
  - configurable, can be enabled and disabled without recompilation
  - can be disabled in production, without affecting runtime performance
- Cons?
  - can have side effects in the predicate of the assertion
  - can be disabled in production

Program must run correctly with assertions disabled!

Assertions Defined

An assertion is a Boolean expression
at a specific point in a program
which will be true
unless there is a bug in the program.

http://wiki.c2.com/?WhatAreAssertions

Assertions Defined

Assertion for third-party libraries?

An assertion is a Boolean expression
at a specific point in a program
which will be true
unless there is a bug in the program.

http://wiki.c2.com/?WhatAreAssertions

Example – Assertions for Using Third-Party Lib

```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```


Example – Assertions for Us

```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```

```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
// assumption: an animal is either a cat or a dog.
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```


Example – Assertions for Us

```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```

```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```

What is the benefit of putting an assertion there?

Example – Assertions for Us

```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```

```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```

What is the benefit of putting an assertion there?

Answer: Imagine the library introduces a new animal type.


```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```



```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```



```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```


```
-- your code: version 2 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```


```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```



```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```



```
-- your code: version 2 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```


```
-- your code: version 3 --
Animal animal = retrieveInLibrary(...);
// assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```


```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```

```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```

```
-- your code: version 2 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```

```
-- your code: version 3 --
Animal animal = retrieveInLibrary(...);
// assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```

```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```

```
-- your code: version 4 --
Animal animal = retrieveInLibrary(...);
if (animal instanceof Cat) {
  //...
} else if (animal instanceof Dog) {
  //...
} else {
  assert false : "unreachable: " + animal;
}
```



```
-- your code: version 1 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  // assumption: this is a dog.
}
```

```
-- your code: version 2 --
Animal animal = retrieveInLibrary(...);
assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```

```
-- your code: version 3 --
Animal animal = retrieveInLibrary(...);
// assert animal instanceof Cat || animal instanceof Dog;
if (animal instanceof Cat) {
  //...
} else {
  assert animal instanceof Dog;
  //...
}
```

```
-- library --
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

public static Animal retrieveInLibrary(...);
```

```
-- your code: version 4 --
Animal animal = retrieveInLibrary(...);
if (animal instanceof Cat) {
  //...
} else if (animal instanceof Dog) {
  //...
} else {
  assert false : "unreachable: " + animal;
}
```


```
-- your code: version 5 --
Animal animal = retrieveInLibrary(...);
if (animal instanceof Cat) {
  //...
} else if (animal instanceof Dog) {
  //...
} else {
  throw new RuntimeException("unreachable:" + animal);
}
```


Run-Time Assertions in Software Testing

Test Oracles

"Software that applies a pass/fail criterion to a program execution is called a *(test) oracle".*

Approaches

1. Value comparison – input/output
2. Version comparisons – differential testing
3. *Property checks*
   - *e.g., s1.length() + s2.length() = (s1 + s2).length()*

In-Code Assertions as Oracles

- Enable run time assertion checking during testing
  - Post-conditions check method outcomes
  - Pre-conditions check correct method usage
  - Invariants check object health
- Run time assertions increase *fault sensitivity*
  - Increase likelihood program fails if there is a fault
  - Desirable during testing!

Assertions don't Replace Testing

- Unit tests still needed to *exercise* methods
- In-code assertions only check **general** properties
  - assertions should pass wrt all valid inputs.
- In-code assertions *on top of* asserts with concrete expected values in tests

Assertions Inspire Testing

- Test inputs should *reach* assertions
  - eg., assertions in compilers.
- Assertions may be *disjunction* P1 or P2
  - Test inputs should trigger both alternatives
- Assertions may contain *boundaries*
  - Test inputs should trigger those boundaries

Assertions Don't Fail

- Test inputs can reach assertions
- Test inputs cannot make assertions fail
  - *That would be a bug in the program!*
- No need to write test cases that let pre-conditions fail
  - *Method behavior* undefined *!*

Should assertion be used in test code?

Should assertion be used in test code?

- No. You tests will always pass if assertions are disabled.
- Prefer assertion functions in Junit.
  - other assertion libraries, e.g., AssertionJ, Google Truth.

Asserting Invariants – Preconditions on Public Methods

Preconditions on public methods should be enforced by explicit checks inside methods resulting in particular, specified exceptions.

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);

    setRefreshInterval(1000 / rate);
}
```

 https://docs.oracle.com/cd/E19683-01/806-7930/assert-13/index.html

Asserting Invariants – Preconditions on Public Methods

Preconditions on public methods should be enforced by explicit checks inside methods resulting in particular, specified exceptions.

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);

    setRefreshInterval(1000 / rate);
}
```

Questions: Can you use “assert” to check the precondition?

 https://docs.oracle.com/cd/E19683-01/806-7930/assert-13/index.html

Asserting Invariants – Preconditions on Public Methods

Preconditions on public methods should be enforced by explicit checks inside methods resulting in particular, specified exceptions.

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);

    setRefreshInterval(1000 / rate);
}
```

Questions: Can you use "assert" to check the precondition?

Answer: Probably no. An assert is inappropriate for such a precondition on a public method. Assertions can be disabled.

Problem with the first assertion?

```
public class MyStack() {

  public Element pop() {
    assert count() > 0;
    ....
    // real method body here
    ....
    assert count() == oldCount - 1;
  }
}
```


Problem with the first assertion?

```
public class MyStack() {

  public Element pop() {
    assert count() > 0;
    ....
    // real method body here
    ....
    assert count() == oldCount - 1;
  }
}
```

```
public class MyStack() {

  public Element pop() {
    if(count() <= 0) {throw …;}
    ....
    // real method body here
    ....
    assert count() == oldCount - 1;
  }
}
```


Asserting Invariants – Preconditions on Public Methods

Preconditions on public methods should be enforced by explicit checks inside methods resulting in particular, specified exceptions.

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);

    setRefreshInterval(1000 / rate);
}
```

Questions: Can you use "assert" to check the precondition?

Answer: Probably no. An assert is inappropriate for such a precondition on a public method. Assertions can be disabled.

Any exceptions where assert is preferred?

Asserting Invariants – Preconditions on Public Methods

Preconditions on public methods should be enforced by explicit checks inside methods resulting in particular, specified exceptions.

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);

    setRefreshInterval(1000 / rate);
}
```

Questions: Can you use “assert” to check the precondition?

Answer: Probably no. An assert is inappropriate for such a precondition on a public method. Assertions can be disabled.

Any exceptions where assert is preferred?
If the predicate is expensive to be checked, then you can consider using assert which can be disabled in production.

Exercise – Precondition

Write assertions for this example.

This algorithm merges two sorted arrays into a single sorted array.

Assume we have a utility method `isSorted(int[])`

```
1  public static int[] mergeSortedArrays(int[] a, int[] b) {
2    int[] result = new int[a.length + b.length];
3    int ai = 0;
4    int bi = 0;
5    for (int i = 0;
6         i < result.length;
7         ++i) {
8      if (ai < a.length
9          && bi < b.length) {
10       if (a[ai] <= b[bi]) {
11         result[i] = a[ai++];
12       } else {
13         result[i] = b[bi++];
14       }
15     } else if (ai < a.length) {
16       result[i] = a[ai++];
17     } else {
18       result[i] = b[bi++];
19     }
20   }
21   return result;
22 }
```


Asserting Invariants – Preconditions on non-public Methods

If there is a precondition on a nonpublic method and the author of a class believes that the precondition to hold no matter what a client does with the class, then an assertion is entirely appropriate.

```
/**
 * Sets the refresh interval (must correspond to a legal frame rate).
 *
 * @param  interval refresh interval in milliseconds.
 */
private void setRefreshInterval(int interval) {
    // Confirm adherence to precondition in nonpublic method
    assert interval > 0 && interval <= 1000 / MAX_REFRESH_RATE;

    ... // Set the refresh interval
}
```

If the assertion fails, then the failure indicates a bug.

Asserting Invariants – Postconditions

Postcondition checks are best implemented via assertions, whether or not they are specified in public methods.

```
public BigInteger modInverse(BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("Modulus not positive: " + m);
    if (!this.gcd(m).equals(ONE))
        throw new ArithmeticException(this + " not invertible mod " + m);

    ... // Do the computation

    assert this.multiply(result).mod(m).equals(ONE);
    return result;
}
```

Exercise – Postcondition

Write assertions for this example.

This algorithm merges two sorted arrays into a single sorted array. Neither a nor b is nullable.

Assume we have a utility method
`isSorted(int[])`

```
1  public static int[] mergeSortedArrays(int[] a, int[] b) {
2    int[] result = new int[a.length + b.length];
3    int ai = 0;
4    int bi = 0;
5    for (int i = 0;
6         i < result.length;
7         ++i) {
8      if (ai < a.length
9          && bi < b.length) {
10       if (a[ai] <= b[bi]) {
11         result[i] = a[ai++];
12       } else {
13         result[i] = b[bi++];
14       }
15     } else if (ai < a.length) {
16       result[i] = a[ai++];
17     } else {
18       result[i] = b[bi++];
19     }
20   }
21   return result;
22 }
```


Asserting Invariants – Class Invariants

It is sometimes convenient to combine many expressions that check required constraints into a single internal method that can then be invoked by assertions. For example, suppose one were to implement a balanced tree data structure of some sort. It might be appropriate to implement a private method that checked that the tree was indeed balanced as per the dictates of the data structure:

```
// Returns true if this tree is properly balanced
private boolean balanced() {
    ...
}


assert balanced();
```


Asserting Invariants – Control-Flow Invariants

```
switch (suit) {
    case Suit.CLUBS:
        ...
        break;

    case Suit.DIAMONDS:
        ...
        break;

    case Suit.HEARTS:
        ...
        break;

    case Suit.SPADES:
        ...
}
```

```
switch (suit) {
    case Suit.CLUBS:
        ...
        break;

    case Suit.DIAMONDS:
        ...
        break;

    case Suit.HEARTS:
        ...
        break;

    case Suit.SPADES:
        ...
    default:
        assert false;
}
```

```
switch (suit) {
    case Suit.CLUBS:
        ...
        break;

    case Suit.DIAMONDS:
        ...
        break;

    case Suit.HEARTS:
        ...
        break;

    case Suit.SPADES:
        ...
    default:
        throw ……;
}
```

Prefer throw over assert.
- No performance difference.
- Deterministic

Asserting Invariants – Control-Flow Invariants

```
void foo() {
    for (...) {
        if (...)
            return;
    }
    // Execution should never reach this point!!!
}
```

Prefer throw over assert.
- No performance difference.
- Deterministic

Asserting Invariants – Control-Flow Invariants

```
void foo() {
    for (...) {
        if (...)
            return;
    }
    // Execution should never reach this point!!!
    // Add a throw statement to replace the comment
    // Using 'assert false' is not good.
    throw new RuntimeException("Unreachable");
}
```

Prefer throw over assert.
- No performance difference.
- Deterministic

Asserting Invariants – Internal Invariants

- In general, it is appropriate to frequently use short assertions indicating important assumption concerning a program's behavior.

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // (i%3 == 2)
    ...
}
```


Asserting Invariants – Internal Invariants

- In general, it is appropriate to frequently use short assertions indicating important assumption concerning a program's behavior.

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // (i%3 == 2)
    ...
}
```

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i%3 == 2;
    ...
}
```

Assertions are better at documenting assumptions.

Exercise – Internal Invariant (loop invariant)

Write assertions for this example.

This algorithm merges two sorted arrays into a single sorted array. Neither a nor b is nullable.

Assume we have a utility method `isSorted(int[])`

```
1  public static int[] mergeSortedArrays(int[] a, int[] b) {
2    int[] result = new int[a.length + b.length];
3    int ai = 0;
4    int bi = 0;
5    for (int i = 0;
6         i < result.length;
7         ++i) {
8      if (ai < a.length
9          && bi < b.length) {
10       if (a[ai] <= b[bi]) {
11         result[i] = a[ai++];
12       } else {
13         result[i] = b[bi++];
14       }
15     } else if (ai < a.length) {
16       result[i] = a[ai++];
17     } else {
18       result[i] = b[bi++];
19     }
20   }
21   return result;
22 }
```

Test Code Quality

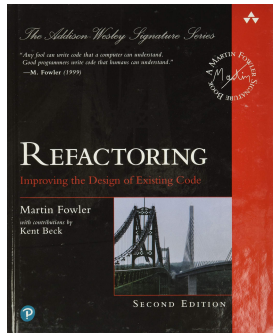
Chengnian Sun
cnsun@

*Slides adapted from sttp.site

Code Smell

Any characteristic in the source code that possibly indicates a deeper problem.

- code duplication (copy-paste)
- long method, large class
- down-casting
- too many parameters



Test Code Smell – smell in test code

- growing test code base
- test code evolution and maintenance

Good Properties of Test Code – [F]IRST

Fast

- Important to run tests fast after each change
- minimize dependencies, e.g., mocks to replace external resources

Good Properties of Test Code – F[I]RST

Isolated

- The test should check an isolated piece of functionality.
- The test should be isolated from other tests.

- Do not assume the execution ordering of tests in the test set.

Good Properties of Test Code – FI[R]ST

Repeatable

- A repeatable test is a test that gives the same result, no matter how many times it is executed.
- A test that is not so well isolated from external resources tend to be not repeatable.

Good Properties of Test Code – FIR[S]T

Self-validating/Self-arranging

- The tests should validate the result themselves.
- Each test should set up/arrange its data itself.

Good Properties of Test Code – FIRS[T]

Timely

- Write tests shortly after you finish a piece of production code.
- Do not wait until you finish all production code.

Test Code Smell

- Duplicate code

Test Code Smell

`void sort(int[])`

- Duplicate code
- Too many assertions / complex assertions

```
@Test
public void testSort() {
  int[] empty = new int[];
  sort(empty);
  assertEquals(new int[0], empty);
  int[] single = new int[]{1};
  sort(single);
  assertEquals(new int[]{1}, single);
  int[] two = new int[]{2, 1};
  sort(two);
  assertEquals(new int[]{1,2}, two);
}
```


Test Code Smell

`void sort(int[])`

- Duplicate code
- Too many assertions / complex assertions

```
@Test
public void testSort() {
  int[] empty = new int[];
  sort(empty);
  assertEquals(new int[0], empty);
  int[] single = new int[]{1};
  sort(single);
  assertEquals(new int[]{1}, single);
  int[] two = new int[]{2, 1};
  sort(two);
  assertEquals(new int[]{1,2}, two);
}
```

```
@Test
public void testSort_empty() {
  int[] empty = new int[];
  sort(empty);
  assertEquals(new int[0], empty);
}

@Test
public void testSort_single() {
  int[] single = new int[]{1};
  sort(single);
  assertEquals(new int[]{1}, single);
}

@Test
public void testSort_two() {
  int[] two = new int[]{2, 1};
  sort(two);
  assertEquals(new int[]{1,2}, two);
}
```


Test Code Smell

- Duplicate code
- Too many assertions / complex assertions
- Slow tests

Test Code Smell

- Duplicate code
- Too many assertions / complex assertions
- Slow tests
- Resource optimism
  - A test should not assume a resource, e.g., db, is ready for use.
  - Instead, it should set up the db itself, or mock objects, to ensure determinism.
  - Or, if the resource is unavailable, skip the test and print a message, or just fail.

Test Code Smell

- Duplicate code
- Too many assertions / complex assertions
- Slow tests
- Resource optimism
  - A test should not assume a resource, e.g., db, is ready for use.
  - Instead, it should set up the db itself, or mock objects, to ensure determinism.
  - Or, if the external resource is unavailable, skip the test and print a message.
- Global mutable data shared among tests. Problems?

Flaky Tests

- Tests that sometimes pass, sometimes fail.

Flaky Tests

- Tests that sometimes pass, sometimes fail.

- Possible causes
  - external infrastructure, e.g., the availability of a database server, on/off
  - global shared data, e.g., files shared by tests
  - timeouts
  - what else?

Flaky Tests

- Tests that sometimes pass, sometimes fail.

- Possible causes
  - external infrastructure, e.g., the availability of a database server, on/off
  - global shared data, e.g., files shared by tests
  - timeouts
  - what else?
    - concurrency
    - hashCode

Testability

Chengnian Sun
cnsun@

*Slides adapted from sttp.site

Testability

```
public List<Invoice> filter() {
  InvoiceDAO dao = new InvoiceDAO();
  List<Invoice> all= dao.all();
  List<Invoice> filtered
      = new ArrayList<>();

  for(Invoice inv : all) {
    if(inv.getValue() < 100.0)
      filtered.add(inv);
  }
  return filtered;
}
```

The code looks fine without obvious code smells. However, suppose we want to test the code with mock objects, the problem surfaces.

Testability

```
public List<Invoice> filter() {
  InvoiceDAO dao = new InvoiceDAO();
  List<Invoice> all= dao.all();
  List<Invoice> filtered
      = new ArrayList<>();

  for(Invoice inv : all) {
    if(inv.getValue() < 100.0)
      filtered.add(inv);
  }
  return filtered;
}
```

```
public class InvoiceFilter {

  private final InvoiceDao dao;

  public InvoiceFilter(InvoiceDao dao) {
    this.dao = dao;
  }

  public List<Invoice> filter() {
    // InvoiceDAO dao = new InvoiceDAO();
    List<Invoice> all= this.dao.all();
    List<Invoice> filtered
        = new ArrayList<>();

    for(Invoice inv : all) {
      if(inv.getValue() < 100.0)
        filtered.add(inv);
    }
    return filtered;
  }
}
```


Testability – Exercise

```
public double applyDiscount(double rawAmount) {
  Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

Testability – Exercise

```
public double applyDiscount(double rawAmount) {
  Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

1. Use ***interface-based*** input space modeling to design tests for this method.

Testability – Exercise

```
public double applyDiscount(double rawAmount) {
  Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

Date
- Christmas
- Not Christmas

rawAmount
- Zero
- Negative
- Positive

1. Use ***interface-based*** input space modeling to design tests for this method.

Testability – Exercise

```
public double applyDiscount(double rawAmount) {
  Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

Date
- Christmas
- Not Christmas

rawAmount
- Zero
- Negative
- Positive

1. Use ***interface-based*** input space modeling to design tests for this method.
2. Write tests for this method.

Testability – Exercise

```
public double applyDiscount(double rawAmount) {
  Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

Date
- Christmas
- Not Christmas

rawAmount
- Zero
- Negative
- Positive

Any problem?

1. Use ***interface-based*** input space modeling to design tests for this method.
2. Write tests for this method.

Testability – Exercise

```
public double applyDiscount(double rawAmount) {
  Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

Date
- Christmas
- Not Christmas

rawAmount
- Zero
- Negative
- Positive

Any problem?

Controllability: always gets the current date!

1. Use ***interface-based*** input space modeling to design tests for this method.
2. Write tests for this method.

Testability

```
public double applyDiscount(
    Calendar today, double rawAmount) {
  // Calendar today = Calendar.getInstance();
  double discountPercentage = 0;
  boolean isChristmas =
      today.get(Calendar.MONTH) == Calendar.DECEMBER
          && today.get(Calendar.DAY_OF_MONTH) == 25;

  if (isChristmas) {
    discountPercentage = 0.15;
  }
  return rawAmount - (rawAmount * discountPercentage);
}
```

Date
- Christmas
- Not Christmas

rawAmount
- Zero
- Negative
- Positive

Dependency Injection

A class needs a dependency to accomplish something

- Option 1: the class instantiates the dependency
- **Option 2**: the class gets the dependency from outside (via constructor or a setter, for example)

Dependency Injection

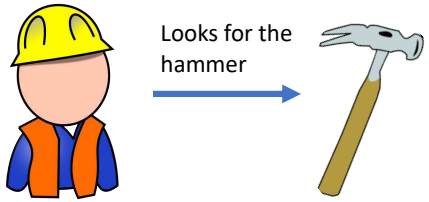
```
public List<Invoice> filter() {
  InvoiceDAO dao = new InvoiceDAO();
  List<Invoice> all= dao.all();
  List<Invoice> filtered
      = new ArrayList<>();

  for(Invoice inv : all) {
    if(inv.getValue() < 100.0)
      filtered.add(inv);
  }
  return filtered;
}
```

Option 1: the class instantiates the dependency

```
public class InvoiceFilter {
  private final InvoiceDao dao;

  public InvoiceFilter(InvoiceDao dao) {
    this.dao = dao;
  }

  public List<Invoice> filter() {
    // InvoiceDAO dao = new InvoiceDAO();
    List<Invoice> all= this.dao.all();
    List<Invoice> filtered
        = new ArrayList<>();
    for(Invoice inv : all) {
      if(inv.getValue() < 100.0)
        filtered.add(inv);
    }
  }
}
```

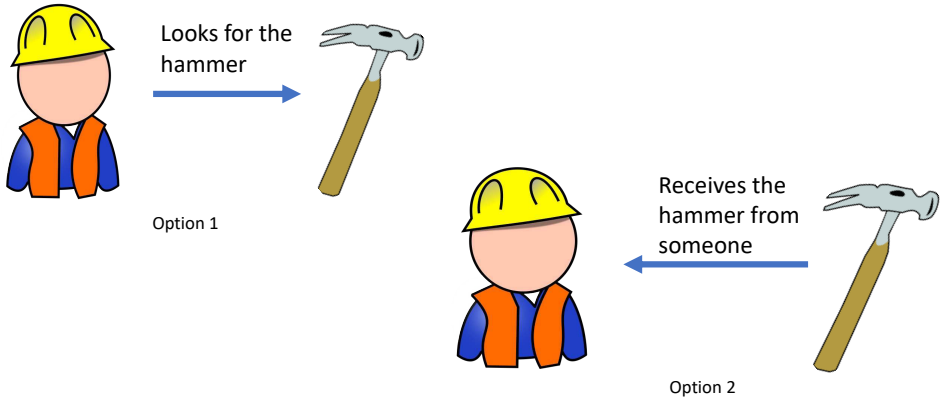
Option 2: the class gets the dependency from outside (via constructor or a setter, for example)

Dependency Injection



Option 1

Dependency Injection

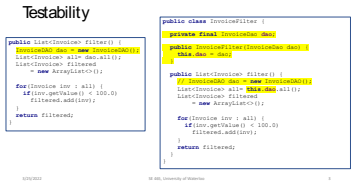


Option 1

Option 2

Benefits of Dependency Injection

- Better testability (controllability)
  - We can mock the dependencies.
- Makes all dependencies explicit
  - constructors, setters
- The class becomes more extensible.
  - Limit the impact of changes in dependencies on the class
  - The class talks with its dependencies via interfaces, not concrete implementation
  - For example, `class InvoiceDAO2 extends InvoiceDAO`, no need to change in the right version, but not in the left version.



Better Testability / Better Design?

- A relationship between how I design my code and how easy it is to test it.

- Better Design → Better Testability?
- Better Testability → Better Design?

Example

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
- hasMoreTokens()
- getNextToken()

```
new ExprEvaluator("1+2*3").evaluate()

expr : expr "+" term
expr : term
term : term "*" primary
term : primary
primary: "(" expr ")"
primary: INTEGER
```


Example

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
- hasMoreTokens()
- getNextToken()

UML Class diagram
- Each row is a method or field
- `+: public`
- `-: private`
- `#: protected`

Lexing
- `hasMoreTokens()`
- `getNextToken()`

Parsing
- `parse()`
- `parseTerm()`
- `parsePrimary()`

Example

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
- hasMoreTokens()
- getNextToken()

Do we need to write tests for hasMoreTokens() and getNextToken()?

Example

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
- hasMoreTokens()
- getNextToken()

Do we need to write tests for hasMoreTokens() and getNextToken()?

- No. They are private methods, and implementation details.

Example

Do we need to write tests for hasMoreTokens() and getNextToken()?

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
- hasMoreTokens()
- getNextToken()

- No. They are private methods, and implementation details.
- But, `ExprEvaluator` is a big class, consisting of lexing, parsing, and evaluation.

Example

How to test hasMoreTokens() and getNextTokens()?

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
- hasMoreTokens()
- getNextToken()

Example

How to test hasMoreTokens() and getNextTokens()?

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
hasMoreTokens()
getNextToken()

Example

How to test hasMoreTokens() and getNextTokens()?
How to test parse()?

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
- parse()
- parseTerm()
- parsePrimary()
hasMoreTokens()
getNextToken()

Example

How to test hasMoreTokens() and getNextTokens()?
How to test parse()?

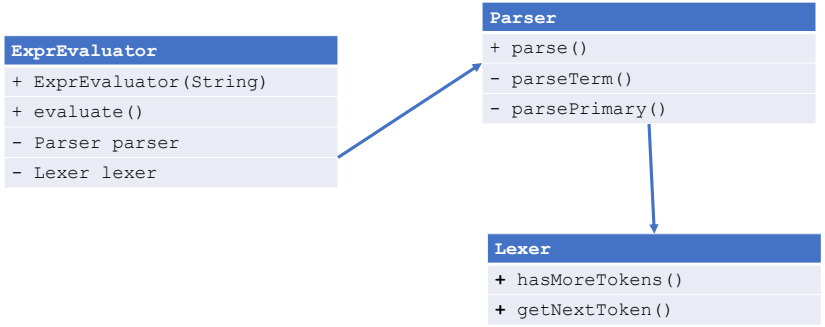
ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
# parse()
- parseTerm()
- parsePrimary()
hasMoreTokens()
getNextToken()

Example

How to test hasMoreTokens() and getNextTokens()?
How to test parse()?
Better Solution?

ExprEvaluator
+ ExprEvaluator(String)
+ evaluate()
# parse()
- parseTerm()
- parsePrimary()
hasMoreTokens()
getNextToken()

Example – Single Responsibility Principle



State Hidden in Method

You find yourself wishing you could access local variables in tests.

```
int f(int p) {
  int t1 = ……;
  int a = p + t1; // I want to test with a
  int t2 = ……;
  int b = a + t2; // I want to test with b
  int t3 = b + ……;
  return t3;
}
```

State Hidden in Method

You find yourself wishing you could access local variables in tests.

```
int f(int p) {
  int t1 = ……;
  int a = p + t1; // I want to test with a
  int t2 = ……;
  int b = a + t2; // I want to test with b
  int t3 = b + ……;
  return t3;
}
```

Methods are too long
Violation of Single Responsibility Principle (SRP)

State Hidden in Method

You find yourself wishing you could access local variables in tests.

```
int f(int p) {
  int t1 = ……;
  int a = p + t1; // I want to test with a
  int t2 = ……;
  int b = a + t2; // I want to test with b
  int t3 = b + ……;
  return t3;
}
```

Methods are too long
Violation of Single Responsibility Principle (SRP)

```
int f(int p) {
  int a = f1(p); // I want to test with a
  int b = f2(a); // I want to test with b
  return f3(b);
}

int f1(int p) {
  int t1 = ……;
  return p + t1;
}

int f2(int a) {
  int t2 = ……;
  return a + t2;
}

int f3(int b) { }
```


Difficult Setup

Instantiating a class involves instantiating 12 others.

Difficult Setup

Instantiating a class involves instantiating 12 others.

Too much coupling.

State-Leaks Across Tests

The state of one test affects another

State-Leaks Across Tests

The state of one test affects another

```
public class A {
  private static int idGenerator = 0;
  public final int id;
  public A() { this.id = idGenerator++;}
}

@RunWith(Junit4.class)
public class TestA {

  @Test
  public void t1() { assertEquals(new A().id, 1); }

  @Test
  public void t2() { assertEquals(new A().id, 1); }


}
```


State-Leaks Across Tests

The state of one test affects another.

Singletons, or other forms of Global Mutable State.


```
public class M {
  public static void main(String[] argv) {
    M obj = new M();
    if (argv.length > 0) obj.m(argv[0], argv.length);
  }

  public void m(String arg, int i) {
    int q = 1;
    A o = null;
    Impossible nothing = new Impossible();
    if (i == 0)
      q = 4;
    q++;
    switch (arg.length()) {
      case 0:
        q /= 2;
        break;
      case 1:
        o = new A();
        new B();
        q = 25;
        break;
      case 2:
        o = new A();
        q = q * 100;
      default:
        o = new B();
        break;
    }
    if (arg.length() > 0) {
      o.m();
    } else {
      System.out.println("zero");
    }
    nothing.happened();
  }

  public static class A {
    public void m() {
      System.out.println("a");
    }
  }

  public static class B extends A {
    public void m() {
      System.out.println("b");
    }
  }

  public static class Impossible {
    public void happened() {
      // "2b||!2b?", whatever the answer nothing happens here
    }
  }
}
```


- What are the problems of testing this program?

```
public class M {
  public static void main(String[] argv) {
    M obj = new M();
    if (argv.length > 0) obj.m(argv[0], argv.length);
  }

  public void m(String arg, int i) {
    int q = 1;
    A o = null;
    Impossible nothing = new Impossible();
    if (i == 0)
      q = 4;
    q++;
    switch (arg.length()) {
      case 0:
        q /= 2;
        break;
      case 1:
        o = new A();
        new B();
        q = 25;
        break;
      case 2:
        o = new A();
        q = q * 100;
      default:
        o = new B();
        break;
    }
    if (arg.length() > 0) {
      o.m();
    } else {
      System.out.println("zero");
    }
    nothing.happened();
  }

  public static class A {
    public void m() {
      System.out.println("a");
    }
  }

  public static class B extends A {
    public void m() {
      System.out.println("b");
    }
  }

  public static class Impossible {
    public void happened() {
      // "2b||!2b?", whatever the answer nothing happens here
    }
  }
}
```


- What are the problems of testing this program?
  - Need to assert System.out.
    - global mutable state. What if two tests need to redirect System.out at the same time?
  - Always not a good idea to assert System.out
    - how to refactor the code for testability?


```
public class M {
  public static void main(String[] argv) {
    M obj = new M();
    if (argv.length > 0) obj.m(argv[0], argv.length);
  }

  public void m(String arg, int i) {
    int q = 1;
    A o = null;
    Impossible nothing = new Impossible();
    if (i == 0)
      q = 4;
    q++;
    switch (arg.length()) {
      case 0:
        q /= 2;
        break;
      case 1:
        o = new A();
        new B();
        q = 25;
        break;
      case 2:
        o = new A();
        q = q * 100;
      default:
        o = new B();
        break;
    }
    if (arg.length() > 0) {
      o.m();
    } else {
      System.out.println("zero");
    }
    nothing.happened();
  }

  public static class A {
    public void m() {
      System.out.println("a");
    }
  }

  public static class B extends A {
    public void m() {
      System.out.println("b");
    }
  }

  public static class Impossible {
    public void happened() {
      // "2b||!2b?", whatever the answer nothing happens here
    }
  }
}
```



```
public class M {
  public static void main(String[] argv) {
    M obj = new M();
    if (argv.length > 0) System.out.println(obj.m(argv[0], argv.length));
  }

  public String m(String arg, int i) {
    int q = 1;
    A o = null;
    Impossible nothing = new Impossible();
    if (i == 0)
      q = 4;
    q++;
    switch (arg.length()) {
      case 0:
        q /= 2;
        break;
      case 1:
        o = new A();
        new B();
        q = 25;
        break;
      case 2:
        o = new A();
        q = q * 100;
      default:
        o = new B();
        break;
    }
    final String result;
    if (arg.length() > 0) {
      result = o.m();
    } else {
      result = "zero";
    }
    nothing.happened();
    return result;
  }

  public static class A {
    public String m() {
      return "a";
    }
  }

  public static class B extends A {
    public String m() {
      return "b";
    }
  }

  public static class Impossible {
    public void happened() {
      // "2b||!2b?", whatever the answer nothing happens here
    }
  }
}
```