

CS 575

Software Testing and Analysis

Özyegin University
Graduate School of Engineering



Hasan Sözer
hasan.sozer@ozyegin.edu.tr

About the Instructor

- 1998 – 2004:
 - B.Sc. & M.Sc.
- 2002 – 2005:
 - Software Engineer
- 2005 – 2011: UNIVERSITY OF TWENTE.
 - Ph.D. & Post-Doc Researcher
- 2011 – present:
 - Faculty Member



<http://faculty.ozyegin.edu.tr/hsozer>

aselsan

Embedded Systems
INSTITUTE

Canon

PHILIPS

ÖZYEGİN
ÜNİVERSİTESİ

 **STM**

EPIAŞ

VESTEL

 **TURKCELL**

Course Objectives

- Explain basic **principles** of software testing
- Summarize basic testing **techniques** and strategies
- Provide a **rationale** for selecting and combining them within a software development process
- Understand **limitations** and **possibilities**



Before we begin...

- Introduction & Context
 - Software Dependability, Reliability
- Course Organization
 - Requirements, goals, expectations
 - Study material
 - Schedule

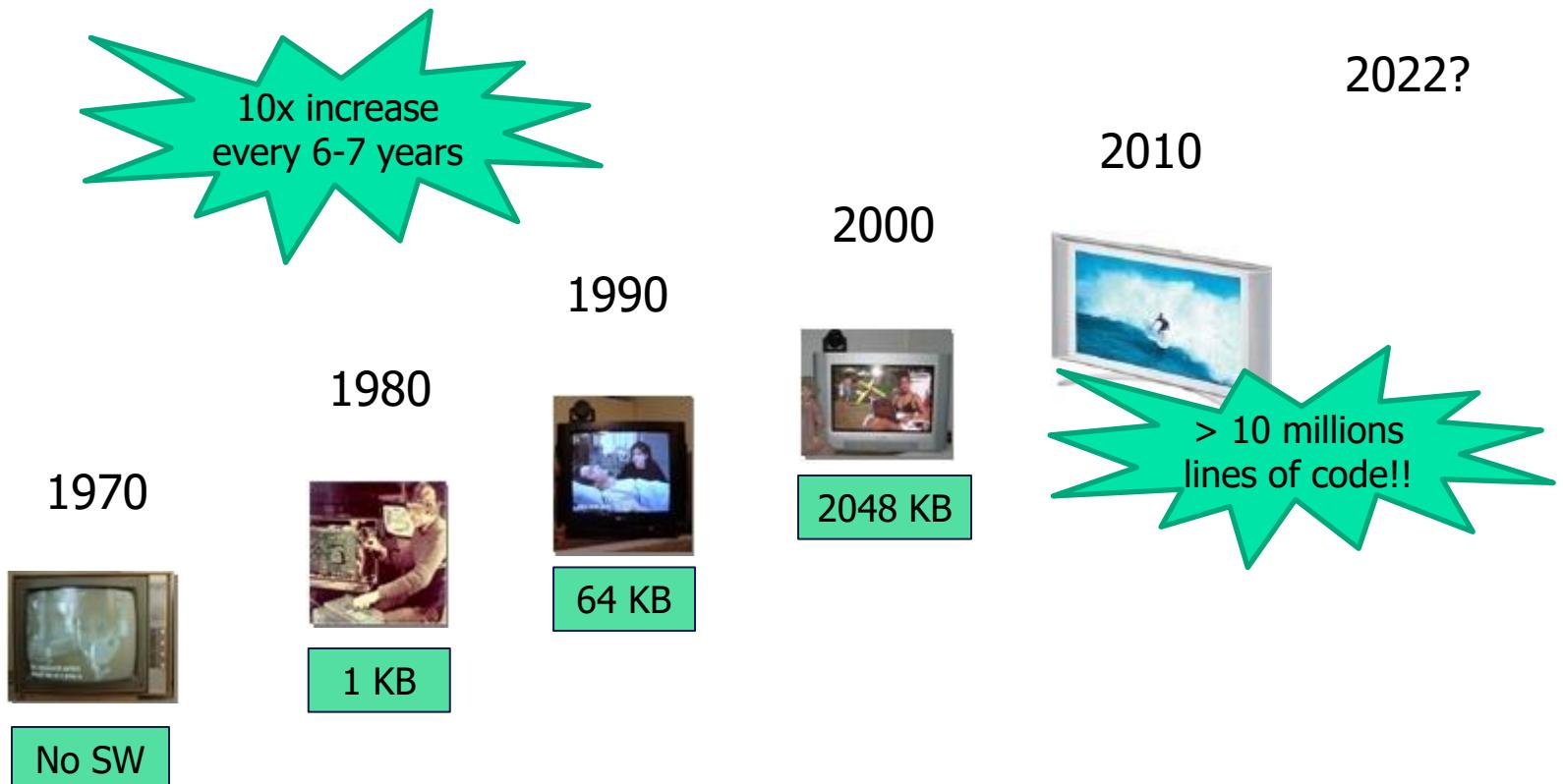


Increasing Software and Complexity in Systems

- Software systems are getting bigger & more complex
 - it becomes harder to ensure their quality
- Yet, we **depend on** software even more than before
 - airplanes, trains, TV sets, ovens, cell phones, ...
- The quality of the software largely defines the behavior and quality of systems we use



Example: Software Size in TVs



Software failure caused \$1.7 trillion in financial losses in 2017

"Software testing company Tricentis found that retail and consumer technology were the areas most affected, while software failures in public service and healthcare were down from the previous year"



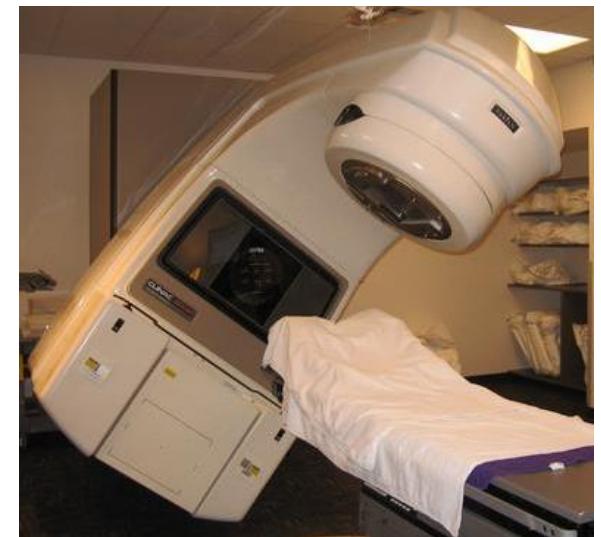
<https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/>

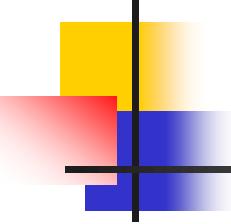
Software Failures as a Threat for Safety

- **Toyota brakes:** dozens dead, thousands of crashes
- **THERAC-25** radiation machine: poor testing resulted in 3 dead



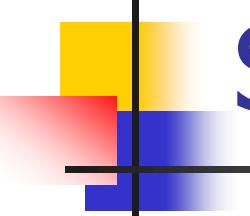
"On February 8th, Toyota announced recalls of tens of thousands of 2010 Prius and Lexus hybrids to address braking problems, this one caused by a software error."
[www.theatlantic.com]





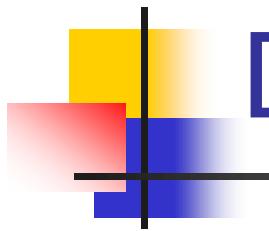
Impact of new technologies

- Advanced development technologies
 - can reduce the frequency of some classes of errors
 - but do **not** eliminate errors
- New development approaches can introduce new kinds of faults, e.g.,
 - deadlock or race conditions for distributed software
 - new problems due to the use of polymorphism, dynamic binding and private state in object-oriented software



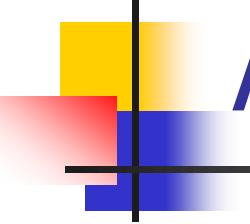
Introducing the Context & Scope of the Course

- Dependability: Ability to deliver service that can justifiably be trusted
 - An integrating concept that encompass 5 quality attributes:
Availability, **Reliability**, Safety, Integrity and Maintainability
- Security is another composite attribute
 - combines Confidentiality, Integrity and Availability



Dependability attributes

- Availability: readiness for correct service
- Reliability: continuity of correct service
- Safety: absence of catastrophic consequences on the user(s) and the environment
- Integrity: absence of improper system alterations
- Maintainability: ability to undergo modifications and repairs



Attribute inter-relationships

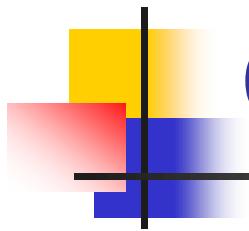
- Safe system operation might depend on the system being available and operating reliably
- A system might be stopped for safety reasons while hindering its availability
- A frequently crashing, unreliable system will also be unavailable
- A system might be very reliable but its availability can be very low due to long-lasting recovery and repair
- Denial of service attacks on a system are intended to make it unavailable.

Reliability

as a software quality attribute

- **Reliability:** The probability that a system will continue to function without **failure** for a specified period in a specified environment
- **Failure:** deviation of the delivered service from compliance with the **specification**



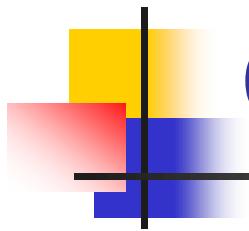


Chain of reliability threats

failure

deviation of
the delivered
service from
compliance with
the specification





Chain of reliability threats

propagates to
system error → **failure**

system state
that is liable to
cause a failure



deviation of
the delivered
service from
compliance with
the specification



Chain of reliability threats



the cause of a
system error

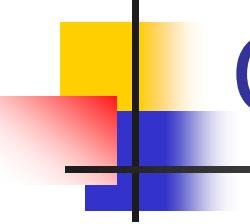


system state
that is liable to
cause a failure



deviation of
the delivered
service from
compliance with
the specification





Chain of reliability threats



mistake



the cause of a system error

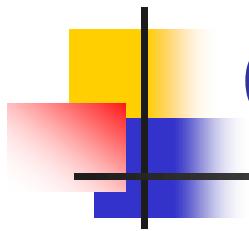


system state that is liable to cause a failure



deviation of the delivered service from compliance with the specification





Chain of reliability threats



Examples:

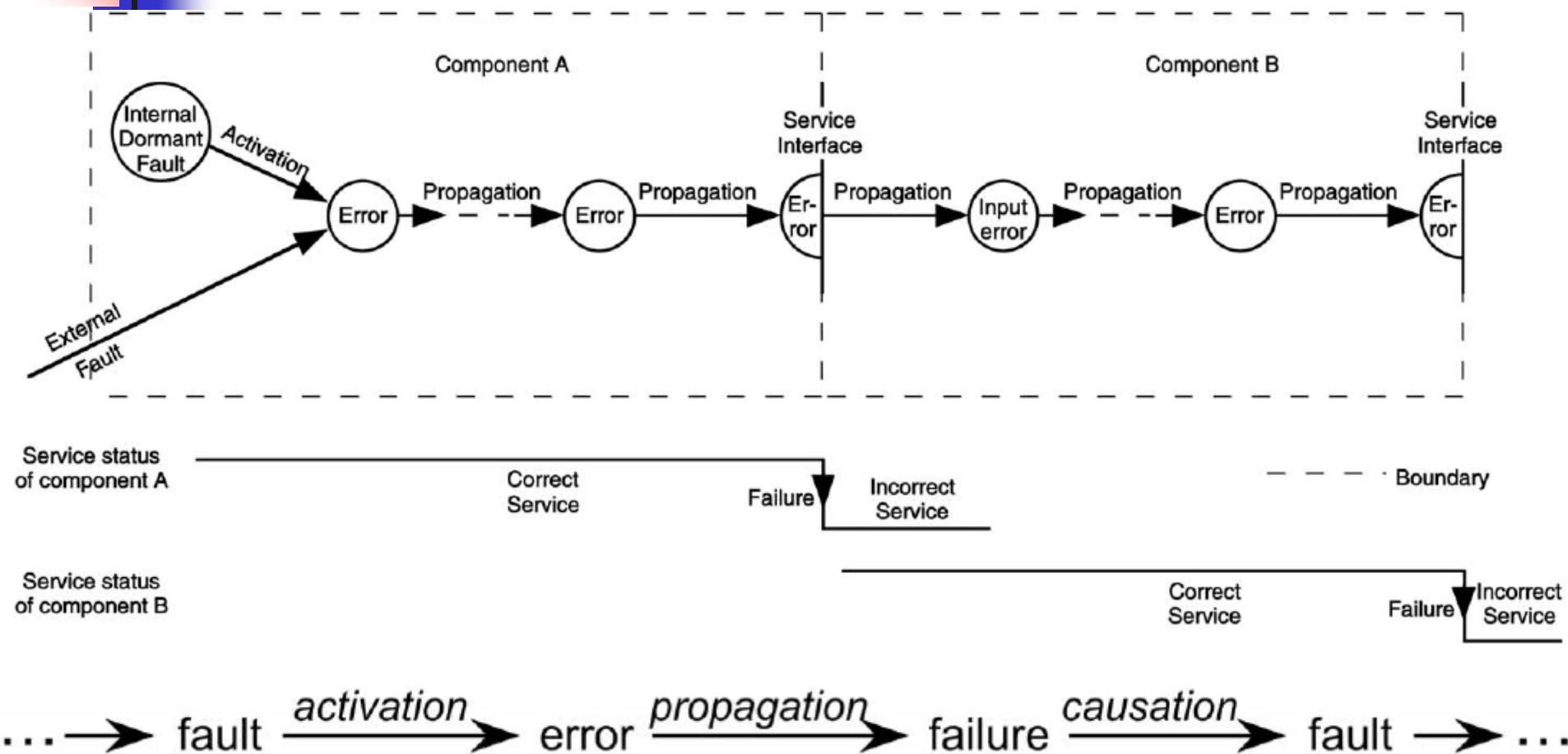
**Bug: array index
uninitialized**

**Index out-of-
bounds**

**Segmentation fault,
system crash**

Chain of reliability threats

[Avizienis 2004]



Fault Classification

Development Faults

Physical Faults

Interaction Faults

[Avizienis 2004]

Development Faults

Operational Faults

Internal Faults

External Faults

Natural Faults

Human-Made Faults

Hardware Faults

Software Faults

Non-Malicious Faults

Malicious Faults

Non-Deliberate Faults

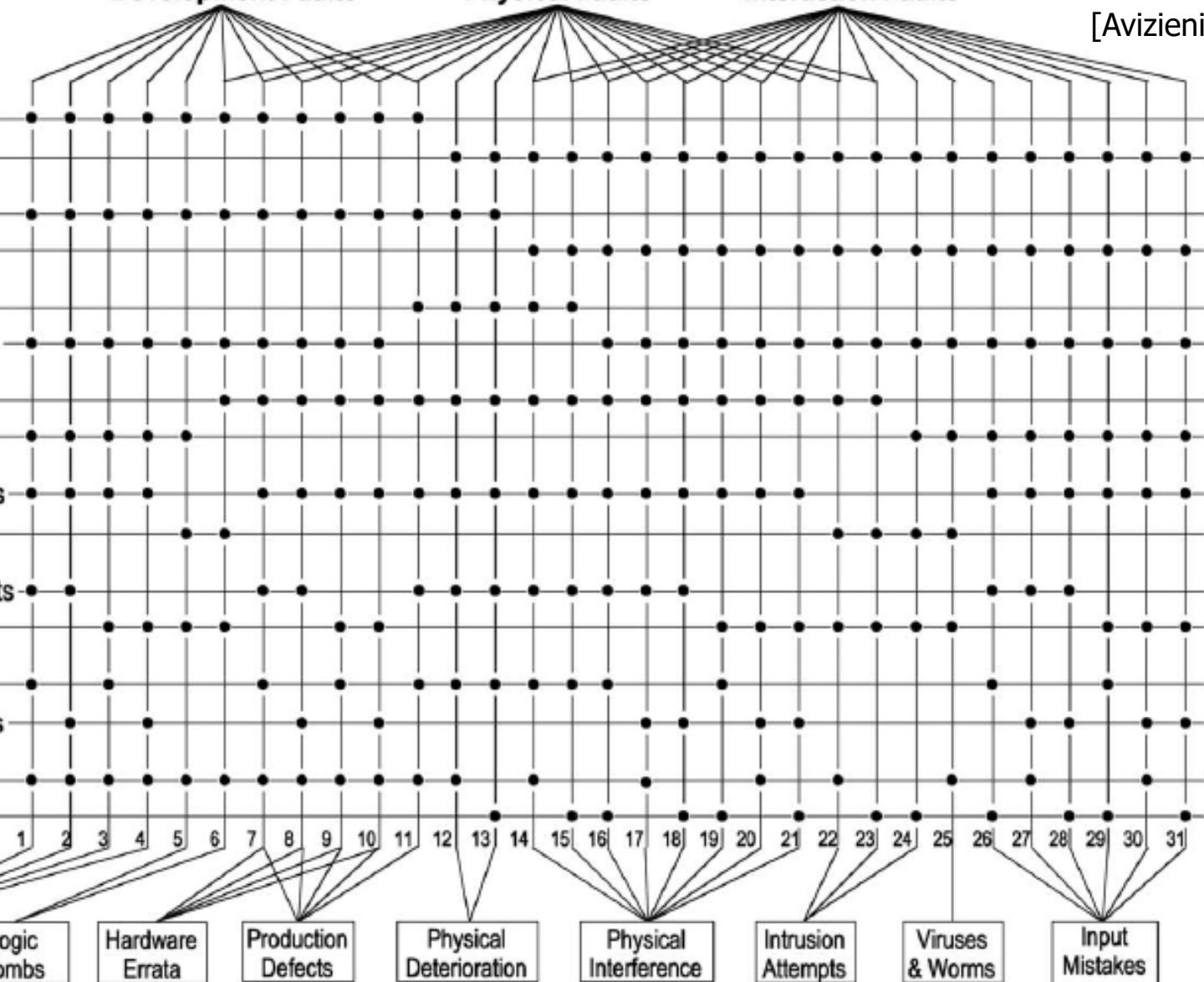
Deliberate Faults

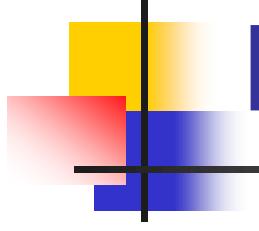
Accidental Faults

Incompetence Faults

Permanent Faults

Transient Faults





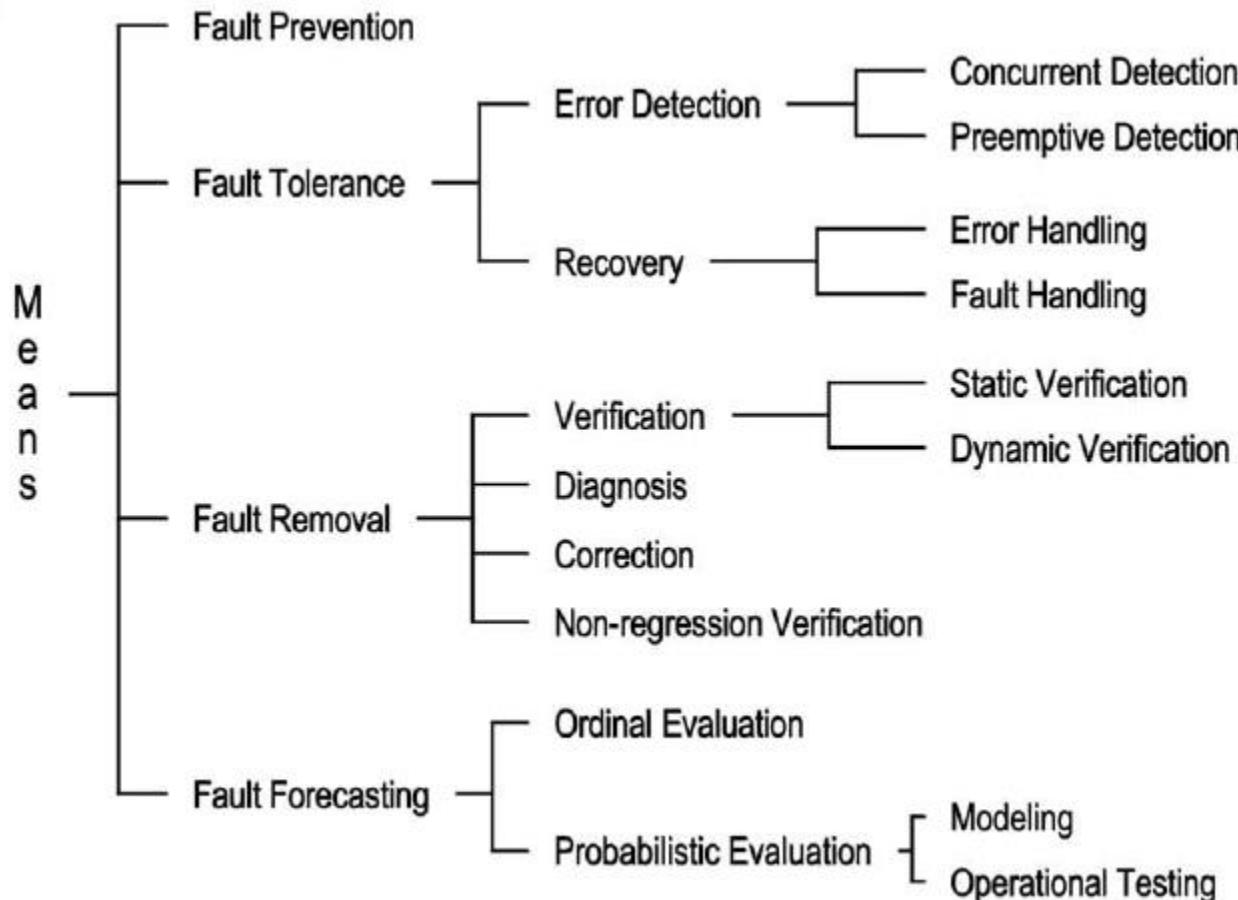
How to prevent failures?



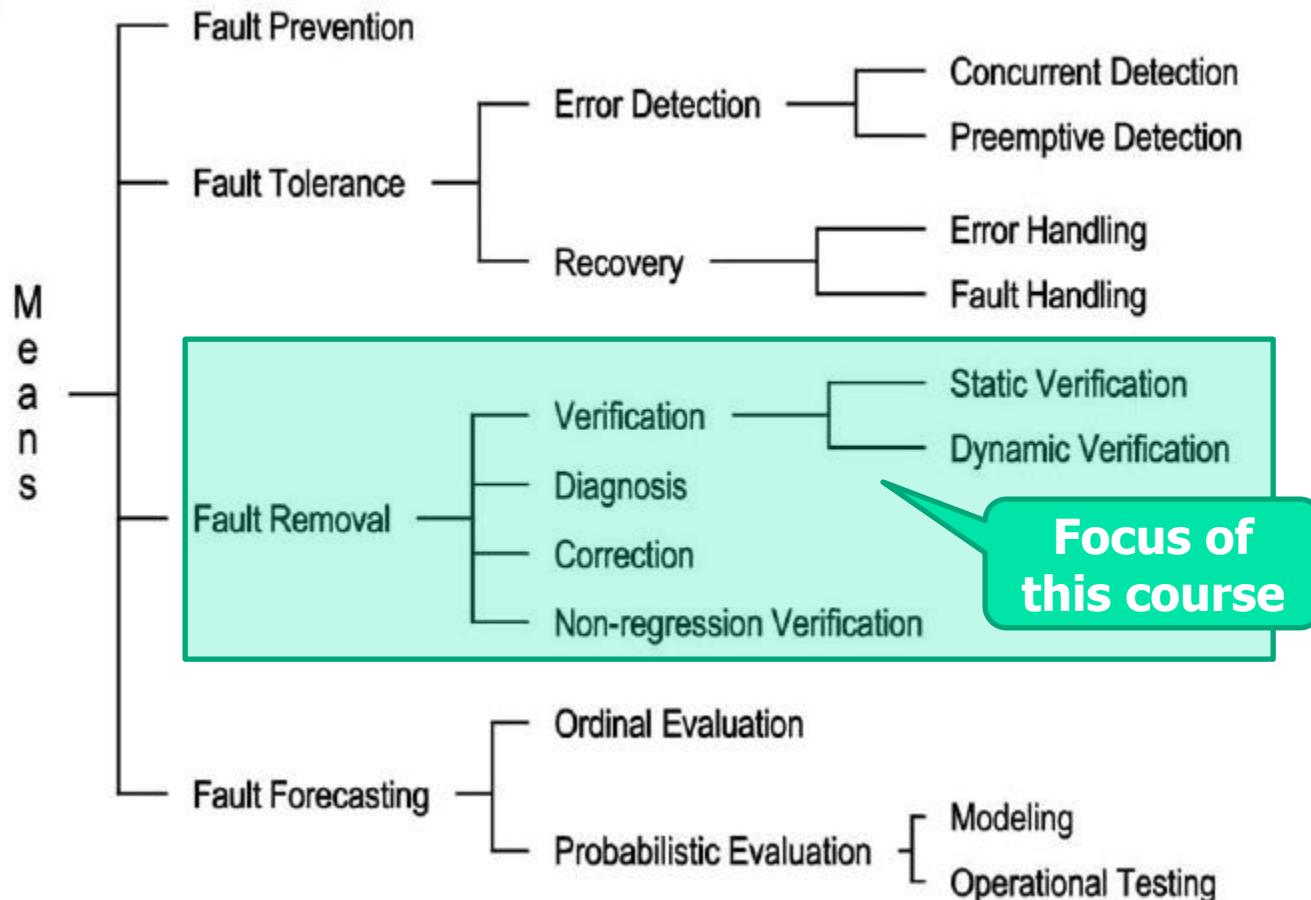
- The chain of threats must be broken
 - Fault prevention (avoidance)
 - Fault removal
 - Fault tolerance

Dependability means

[Avizienis 2004]

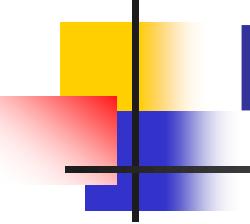


Dependability means



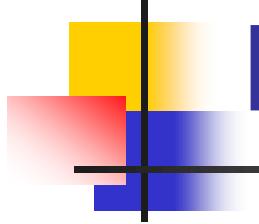
**Focus of
this course**

Avizienis 2004



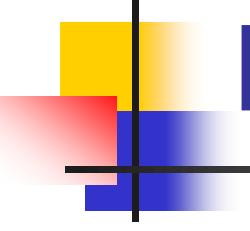
Fault Prevention

- Development techniques that either minimize the possibility of mistakes or trap mistakes before they result in the introduction of faults
- e.g., process improvement approaches to follow a (more) rigorous software development



Fault Removal

- Verification and validation techniques that increase the probability of detecting and correcting faults before the system becomes operational
- i.e., **Software Testing and Analysis**



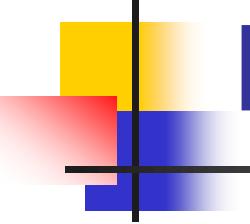
Fault Tolerance

- It is usually not feasible to prevent and/or remove all the faults.

*"There are two ways to
write error-free programs;
only the third one works."*

Alan Perlis

- Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its components.

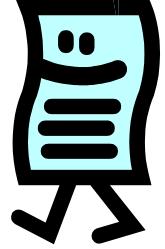


Fault Tolerance (cont'd)

- Fault-tolerant design

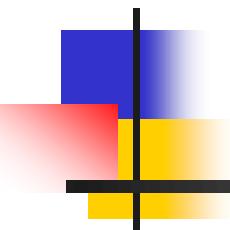
- refers to a method for designing a system so that it will continue to operate,
- possibly with a reduced functionality/quality,
- rather than failing completely, when some part of the system fails.





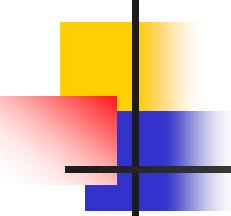
Further reading material..

- A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr,
**Basic Concepts and Taxonomy of Dependable
and Secure Computing**, IEEE Transactions on
Dependable and Secure Computing, Vol. 1(1), 2004.

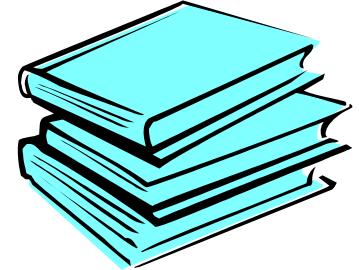


Course Organization





Study Material



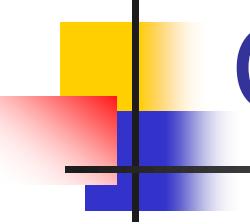
- Course Book:
 - P. Amman and J. Offut: Introduction to Software Testing, Cambridge University Press, 2008.
- Recommended supplementary book:
 - M. Pezze and M. Young: Software Testing and Analysis: Process, Principles, and Techniques, Wiley, 2008.
- Course slides, assignments on LMS
 - <http://lms.ozyegin.edu.tr>



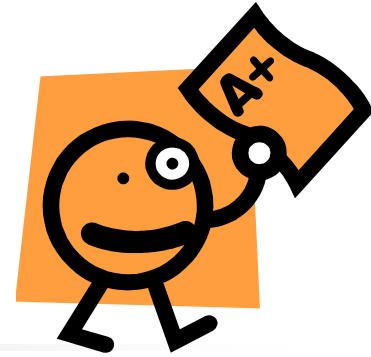
The List of Topics



Week	Topic
1.	Basic Terminology and Testing Principles
2.	Testing Process and Test-driven Development
3.	Testing Techniques
4.	Control Flow Analysis
5.	Test Coverage Measures
6.	Graph Testing
7.	Software Analysis Techniques and Tools
8.	Fault Localization and Cost Estimation
9.	Concolic Testing
10.	Test Effectiveness Analysis and Test Case Prioritization
11.	Testing Tools and Automation
12.	Testing Web Applications
13.	Testing Graphical User Interfaces
14.	Qualitative Analysis and Diagnosis Techniques



Grading



- 50% Project
 - 5% Proposal
 - 10% Progress Presentation
 - 15% Final Presentation
 - 20% Project Report
- 50% Midterm Exam (x2)

Overall Schedule



October		November						December		
21	28	4	11	18	25	2	9	16	23	30
22	29	5	12	19	26	3	10	17	24	31
Online Lecture										
Exam										
Project Presentation										
Report Submission										

CS 575

Software Testing and Analysis



Ozyegin University
Graduate School of Engineering



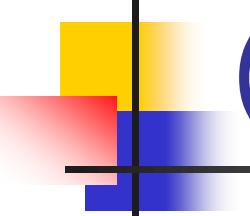
Potential Project Topics and Examples

Hasan Sözer
hasan.sozer@ozyegin.edu.tr

Review of Potential Project Topics

- Methods, tools, techniques related to Software Testing and Analysis
- Also a short **overview** of the subject material
- Towards determining a project topic and scope
 - ... and preparing a **project proposal**



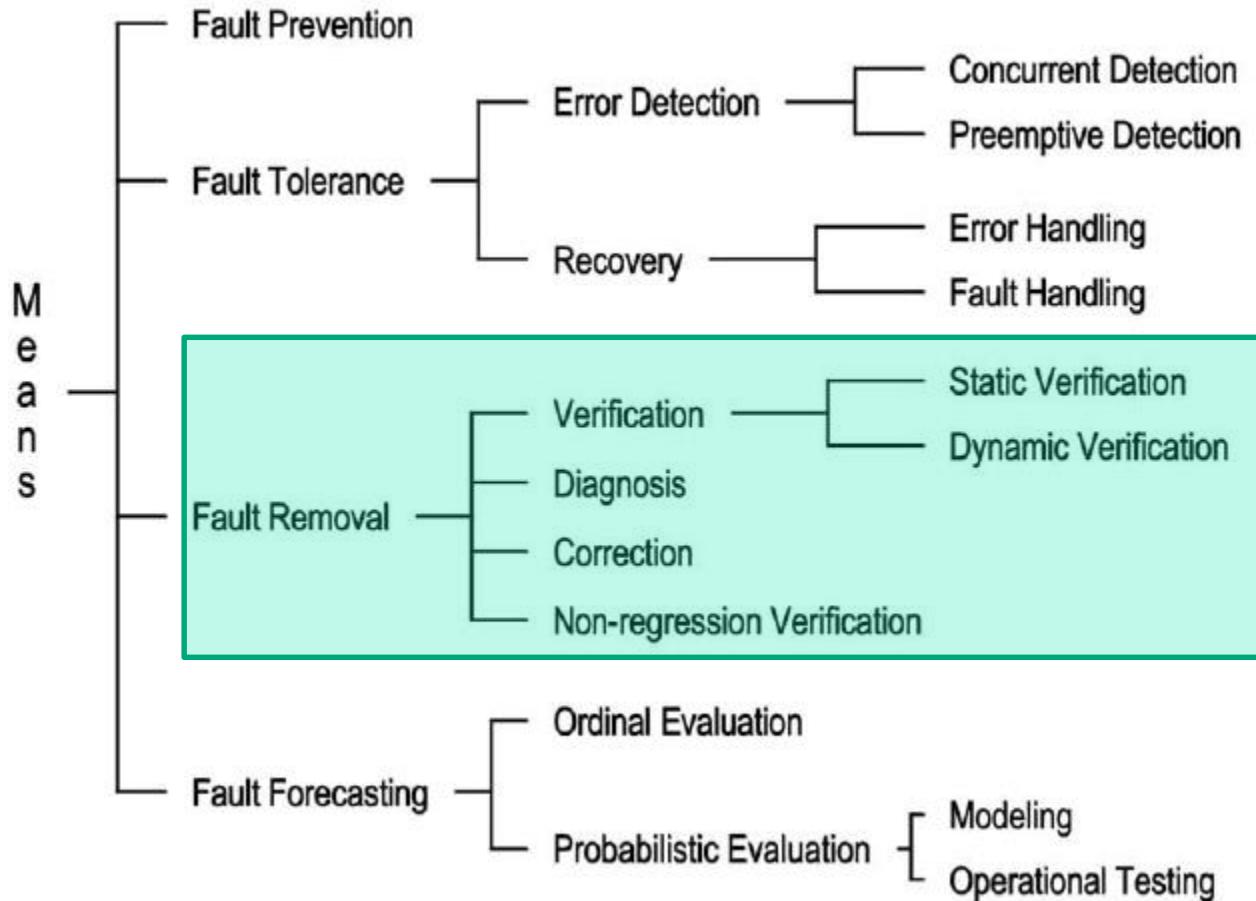


Proposal Contents

(Template available on LMS)

- Group members
- Application domain
 - embedded, Web, mobile, stand-alone application, etc.
- The problem being addressed
 - measuring reliability, detecting errors, diagnosing faults, etc.
- The proposed solution approach
 - type of methods, techniques, tools considered
- Deliverables
 - a framework, integrated tool-set, evaluation report, etc.

Scope

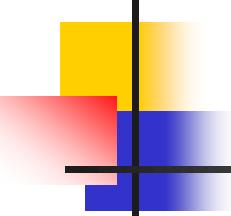


[Avizienis 2004]



Outline

- Static Code Analysis
- Model-Based Testing
- Combinatorial Testing
- Spectrum-based Fault Localization
- Mutation Testing
- Test Automation
 - Mobile Applications
 - Web Applications
- Test Case Selection / Prioritization



New Ideas Welcome!



- A new application domain
- Specific types of faults, errors, failures
- Different concerns
 - Maintenance of test cases, focus on user-perceived failures, etc.
- Our discussion is just to inspire you
- Possible topics are not limited to the discussed examples



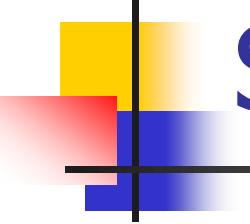
Static Code Analysis

- Analyzing source code without executing it
- Finding potential faults

- Bug Patterns
- Programming Rules
- Scalable but subject to false positives

- Extending tools with custom rules?
- Automatically filtering out false positives?





Static Code Analysis Tools

- Findbugs
- PMD
- Klocwork (commercial)
- CppCheck
- Frama-C
- Clang



Analysis for Program Slicing

- Focus on a variable that causes the failure
- Slice the program to filter out the irrelevant parts
- Makes it easier to debug the program

```
Pass = 0 ;
Fail = 0 ;
Count = 0 ;
while (!eof()) {
TotalMarks=0;
scanf("%d",Marks);
if (Marks >= 40)
Pass = Pass + 1;
if (Marks < 40)
Fail = Fail + 1;
Count = Count + 1;
TotalMarks = TotalMarks+Marks ;
}
printf("Out of %d, %d passed and %d failed\n",Count,Pass,Fail) ;
average = TotalMarks/Count;
/* This is the point of interest */
printf("The average was %d\n",average) ;
PassRate = Pass/Count*100 ;
printf("This is a pass rate of %d\n",PassRate) ;
```



```
while (!eof()) {
TotalMarks=0;
scanf("%d",Marks);
Count = Count + 1;
TotalMarks = TotalMarks+Marks;
}
average = TotalMarks/Count;
printf("The average was %d\n",average) ;
```

Example by Mark Harman

Backward vs. Forward Slicing

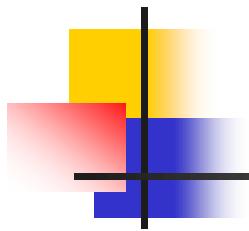
- Forward Slicing: include lines that are affected by the variable in the rest of the program
- Makes it easier to maintain the program

```
x = 1; /* considering changing this line */
y = 3;
p = x + y ;
s = y -2 ;
if(p==0)
x++ ;
```



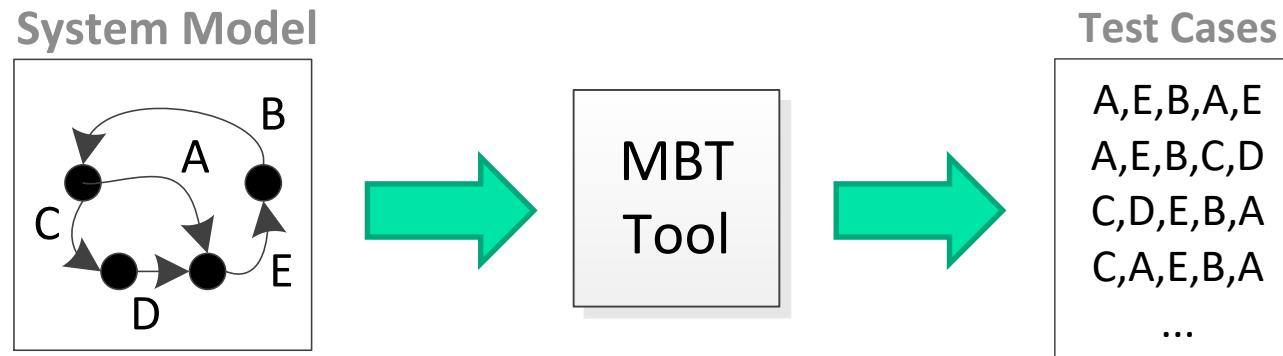
```
/* Change to first line will affect */
p = x + y ;
if(p==0)
x++ ;
```

Example by Mark Harman



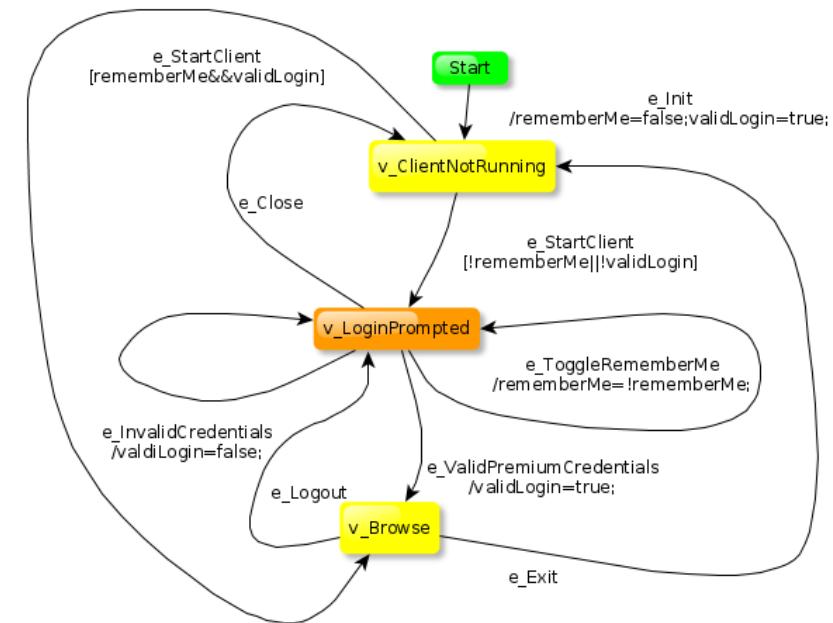
Model-Based Testing

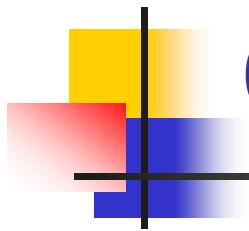
- Automatically generating test cases based on a model of the system



Model-Based Testing Tools

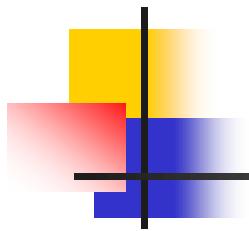
- GraphWalker: Generates Junit test cases
- MaTeLo (commercial)
- Learning the usage model
- Reflecting usage profile
- Analyze coverage
- http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html
- <http://robertvbinder.com/open-source-tools-for-model-based-testing/>





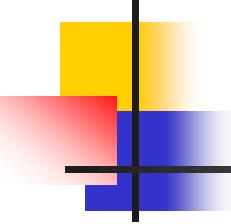
Combinatorial Testing

- Exploring different combinations of parameters and configuration parameters
- Systematically generate combinations to be tested
 - e.g., IE on Vista, IE on XP, Firefox on Vista, ...
- Rationale: Test cases should be varied and include possible “corner cases”



Pairwise testing

- Generate combinations that efficiently **cover all pairs (triples,...) of classes**
- Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,...) reduces the number of test cases, but reveals most faults



Example for Pairwise testing

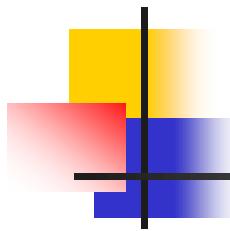
based on the slides of M. Pezze and M. Young

- 432 (3x4x3x4x3) test cases if we consider all combinations

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

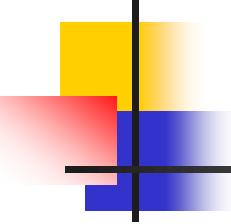
Pairwise combinations: 17 test cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held



Combinatorial Testing Tools

- <http://www.pairwise.org/tools.asp>
- Tcase
- Pict
- ...



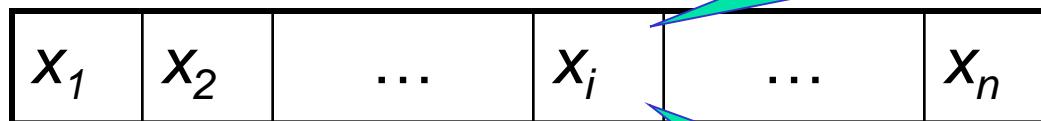
Spectrum-based Fault Localization

- Helping developers for debugging
- Correlating execution traces with test results
 - Which program locations are executed?
 - Which components are involved?
 - Which branches are taken?
 - ...
- Example slides follow for so called **block-hit spectra**
 - Adopted slides of Rui Abreu, Peter Zoeteweij and Arjan van Gemund

Block / function hit spectra

Function hit spectrum

1: function i called
0: function i not called



Block hit spectrum

1: block i executed
0: block i not executed

Block:

- C statement (compound stmt)
- cases of a switch statement

Collected Data

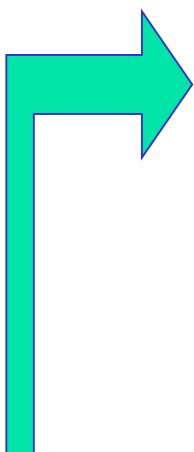
n blocks

m cases

x_{11}	x_{12}	\dots	x_{1n}
x_{21}	x_{22}	\dots	x_{2n}
\dots	\dots	\dots	\dots
x_{m1}	x_{m2}	\dots	x_{mn}

e_1
e_2
\dots
e_m

Collected Data



x_{11}	x_{12}	...	x_{1n}
x_{21}	x_{22}	...	x_{2n}
...
x_{m1}	x_{m2}	...	x_{mn}

e_1
e_2
...
e_m

Row i : the blocks that are executed in case i

Collected Data

x_{11}	x_{12}	...	x_{1n}
x_{21}	x_{22}	...	x_{2n}
...
x_{m1}	x_{m2}	...	x_{mn}

e_1
 e_2
...
 e_m

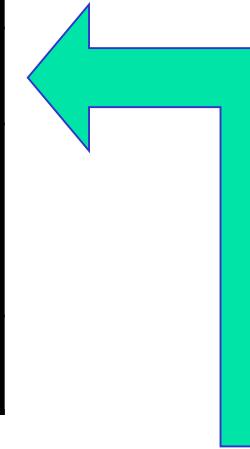


Column j : the test cases in which block j was executed

Collected Data

x_{11}	x_{12}	...	x_{1n}
x_{21}	x_{22}	...	x_{2n}
...
x_{m1}	x_{m2}	...	x_{mn}

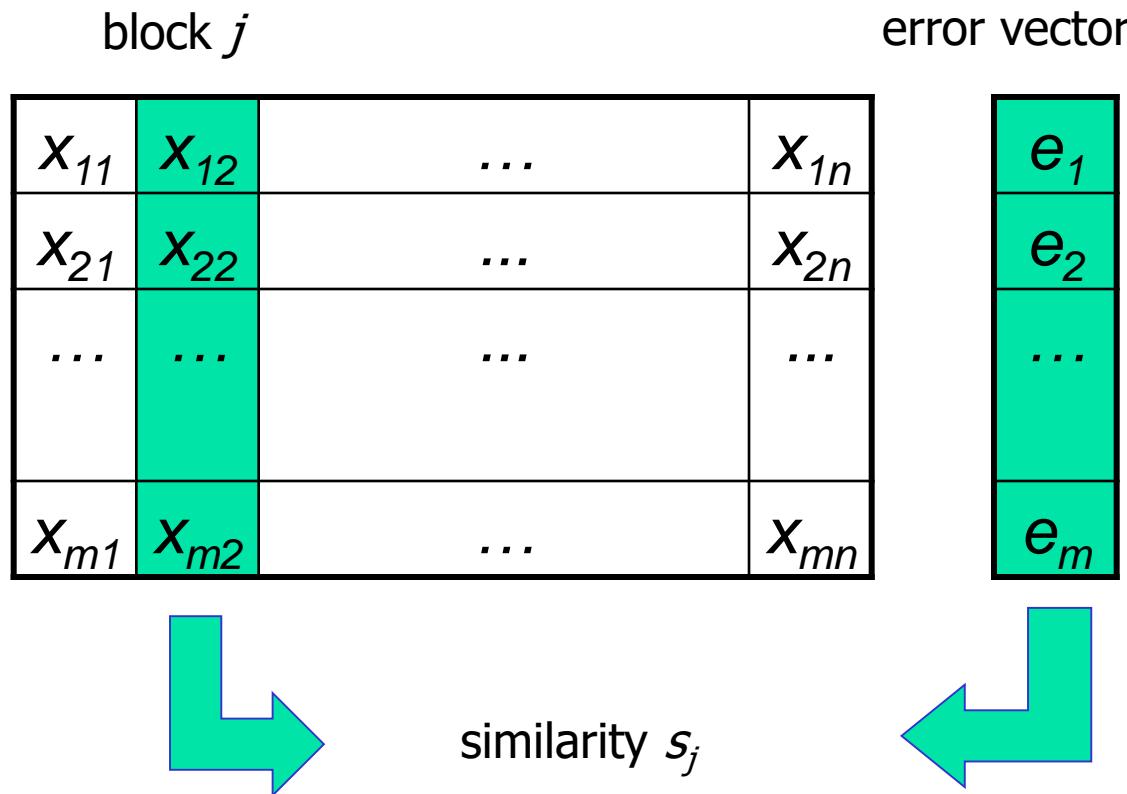
e_1
e_2
...
e_m



$e_i=1$: error in the i -th test

$e_i=0$: no error in the i -th test

Calculation of Similarity



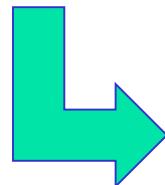
Jaccard Similarity Coefficient

block j

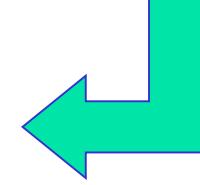
1
0
1
0
1

error vector

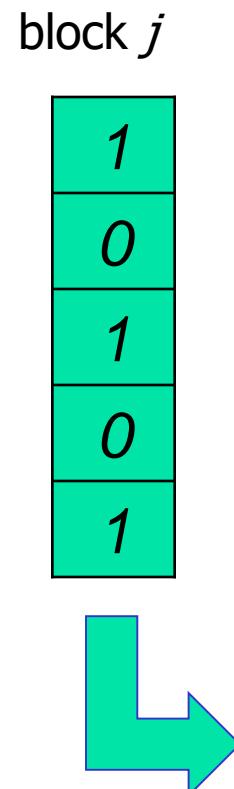
0
1
1
0
1



$$s_j = \frac{a_{11}}{a_{11} + a_{10} + a_{01}}$$



Jaccard Similarity Coefficient



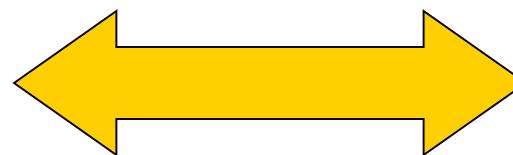
error vector

$$s_j = \frac{a_{11}}{a_{11} + a_{10} + a_{01}}$$

Jaccard Similarity Coefficient

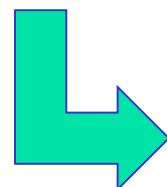
block j

1
0
1
0
1

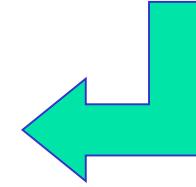


error vector

0
1
1
0
1



$$s_j = \frac{2}{2 + \cancel{a_{10}} + \cancel{a_{01}}}$$



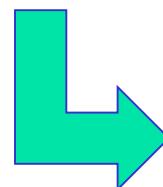
Jaccard Similarity Coefficient

block j

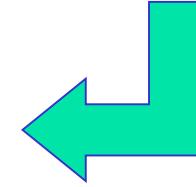
1
0
1
0
1

error vector

0
1
1
0
1



$$s_j = \frac{2}{2 + 1 + a_{01}}$$



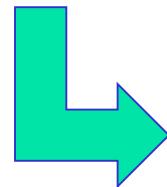
Jaccard Similarity Coefficient

block j

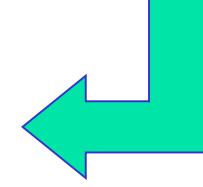
1
0
1
0
1

error vector

0
1
1
0
1



$$s_j = \frac{2}{2 + 1 + 1}$$



Similarity for Each Block

m cases

<i>n</i> blocks			
		error vector	
X_{11}	X_{12}	...	X_{1n}
X_{21}	X_{22}	...	X_{2n}
...
X_{m1}	X_{m2}	...	X_{mn}
$s_1 \quad s_2 \quad \dots \quad s_n$			

e_1
e_2
...
e_m

The block with the highest s_i most likely contains the fault.

Example

m cases

n blocks

error vector

0	1	1	1	1	0	0
0	0	0	1	0	1	1
1	1	1	1	0	0	0
0	0	0	0	0	0	0
1	1	0	1	1	0	1

0
1
1
0
1

Example

m cases

<i>n</i> blocks						
0	1	1	1	1	0	0
0	0	0	1	0	1	1
1	1	1	1	0	0	0
0	0	0	0	0	0	0
1	1	0	1	1	0	1

$\frac{2}{3}$ $\frac{1}{2}$ $\frac{1}{4}$ $\frac{3}{4}$ $\frac{1}{4}$ $\frac{1}{3}$ $\frac{2}{3}$

error vector

0
1
1
0
1

Example

m cases

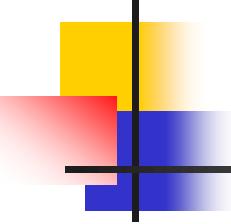
n blocks

error vector

0	1	1	1	1	0	0
0	0	0	1	0	1	1
1	1	1	1	0	0	0
0	0	0	0	0	0	0
1	1	0	1	1	0	1

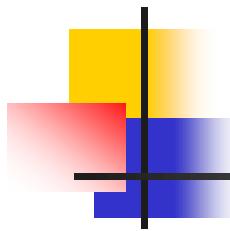
$\frac{2}{3}$ $\frac{1}{2}$ $\frac{1}{4}$ $\frac{3}{4}$ $\frac{1}{4}$ $\frac{1}{3}$ $\frac{2}{3}$

0
1
1
0
1



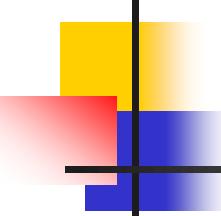
Fault Localization Tools

- Zoltar
- Gzoltar: Eclipse plug-in for Java Applications
 - Paper: W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization," in *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740, Aug. 1 2016.
- Application to new case studies?
- New type of applications?
- A comparison of tools? Similarity metrics?



Mutation Testing

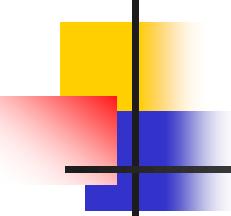
- Testing your tests by injecting faults to your program
- Assume, we have a program P
- Makes 100 different copies of P
- A bug is injected to each copy
- Run all the tests on each of the 100 copies with bugs
- Let's say, all the tests passed for 80 of them
- What would you think about your tests then?



Mutants

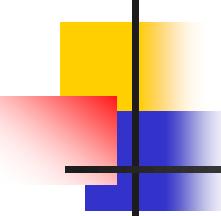


- A **mutant** is a copy of a program with a mutation
- A **mutation** is a syntactic change (a seeded bug)
 - e.g., change $(i < 0)$ to $(i <= 0)$
- A mutant is **killed** if it fails on at least one test case
- If many mutants are killed, infer that the test suite is also effective at finding real bugs



Mutation Testing Assumptions

- Competent programmer hypothesis:
 - Programs are nearly correct
 - Real faults are small variations from the correct program
 - => Mutants are reasonable models of real buggy programs
- Coupling effect hypothesis:
 - Tests that find simple faults also find more complex faults
 - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too

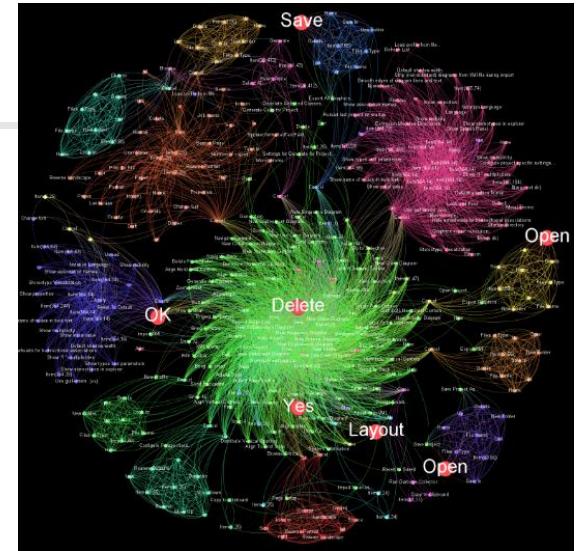
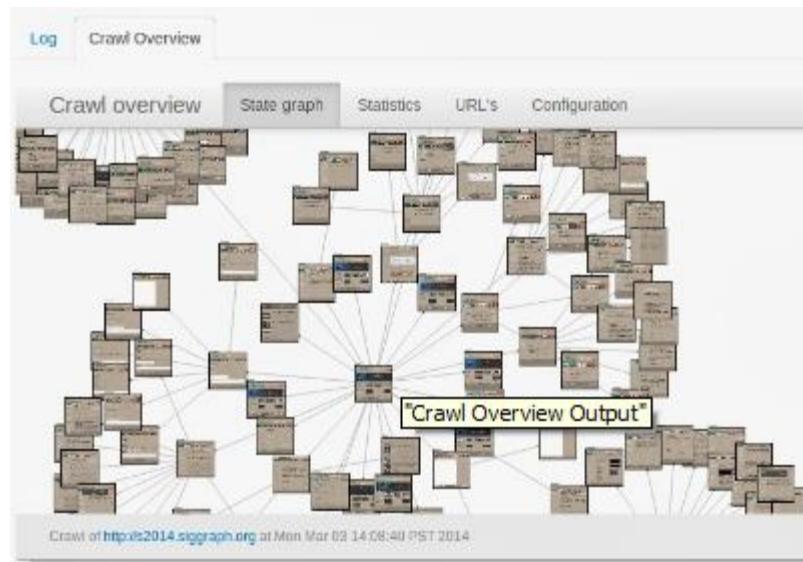


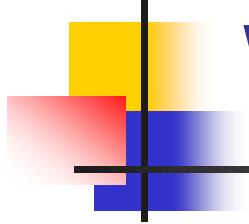
Mutation Testing Tools

- MuClipse: open-source tool for Java
 - Judy
 - PIT
-
- Paper: Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (September 2011), 649-678
-
- Application to new case studies?
 - A comparison of tools?
 - A new mutation testing tool with new type of operators?

Test Automation

- Exploring user interfaces
- Automated Execution of test cases
- Desktop GUI
- Web
- Mobile



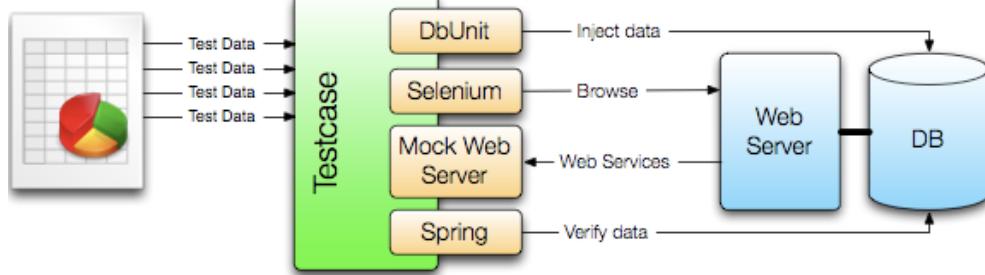


Test Automation Tools for Web and Mobile Applications

- <https://www.selenium.dev/>
- <https://appium.io/>
- <https://developer.android.com/studio/test/espresso-test-recorder.html>
- <http://developer.android.com/tools/help/monkey.html>

Data Driven Acceptance Tests

- DDSteps (<http://ddsteps.sourceforge.net/>)



- FitNesse



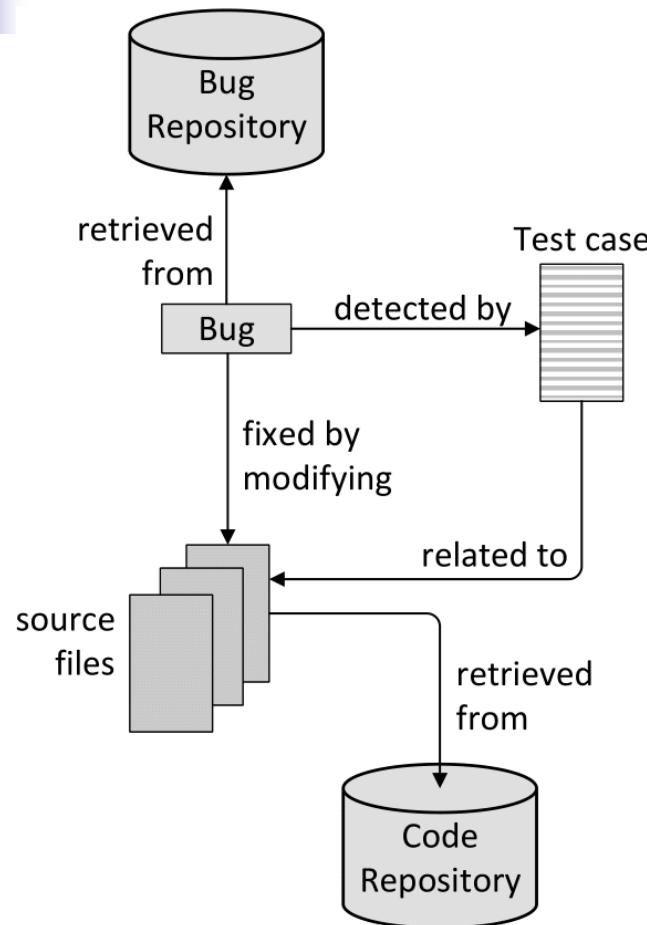
fit.ActionFixture		
start	eg.Page	
enter	location	http://google.com
check	title	Google
enter	link	Jobs
check	title	About Google
enter	link	Press
enter	link	Review
check	title	Google Press Room

Test Case Selection and Prioritization

- Too many test cases, limited resources
- Elimination of redundant test cases
- Early detection of faults by ordering test cases

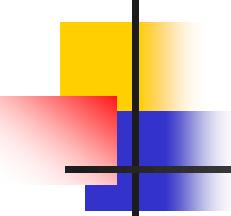
Tutku Çingil and Hasan Sözer. 2022. **Black-box Test Case Selection by Relating Code Changes with Previously Fixed Defects**. In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering. Association for Computing Machinery, New York, NY, USA, 30–39.

Black-box Test Case Selection



- A file can be modified multiple times to fix various bugs detected by a test case
- The more such modifications exist, the stronger relation becomes
- Selection is made according to S , threshold for the relation strength

$$s(t_j) = \begin{cases} 1, & \text{if } \exists f \in F, \text{ s.t. } \frac{e(f,t_j)}{\sum_{i=1}^n e(f_i,t_j)} \times 100 \geq S \\ 0, & \text{otherwise} \end{cases}$$



Case Study

Project ID	Software Version	SVN Revision
P1	V1.37.0.0	319.405 → 320.082
P1	V1.48.0.0	323.377 → 323.742
P1	V1.71.0.0	334.026 → 334.618
P2	V1.44.0.0	328.479 → 328.822
P2	V1.47.0.0	329.316 → 329.819
P2	V1.53.0.0	332.343 → 332.631
P3	V0.51.19.0	311.881 → 312.796
P3	V0.51.28.0	324.786 → 325.791

- Applied to 3 Smart TV projects
- 38 test cases, 437 minutes
- 438 source code files
- 482 defects

A Sample Bug Report

No Audio after Netflix Exit

[Edit](#) [Comment](#) [Assign](#) [More](#) [Close Issue](#) [Reopen Issue](#) [Export](#)

Details

Type:	Bug	Status:	RESOLVED
Priority:	Showstopper	Resolution:	Fixed
Affects Version/s:	v0.9.0	Fix Version/s:	[REDACTED]
Component/s:	phase1	Security Level:	[REDACTED]
Labels:	None		

[Bug](#) [Environment](#) [Misc](#) [Resolution](#)

Frequency: always-2/2
Approval Type: P1-MP
Project Category: Software-TV
Reopen count: 0
Redefine count: 0
Reassign count: 3
Effect: Spec & Performance Unusability
Scenario: Easy
Recovery: Power On Off
Copy to Clipboard:

Maturity Level: 43
MTK CQ: DTV03456720

Description

Attachments

[teraterm.log](#) 36 kB 22/Mar/22 9:30 AM

Sub-Tasks

There are no Sub-Tasks for this issue.

People

Assignee: [REDACTED] Assign to me
Reporter: [REDACTED]
Reporter Group: RD, RD-DVT, RD-DVT-Test
Assignee Group: [REDACTED]
Resolver: [REDACTED]
Votes: 0 Vote for this issue
Watchers: 3 Start watching this issue

Dates

Created: 22/Mar/22 9:31 AM
Updated: 6 days ago
Resolved: 6 days ago
Assignee last set: 6 days ago

Collaborators

Drag and Drop

Drop files here to attach them
or
[Select files](#)

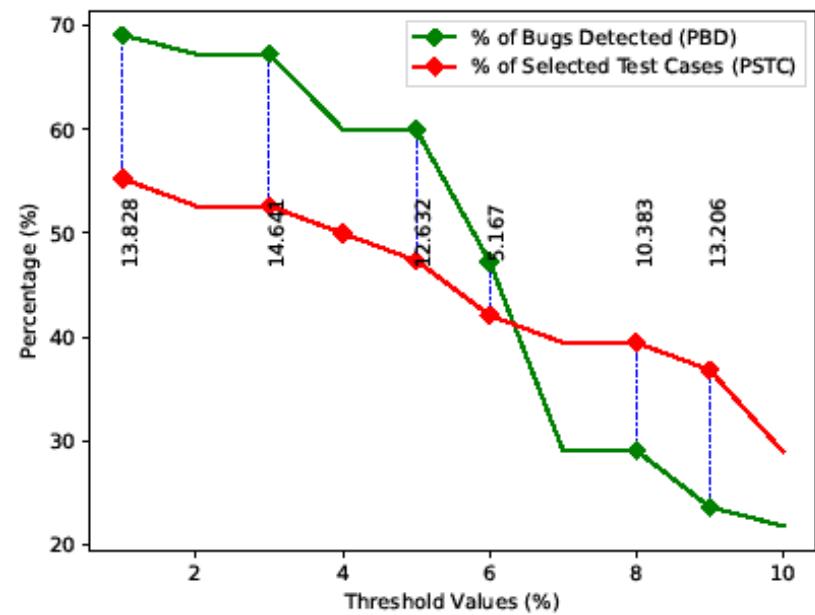
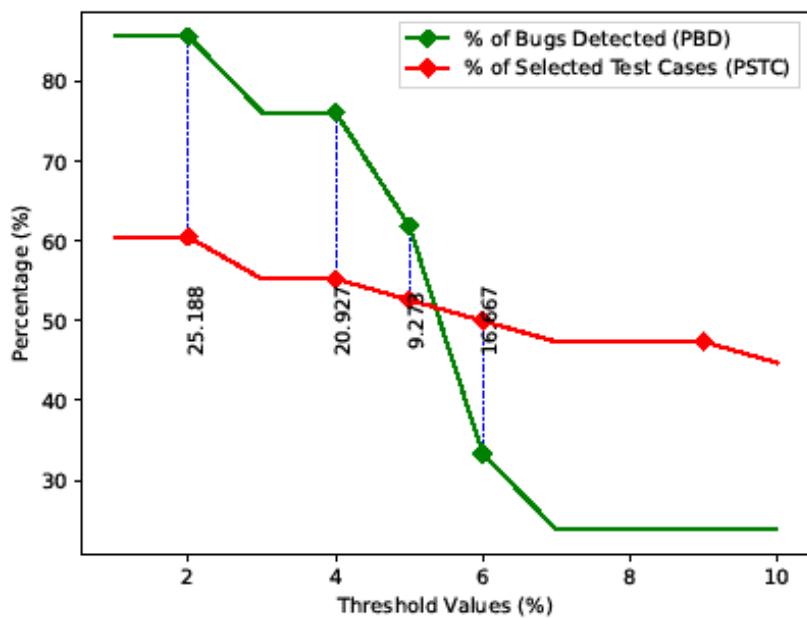
Agile

[View on Board](#)

A Sample Test Case

TS ID	Stimulation	Verification
1	Press the Netflix button on the remote. Search for Pokemon : Indigo Legend from Netflix. Open the first episode.	There should be no problem on video playback, A/V should be fine. The video should appear in 4:3 format.
2	Change the volume with V+/-, mute/unmute while any video is playing.	V+/-, mute/unmute changes should not be a problem. Volume OSD should appear on the screen.
3	Exit Netflix by pressing the Exit button.	Exit Netflix by pressing the Exit button.
4	Log in to Netflix via Home Launcher. Go to Netflix search field and search for ILIZA. While the video is playing, press the subtitle button on the remote to select the subtitle language in Turkish and check the subtitle.	There should be no problem when subtitle is displayed.
5	While the video is playing, change the audio information from the netflix options menu.	There should be no problem in audio changes, audio distortion, cutting etc. should not be a problem.
6	Exit Netflix by pressing the Exit button.	There should be no problem exiting Netflix, A/V should be ok when returning to the channel.
7	Enter Netflix again. Go to Netflix search field and search for IRON FIST. While the video is playing, check Pause/Play/FF/FR.	There should be no problem with Pause/Play/FF/FR operations. A/V should be OK.
8	Exit Netflix by pressing the Exit button.	There should be no problem exiting Netflix, A/V should be ok when returning to the channel.
9	Enter Netflix by pressing the netflix button on the remote. Go to the Netflix search field and search for IRON FIST. Select the series with OK button and open the first episode.	The desired video can be selected with OK button. A/V should be OK, no problem should be observed in video playback.
10	While the video is playing, press the back button on the remote.	Video playback should stop, menu options should appear.
11	Select the Resume option.	The video should continue playing where it left off.
12	Press the back button again and select Play from beginning.	The video should start from the beginning.
13	Log into Netflix via Home Launcher. Search and play for 4K (Moving Art: Flowers / Oceans) video.	A/V should be OK (if the project does not support 4K, it will appear in HD)
14	Press the info button while the video is playing.	Video duration, bitrate, resolution, etc. should appear on the screen. The video should appear in 4K resolution (if the project does not support 4K, it will appear in HD). Audio/text information should appear on the screen.

Some Results





CS 575

Software Testing and Analysis

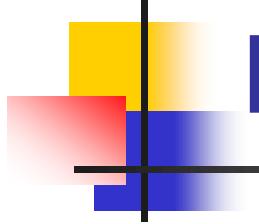
Challenges, basic principles and terminology



Software Testing to Increase Reliability

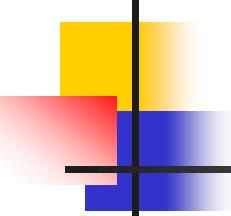
- Widely utilized technique
 - Much common relative to fault prevention and fault tolerance
- Getting more important and challenging as well..





Need for Software Testing

- Accounts for at least half of the development budget
- Principle post-design activity in practice
- Restricting early testing usually increases cost
- Extensive hardware-software integration requires even more testing

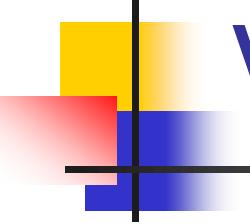


Peculiarities of Software

- Challenging characteristics of software
 - Many different quality requirements
 - Evolving (and deteriorating) structure
 - Inherent non-linearity
 - Uneven distribution of faults

e.g., if an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load;

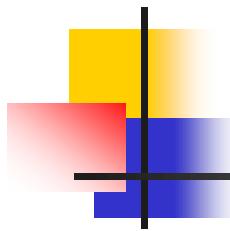
if a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 or 12 elements, as well as on 257 or 1023.



Variety of approaches

- There are **no fixed recipes**
- Test designers must
 - choose and schedule the **right blend of techniques**
 - to reach the required level of quality
 - within cost constraints
 - design a **specific solution** that suits
 - the problem
 - the requirements
 - the development environment



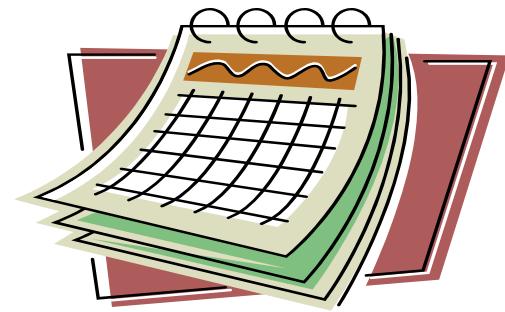


When, What, How



- When does testing activities start? When are they complete?
- What particular techniques should be applied during development?
- How can we assess the readiness of a product?

When to start? When are we complete?



- Test is **not** a (late) phase of software development
- Execution of tests is a small part of the story
- Testing activities **start as soon as** we decide to build a software product, or even before
- Testing **last far beyond** the product delivery as long as the software is in use, to cope with evolution and adaptations to new conditions

Early start..

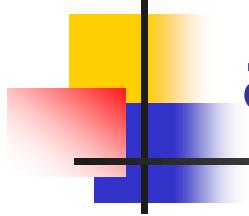
- **Written test objectives** and requirements must be documented
- What are your planned **coverage** levels?
- How much testing is **enough**?
- Common objective – **spend the budget ... test until the ship-date** ...
 - Sometimes called the “**date criterion**”



Long lasting: beyond maintenance

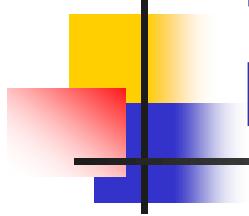
- analysis of changes and extensions
- generation of new test suites for the added functionalities
- re-executions of tests to check for non regression of software functionalities after changes and extensions
- fault tracking and analysis





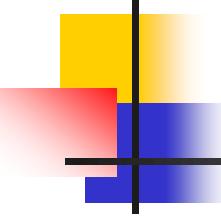
What particular techniques should be applied during development?

- No single analysis or testing technique can serve all purposes; primary reasons for combining them are:
 - Effectiveness for different **fault classes**
e.g., analysis instead of testing for race conditions
 - Applicability at different **points in a project**
e.g., inspection for early requirements validation
 - Differences in **purpose**
e.g., statistical testing to measure reliability
 - Tradeoffs in **cost** and assurance
e.g., expensive technique for key properties

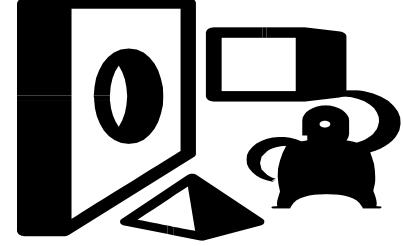


How can we assess the readiness of a product?

- We can not! :)
- Ideal case: *If the system passes an adequate suite of test cases, then it must be correct (or dependable)*
 - But that's impossible!
 - Adequacy of test suites, in the sense above, is **provably undecidable**



Practical (in)Adequacy Criteria



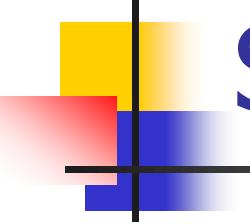
- Criteria that identify inadequacies in test suites
 - *e.g., no test executes a particular program statement*
- *If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite*
- *If a test suite satisfies all the obligations by all the criteria, we do not know definitively that it is an effective test suite, but we have some evidence of its thoroughness*



Analogy: Building Codes



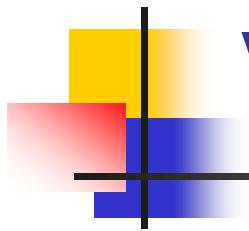
- Building codes are sets of design rules
 - Maximum span between beams in ceiling, floor, and walls; acceptable materials; wiring insulation; ...
 - Minimum standards, subject to judgment of building inspector who interprets the code
- You wouldn't buy a house just because it's "up to code"
 - It could be ugly, badly designed, inadequate for your needs
- But you might avoid a house because it isn't
 - Building codes are inadequacy criteria, like practical test "adequacy" criteria



Software Testing Terms

- Like any field, software testing comes with a large number of **specialized terms** that have particular meanings in this context
- Some of the following terms are **standardized**, some are used **consistently** throughout the literature and the industry, but some **vary** by author, topic, or test organization
- Some **most commonly** used definitions follow..

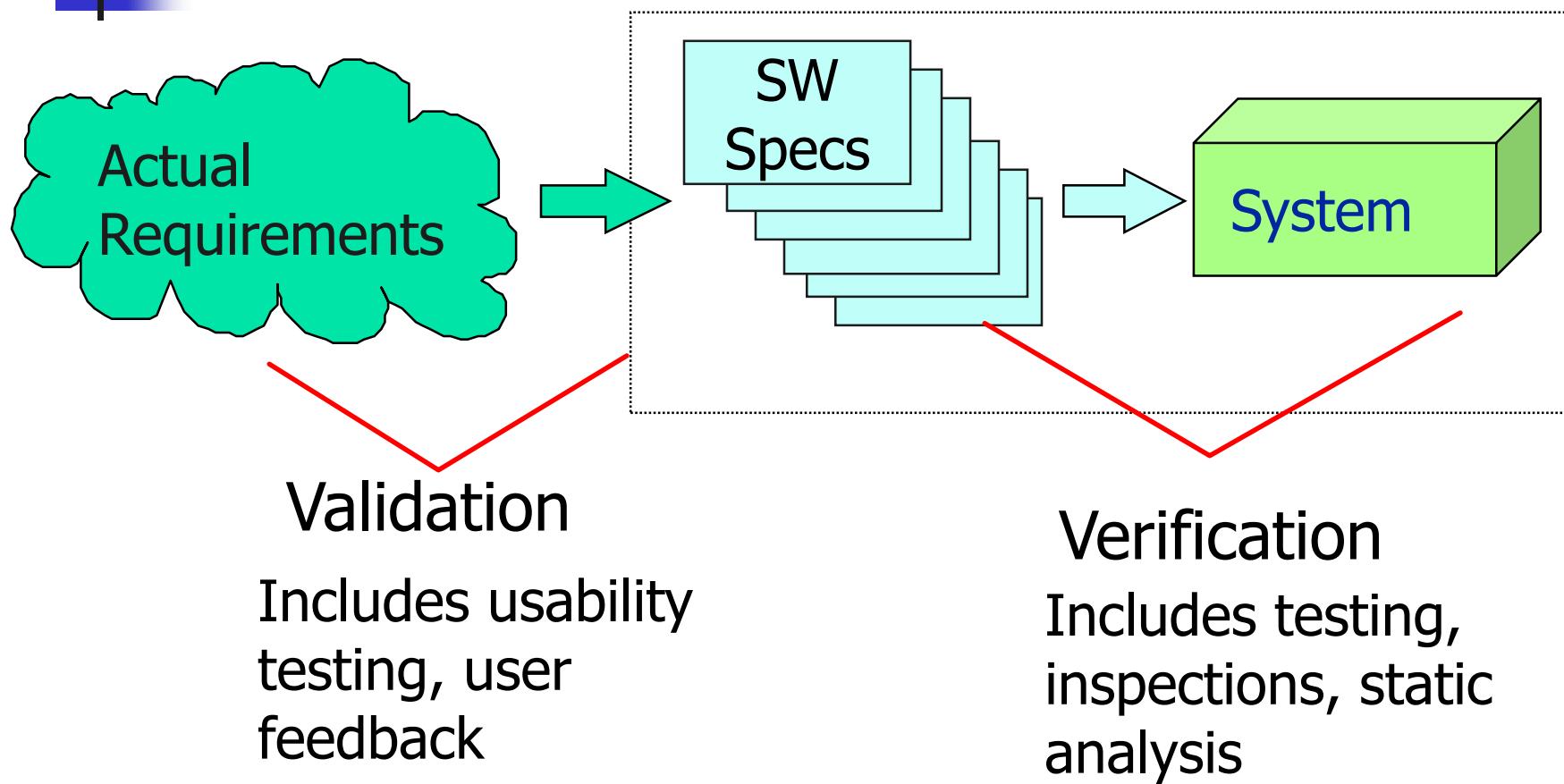




Verification and validation

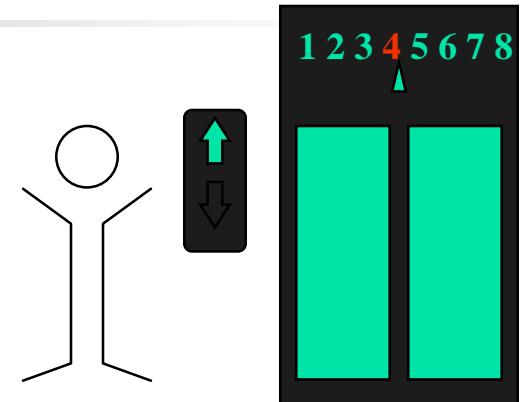
- **Validation:** does the software system meets the user's real needs?
are we building the right software?
- **Verification:** does the software system meets the requirements specifications?
are we building the software right?

Validation and Verification



Verification or validation depends on the specification

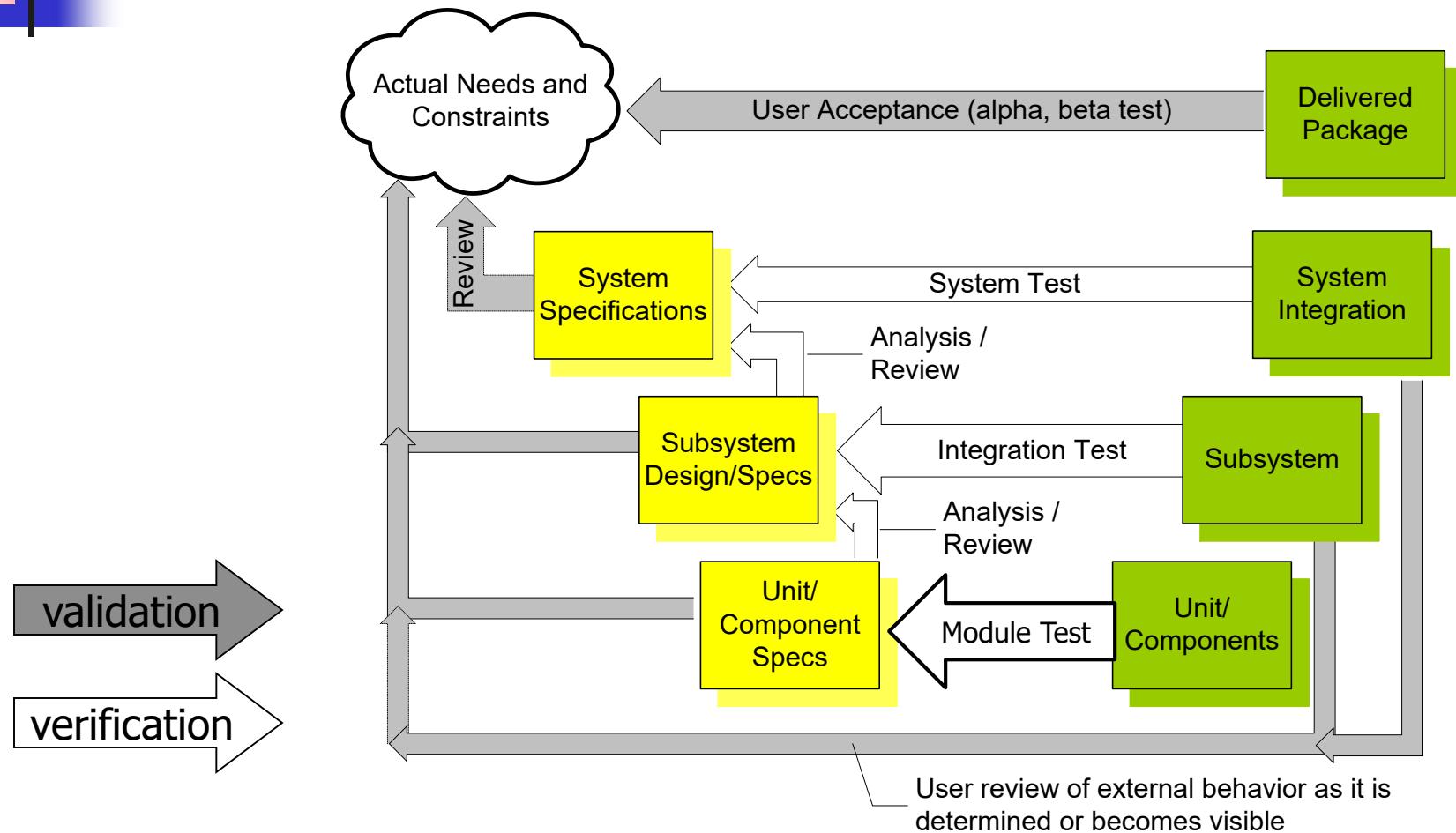
Example: elevator response



Unverifiable (but validatable) spec: ... if a user presses a request button at floor i , an available elevator must arrive at floor i **soon...**

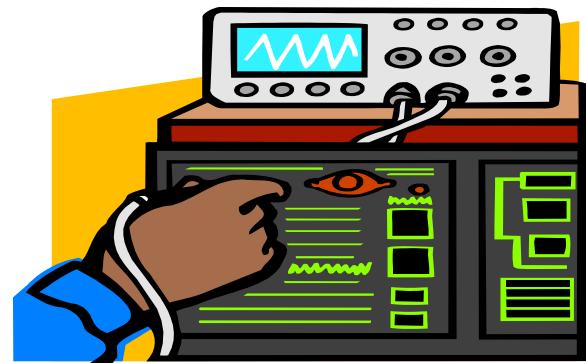
Verifiable spec: ... if a user presses a request button at floor i , an available elevator must arrive at floor i **within 30 seconds...**

Validation and Verification Activities

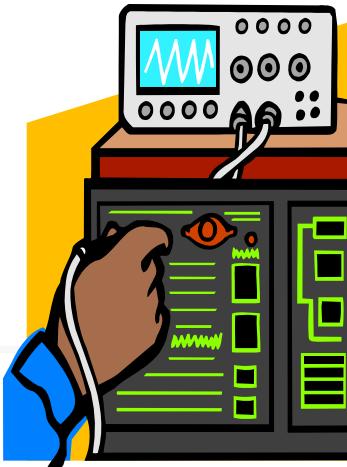


User Acceptance Tests..

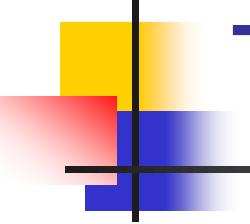
- **Alpha test:** tests performed by users in a controlled environment, observed by the development organization
- **Beta test:** tests performed by real users in their own environment, performing actual tasks without interference or close monitoring



Some useful terminology



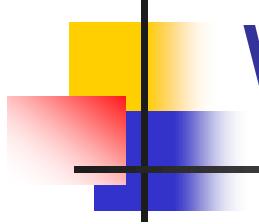
- **Test case:** a set of inputs, execution conditions, and a pass/fail criterion
- **Test suite:** a set of test cases
- **Test case specification:** a requirement to be satisfied by one or more test cases
- **Test or test execution:** the activity of executing test cases and evaluating their results
- **Adequacy criterion:** a predicate that is true (satisfied) or false (not satisfied) of a ⟨program, test suite⟩ pair



Test case, spec, suite..

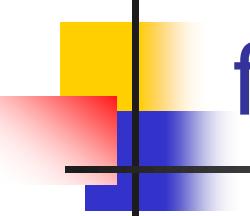
Example:

- A test case specification:
a sorted sequence of length > 2
- A corresponding test case:
“Alpha, Beta, Chi, Omega”
- A test suite:
{ “Alpha, Beta, Chi, Omega”, “Beta, Chi, Omega”, “Alpha, Beta, Gama” }



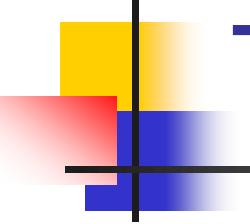
What is a Test Plan?

- Scope
 - Approach
 - Resources
 - Schedule
 - Test Items
 - Testing Tasks
 - Responsibilities
 - Risks and Contingency plan
- 
- of Testing Activities



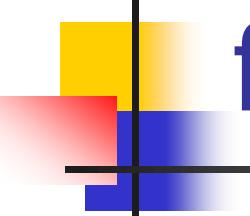
Example Test Plan Contents for System Testing

- Purpose
- Target audience and application
- Deliverables
- Information included
 - Introduction
 - Test items
 - Features tested
 - Features not tested
 - Test criteria
 - Pass / fail standards
 - Hardware and software requirements
 - Responsibilities for severity ratings
 - Staffing & training needs
 - Test schedules
 - Risks and contingencies
 - Approvals



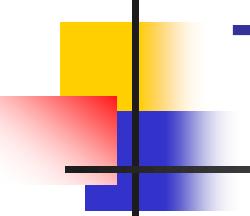
Testing vs. Debugging

- Testing: Finding inputs that cause the software to fail
- Debugging: The process of finding a fault given a failure
 - Diagnosis
 - Fault localization



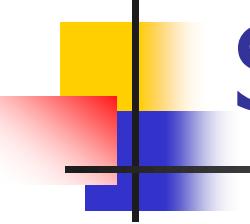
Conditions for observing failures

- Reachability: The location or locations in the program that contain the fault must be reached
- Infection: The state of the program must be incorrect
- Propagation: The infected state must propagate to cause some output of the program to be incorrect



Types of Testing..

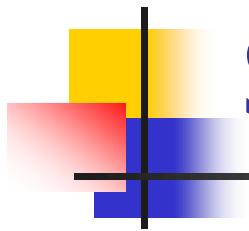
- Functional (*black box*): from software specifications
 - e.g., if spec requires robust recovery from power failure, test obligations should include simulated power failure
- Structural (*white* or *glass box*): from code
 - e.g., traverse each program loop one or more times.
- Model-based: from a model of the system
 - Models used in specification or design, or derived from code
 - e.g., exercise all transitions in communication protocol model
- Fault-based: from hypothesized faults (common bugs)
 - e.g., Check for buffer overflow handling (common vulnerability) by testing on very large inputs



Stress Testing

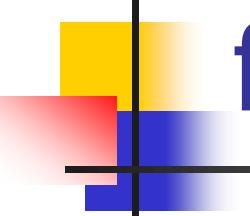
- "Tests that are at (out of) the limit of the software's expected input domain"

- Very **large** numeric values (or very small)
- Very **long** strings, **empty** strings
- **Null** references
- Very **large** files
- **Many users** making requests at the same time
- **Invalid** values



Some (more) terminology

- An analysis of a program P with respect to a formula F is **sound** if the analysis returns true only when P does satisfy F
- An analysis of a program P with respect to a formula F is **complete** if the analysis always returns true when P actually does satisfy F



Sound vs. Complete for a fault detection technique F

- Sound: Fault found only when there is actually a fault
 - but it can miss some of the faults
- ~Precision
- Complete: Fault always found if there is any
 - but can it can lead to false alarms (false positives)
- ~Recall

	Property true	Property false
Answer true	sound, complete	complete
Answer false	sound	



CS 575

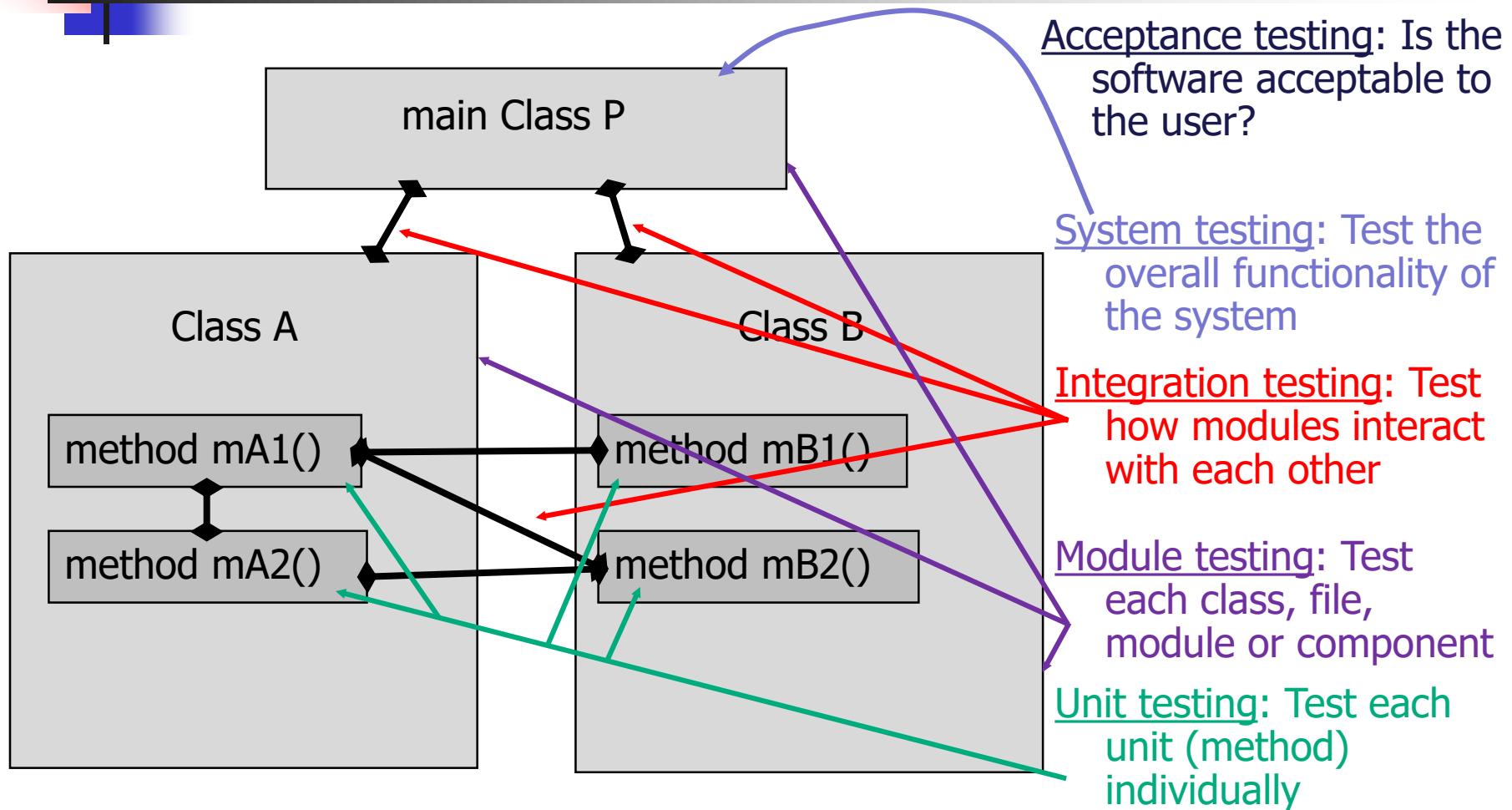
Software Testing and Analysis

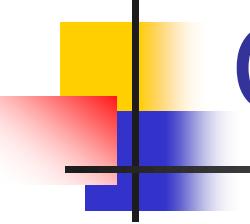
Testing Techniques



(c) Slides adopted from the slides P. Amman & J. Offut (Chapter 1)

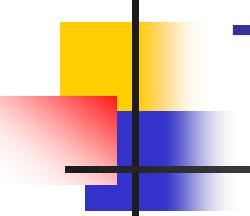
Testing at Different Levels





Changing Notions of Testing

- Traditional categorization is based on **phases** of software development
 - Unit, module, integration, system, etc. ...
- Another categorization can be in terms of **structures** and **criteria**
 - Graphs, logical expressions, syntax, input space, etc.
- Test design is largely the same at each phase
 - Creating the **model** is different
 - Choosing **values** and **automating** the tests is different



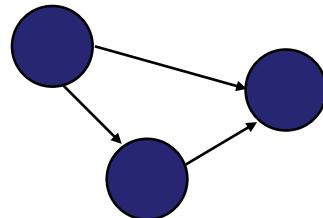
Test Requirements & Criteria

- Test Requirements: must be satisfied or covered
- Test Criterion: A collection of rules and a process that define test requirements
- The Usual Case: Define a model of the software, then find ways to cover it

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on **four types of structures** ...

Criteria based on Structures

- Graphs



- Logical Expressions

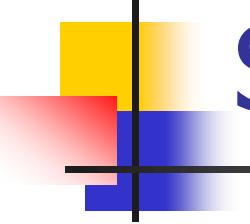
(not X or not Y) and A and B

- Input Domain Characterization

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infis}

- Syntactic Structures

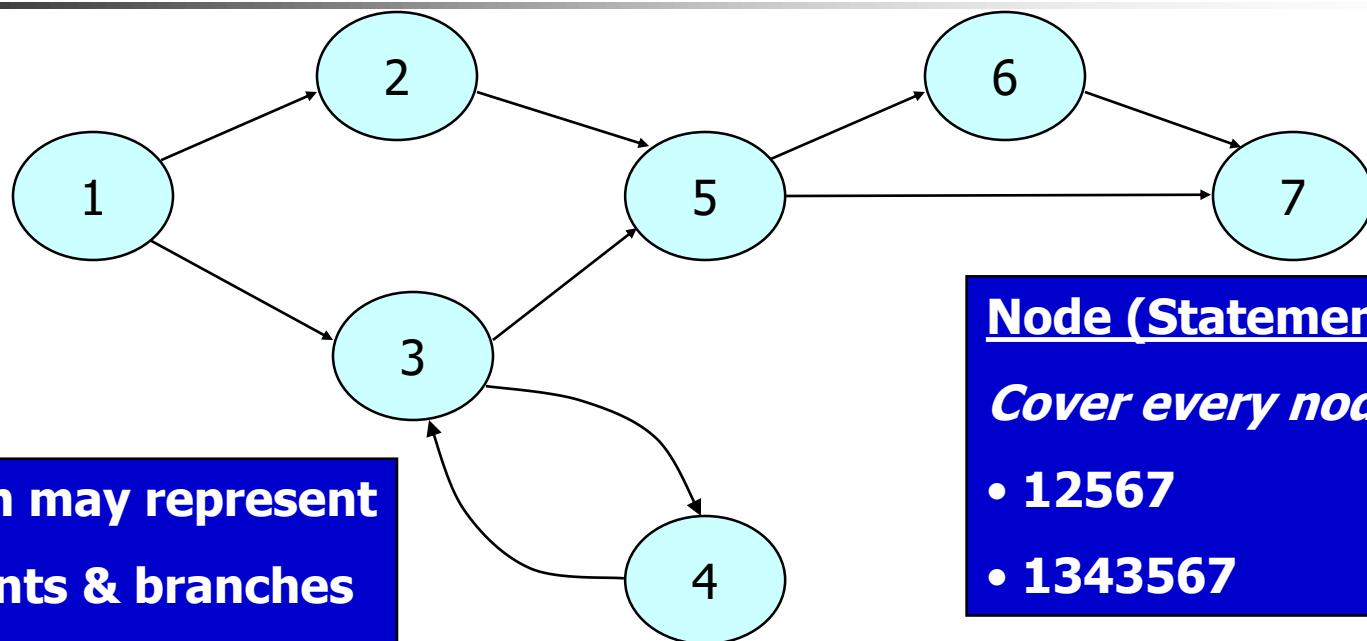
```
if (x > y)  
    z = x - y;  
else  
    z = 2 * x;
```



Source of Structures

- These structures can be **extracted** from lots of software artifacts
 - **Graphs** can be extracted from UML use cases, finite state machines, source code, ...
 - **Logical expressions** can be extracted from decisions in program source, guards on transitions, conditionals in use cases, ...
- This is not the same as “***model-based testing***,” which derives tests from a model that describes some aspects of the system under test
 - The model usually describes part of the **behavior**
 - The **source** is usually **not** considered a model

1. Graph Coverage: Structural



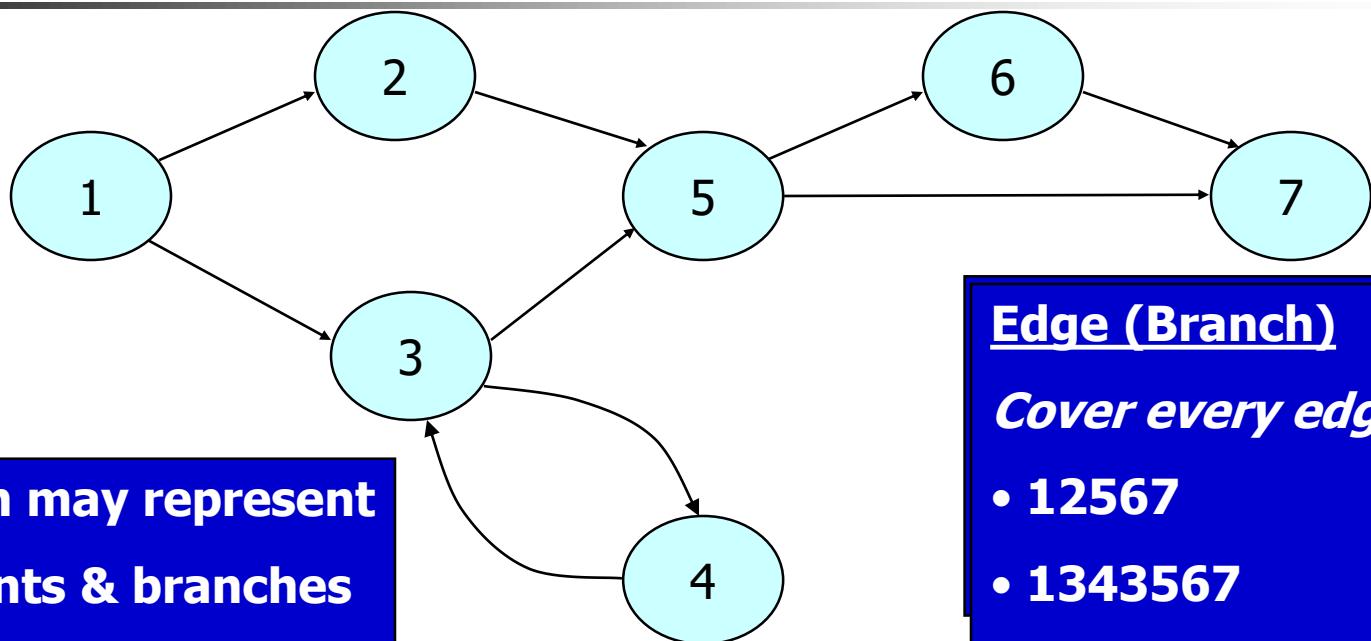
This graph may represent

- statements & branches
- methods & calls
- components & signals
- states and transitions
 -
 -
 -

Node (Statement)
Cover every node

- 12567
- 1343567

1. Graph Coverage: Structural



This graph may represent

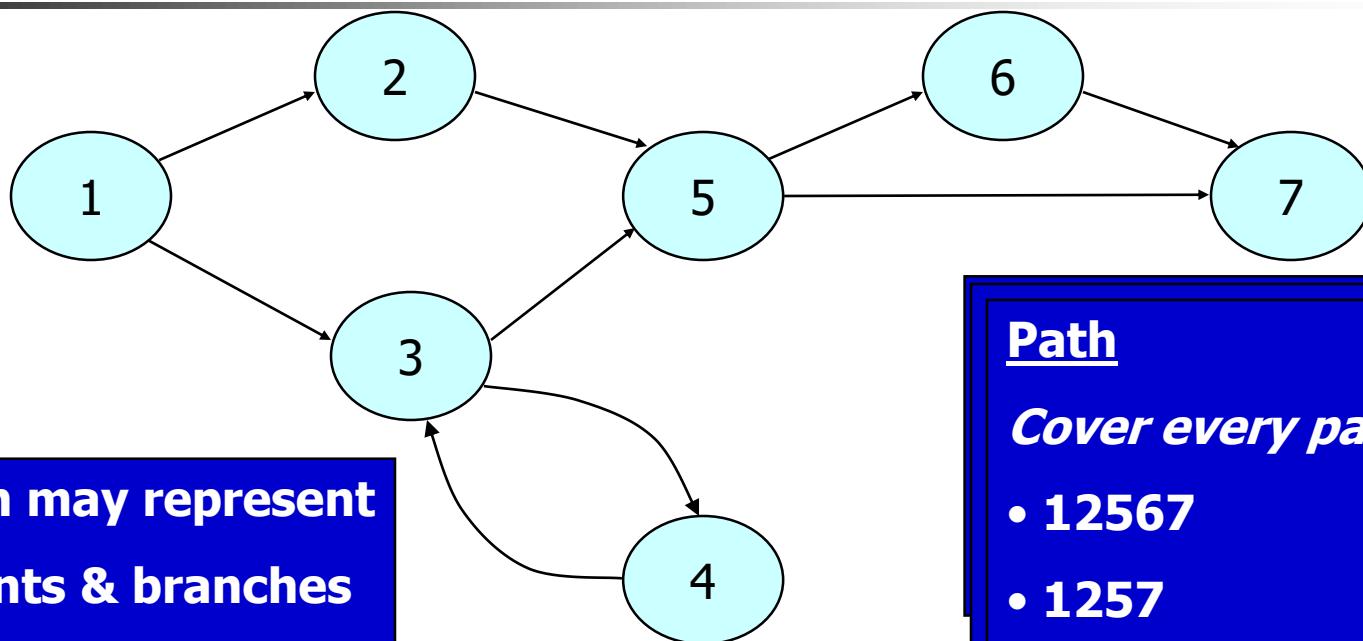
- statements & branches
- methods & calls
- components & signals
- states and transitions
 -
 -
 -

Edge (Branch)

Cover every edge

- 12567
- 1343567
- 1357

1. Graph Coverage: Structural



This graph may represent

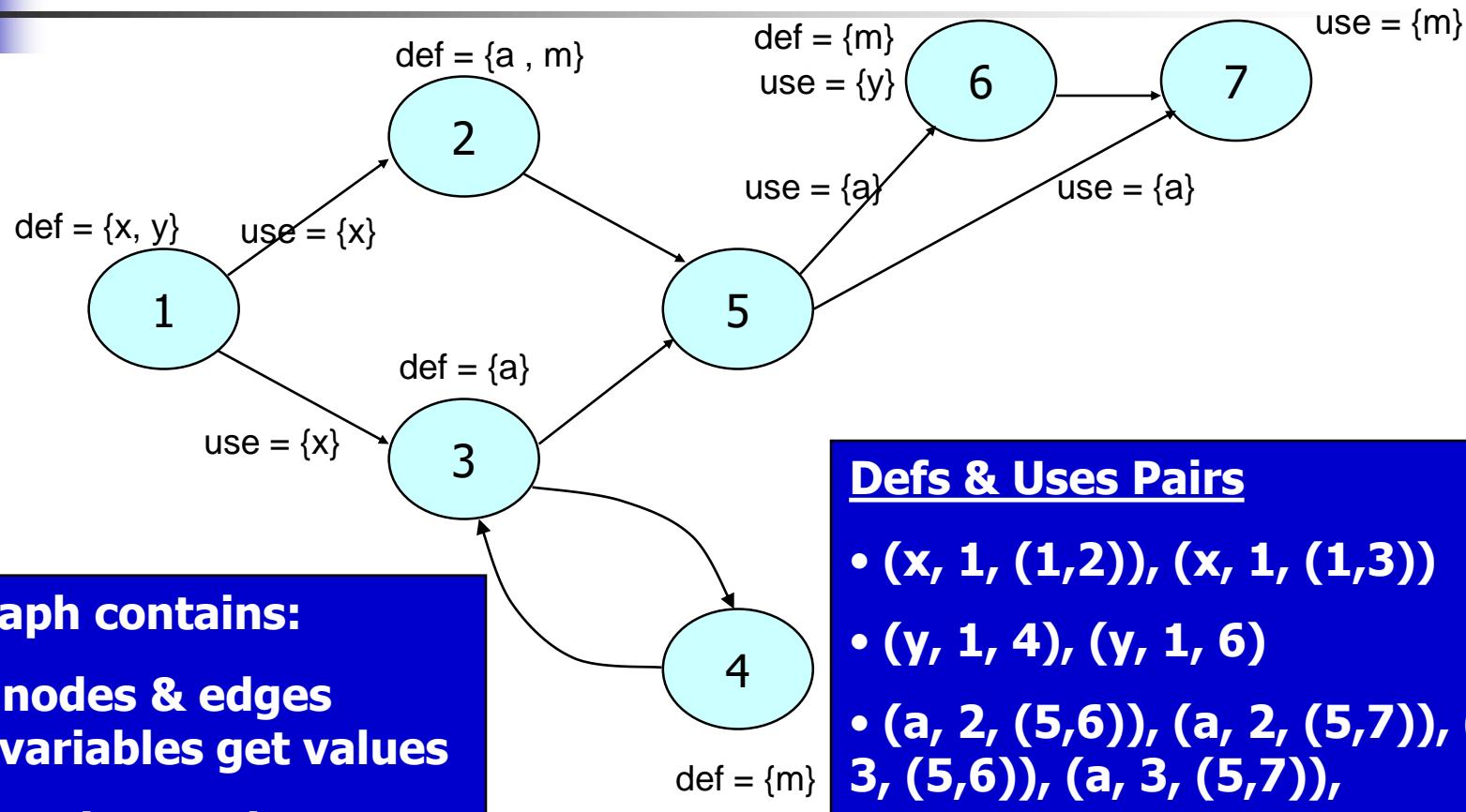
- statements & branches
- methods & calls
- components & signals
- states and transitions
- \vdots

Path

Cover every path

- 12567
- 1257
- 13567
- 1357
- 1343567
- 134357 ...

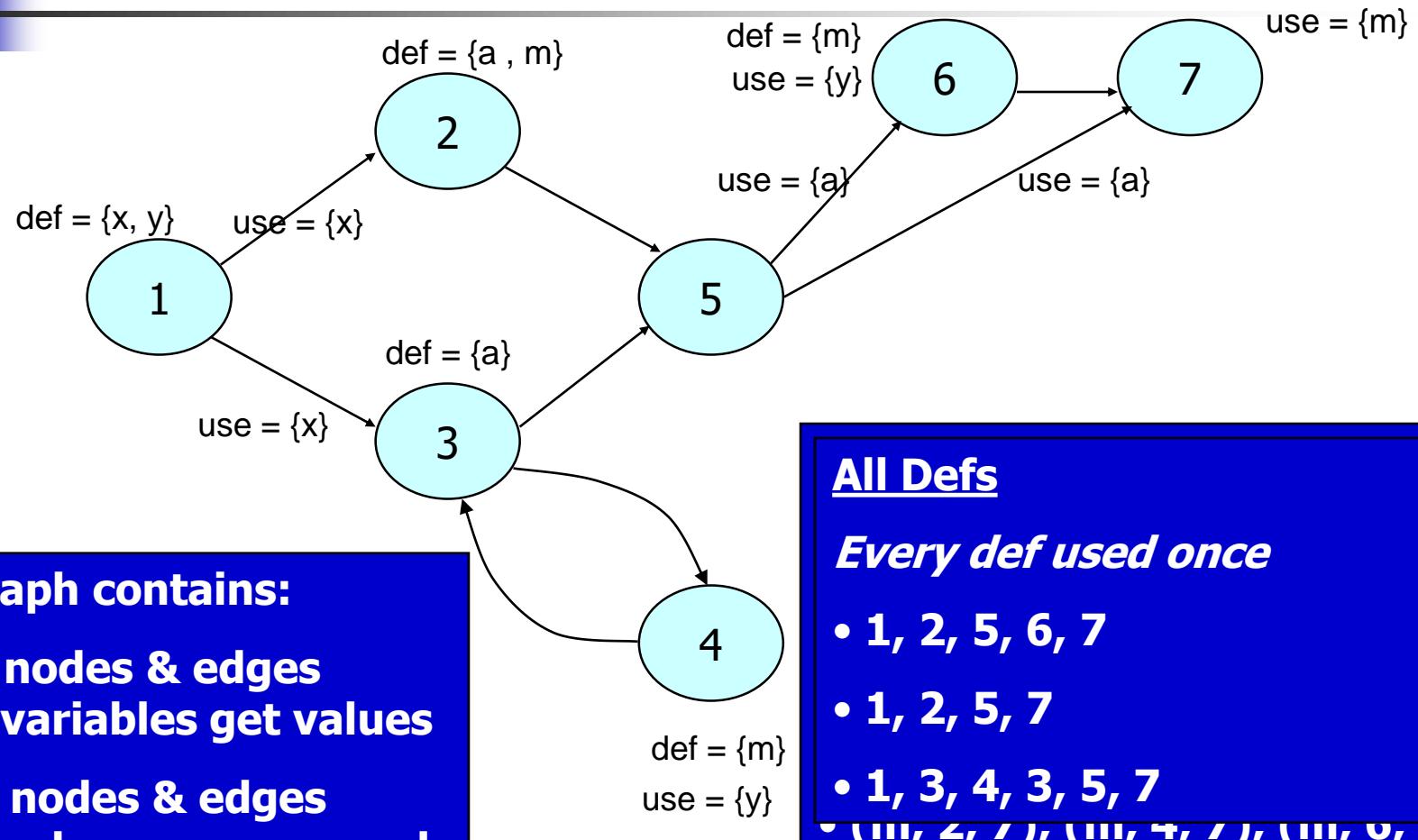
1. Graph Coverage: Data Flow



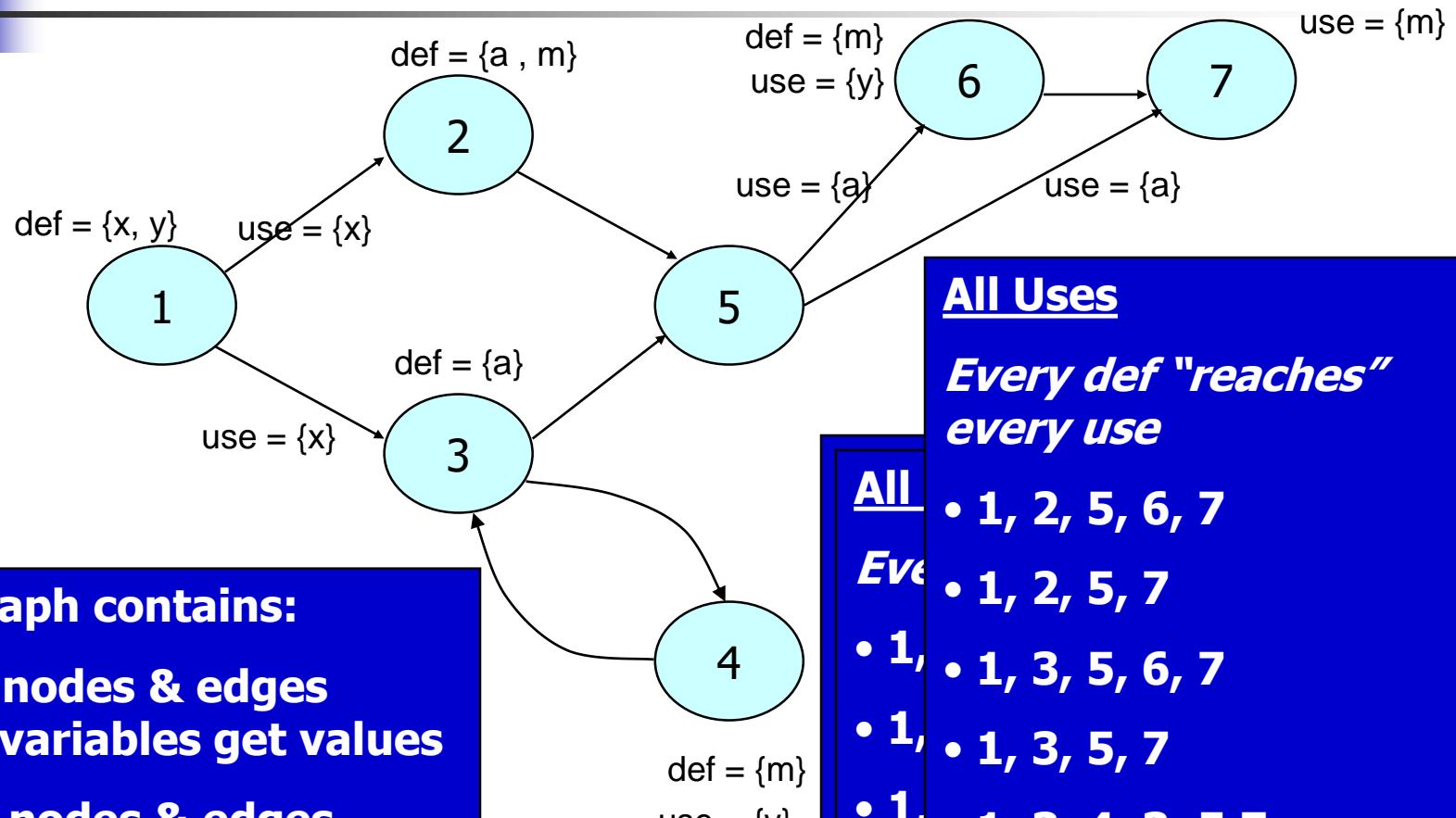
Defs & Uses Pairs

- $(x, 1, (1,2)), (x, 1, (1,3))$
- $(y, 1, 4), (y, 1, 6)$
- $(a, 2, (5,6)), (a, 2, (5,7)), (a, 3, (5,6)), (a, 3, (5,7)),$
- $(m, 2, 7), (m, 4, 7), (m, 6, 7)$

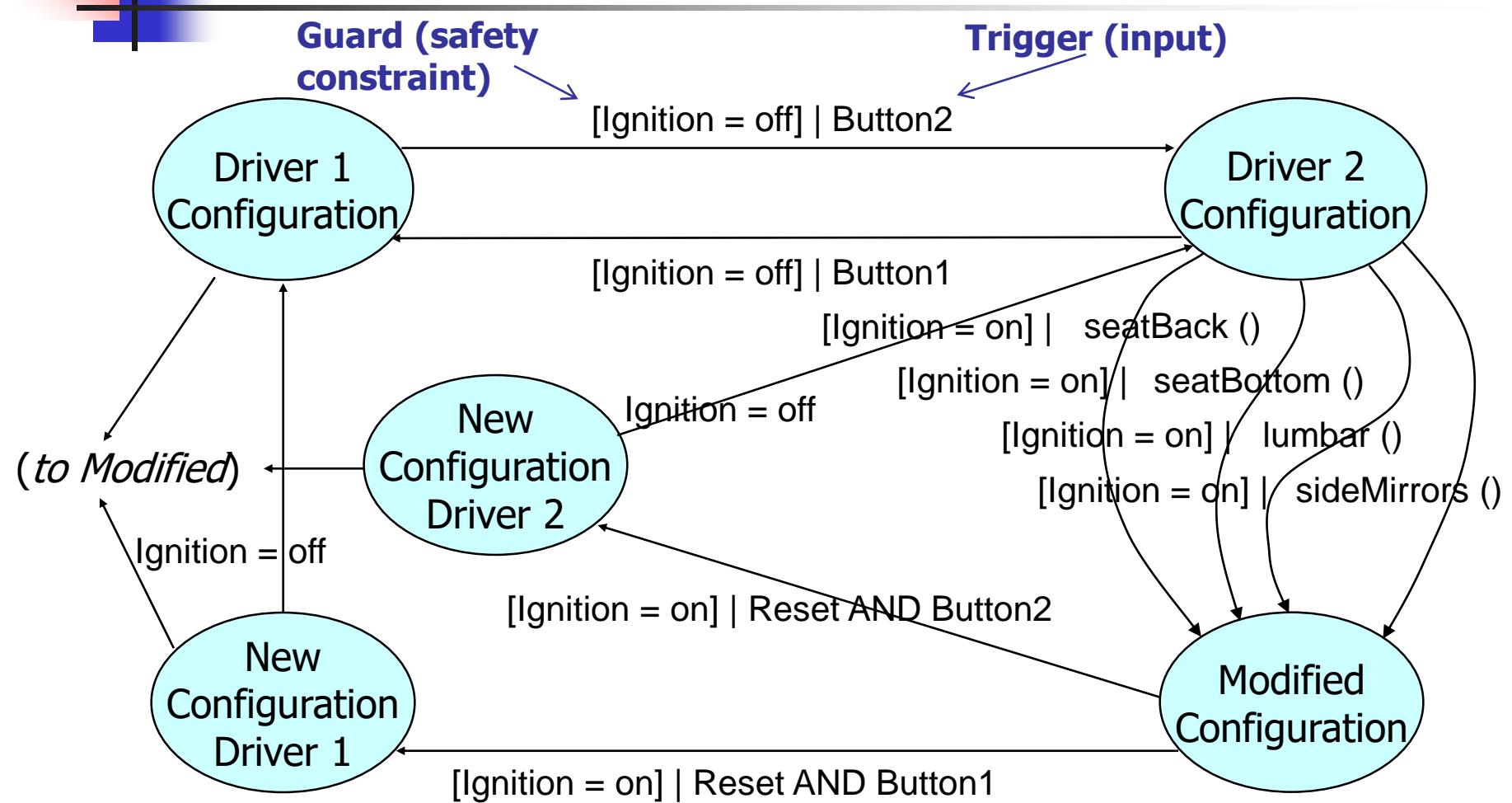
1. Graph Coverage: Data Flow

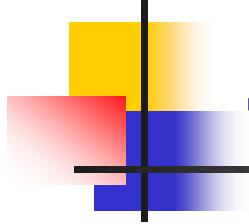


1. Graph Coverage: Data Flow



1. Graph Coverge: Example FSM for Memory Seats in Lexus ES 300





2. Logical Expressions

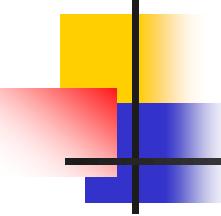
$$((a > b) \text{ or } G) \text{ and } (x < y)$$

Transitions

Program Decision Statements

Software Specifications

**Logical
Expressions**



2. Logical Expressions

$((a > b) \text{ or } G) \text{ and } (x < y)$

- Predicate Coverage: Each predicate must be true and false
 - $(a > b) \text{ or } G = \text{True, False}$
- Clause Coverage : Each clause must be true and false
 - $(a > b) = \text{True, False}$
 - $G = \text{True, False}$
 - $(x < y) = \text{True, False}$
- Combinatorial Coverage : Various combinations of clauses
 - Active Clause Coverage: Each clause must determine the predicate's result

2. Logical Expressions

Active Clause Coverage

$((a > b) \text{ or } G) \text{ and } (x < y)$

With these values for
G and $(x < y)$, $(a > b)$
determines the value
of the predicate

	1	2	3	4	5	6
	T	F	F	F	T	T
	F	F	T	F	T	T

duplicate

3. Input Domain Characterization

- Describe the **input domain** of the software
 - Identify inputs, parameters, or other categorization
 - Partition each input into finite sets of representative values
 - Choose combinations of values

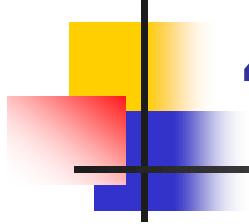
3. Input Domain Characterization

- System level

- Number of students $\{ 0, 1, >1 \}$
- Level of course $\{ 600, 700, 800 \}$
- Major $\{ swe, cs, isa, inf \}$

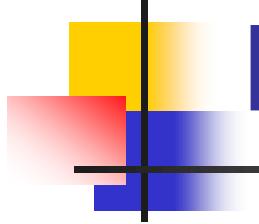
- Unit level

- Parameters $F(\text{int } X, \text{int } Y)$
- Possible values $X: \{ <0, 0, 1, 2, >2 \}, Y: \{ 10, 20, 30 \}$
- Tests $F(-5, 10), F(0, 20), F(1, 30), F(2, 10), F(5, 20)$



4. Syntactic Structures

- Based on a grammar, or other syntactic definition
- Primary example is mutation testing
 1. Induce **small changes** to the program: mutants
 2. Find **tests** that cause the mutant programs to fail
 3. Failure is defined as different output from the original program
 4. Check the output of useful tests on the original program



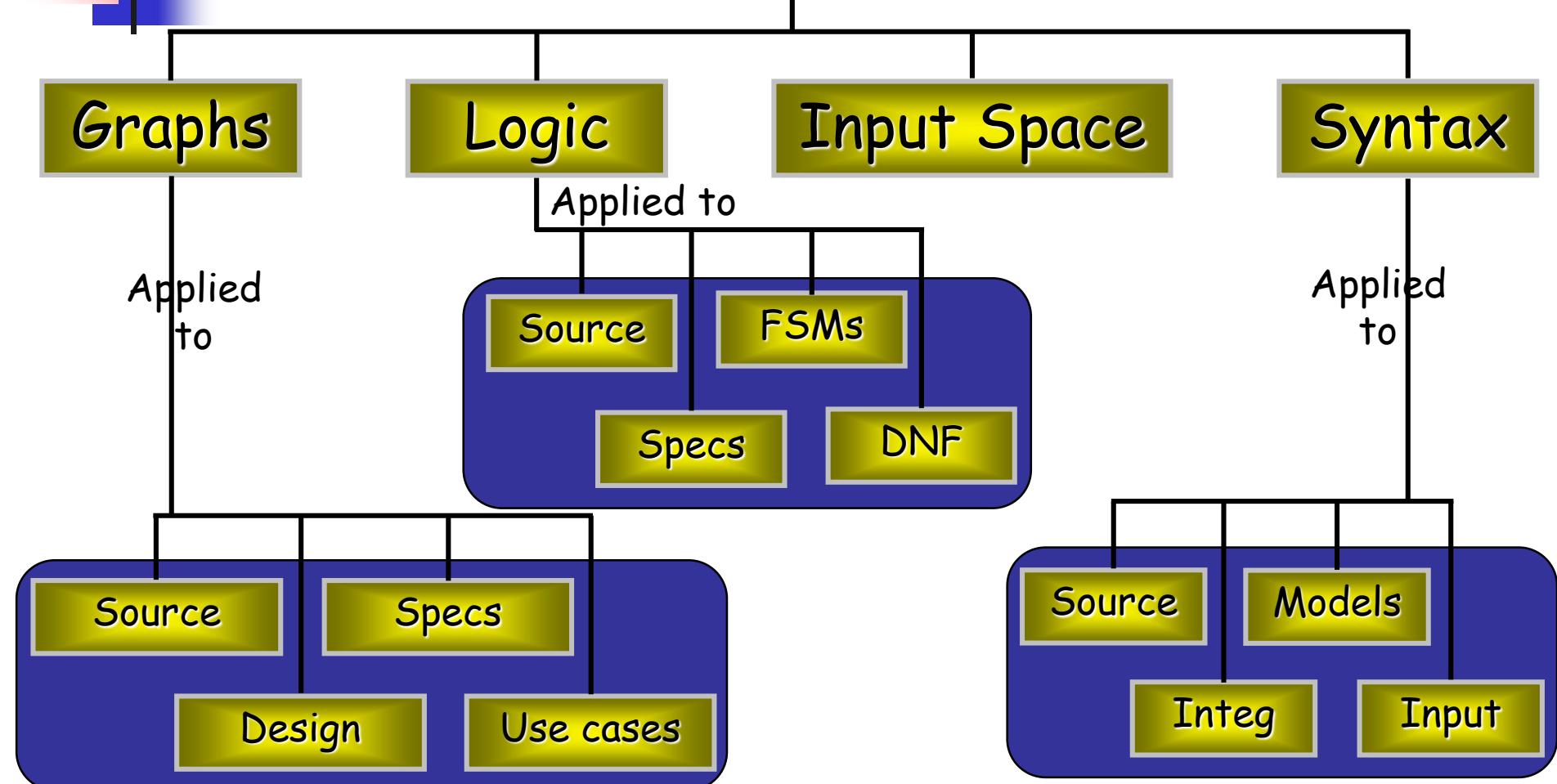
Mutation Testing

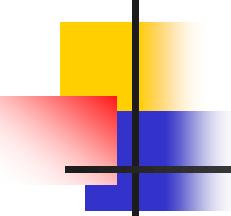
- Example program and mutants

```
if (x > y)
    z = x - y;
else
    z = 2 * x;
```

```
if (x > y)
Δif (x >= y)
    z = x - y;
    Δ z = x + y;
    Δ z = x - m;
else
    z = 2 * x;
```

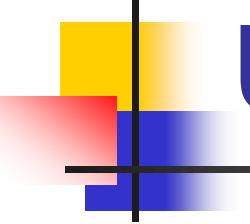
4 Structures for Modeling Software





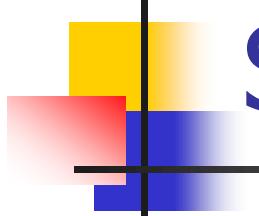
Coverage

- Given a set of test requirements TR for coverage criterion C , a test set T satisfies C coverage if and only if for every test requirement tr in TR , there is at least one test t in T such that t satisfies tr
- 100% coverage is **impossible** in practice
 - No test case values exist that meet the test requirements
 - Dead code
 - Detection of **infeasible test requirements** is formally undecidable for most test criteria



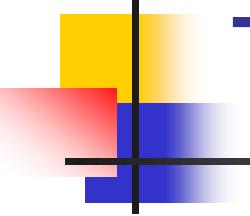
Using Criteria (Metrics)

- Directly generate test values to **satisfy** the criterion
 - research focus
- Generate test values **externally** and **measure** against the criterion
 - industrial practice
 - sometimes misleading, e.g, if tests do not reach 100% coverage, what does that mean?



Comparing Criteria: Subsumption

- Criteria Subsumption : A test criterion $C1$ subsumes $C2$ if and only if every set of test cases that satisfies criterion $C1$ also satisfies $C2$
- Must be true for **every** set of test cases
- *Example* : If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement



Test Coverage Criteria

- Traditional software testing is **expensive** and **labor-intensive**
- Formal coverage criteria are used to decide **which test inputs** to use
- More likely that the tester will **find problems**
- Greater assurance that the software is of **high quality** and **reliability**
- A goal or **stopping rule** for testing
- Criteria makes testing more **efficient** and **effective**



CS 575

Software Testing and Analysis

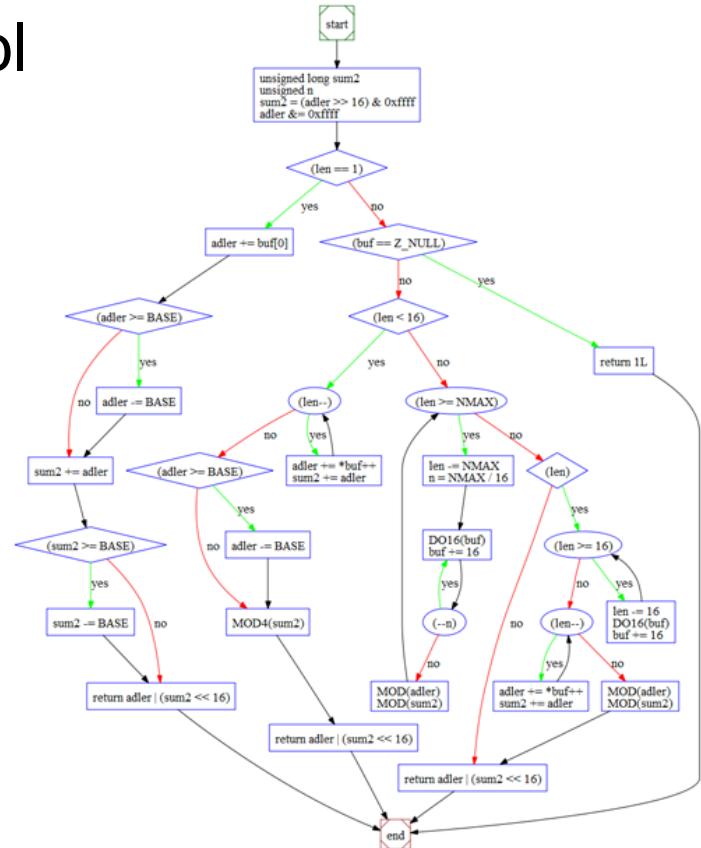
Control Flow Analysis

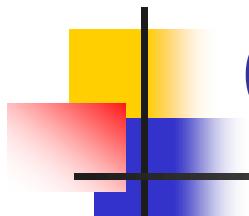


(c) Slides partially adopted from the slides of P. Amman & J. Offutt
and of M. Pezze and M. Young

Control Flow

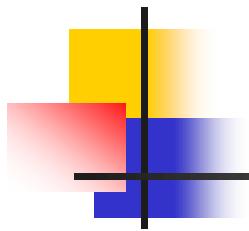
- Goal: Quantify flow of control in a program
 - sequencing of activities
- Basic control structures:
 - Sequence
 - Selection
 - Iteration
- Advanced control structures:
 - Procedure/function/agent call
 - Recursion (self-call)
 - Interrupt
 - Concurrence





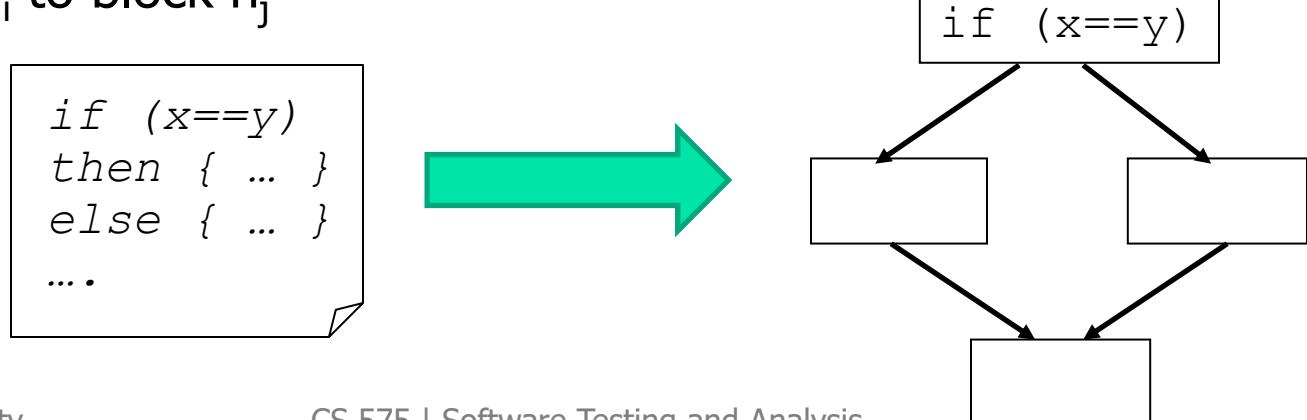
Control Flow Analysis

- **Control Flow** is a sequence of operations represented by:
 - Control flow graph
 - Control dependence graph
 - Call graph
- **Control Flow Analysis:** analyzing a program to discover its control structure



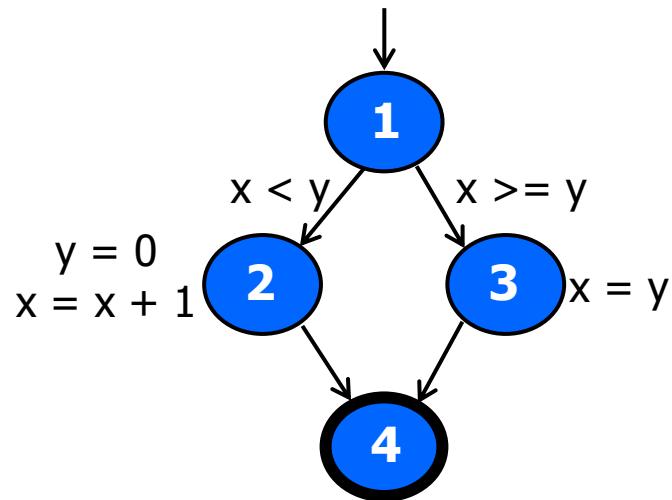
Basic Control Flow Graph

- Models flow of control in the program
- CFG = (N, E) is a directed graph
 - Node $n \in N$: basic blocks, i.e., a maximal sequence of statements with a single entry point and single exit point (no internal branches)
 - Edge $e = (n_i, n_j) \in E$: possible transfer of control from block n_i to block n_j

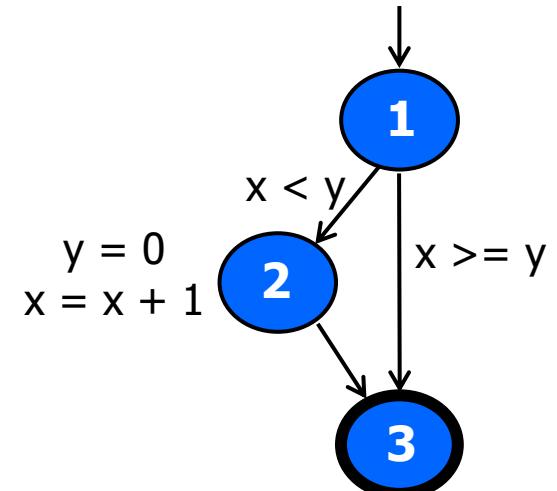


CFG: The if Statement

```
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```

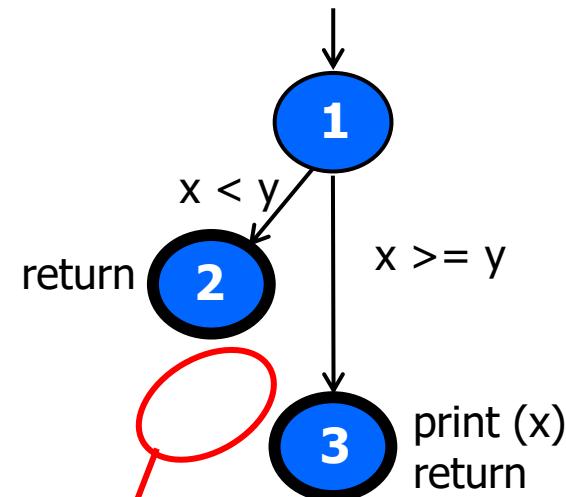


```
if (x < y)
{
    y = 0;
    x = x + 1;
}
```



CFG: The if-Return Statement

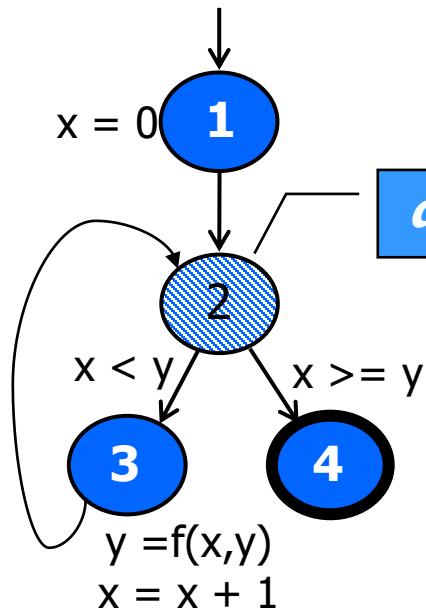
```
if (x < y)
{
    return;
}
print (x);
return;
```



No edge from node 2 to 3.
The return nodes must be distinct.

CFG : while and for Loops

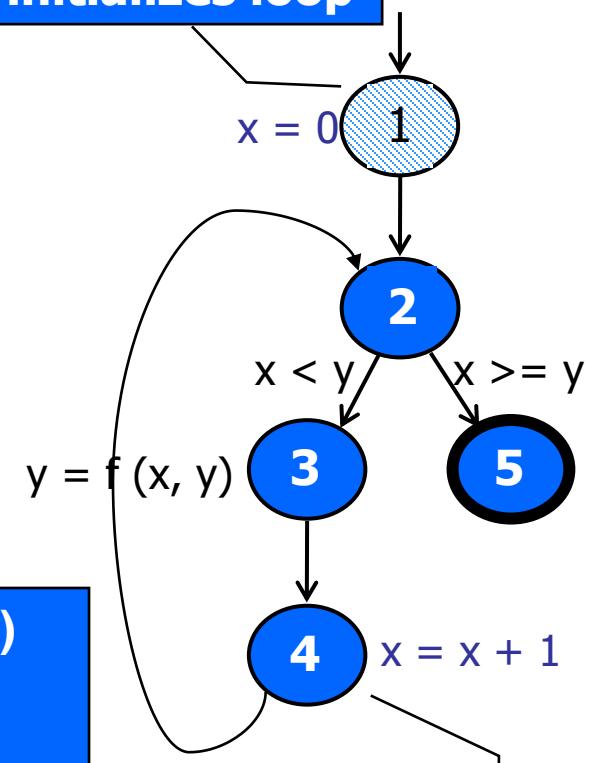
```
x = 0;  
while (x < y)  
{  
    y = f (x, y);  
    x = x + 1;  
}
```



dummy node

initializes loop

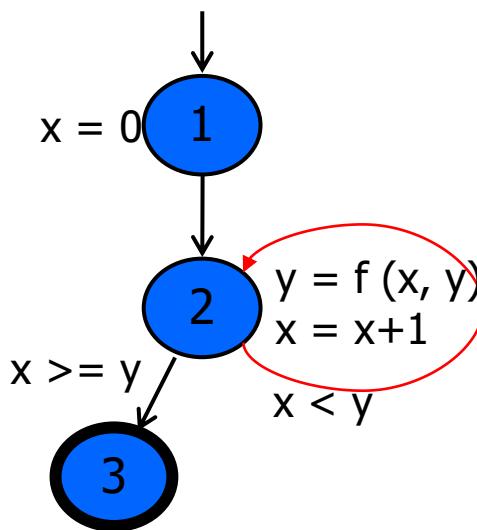
```
for (x = 0; x < y; x++)  
{  
    y = f (x, y);  
}
```



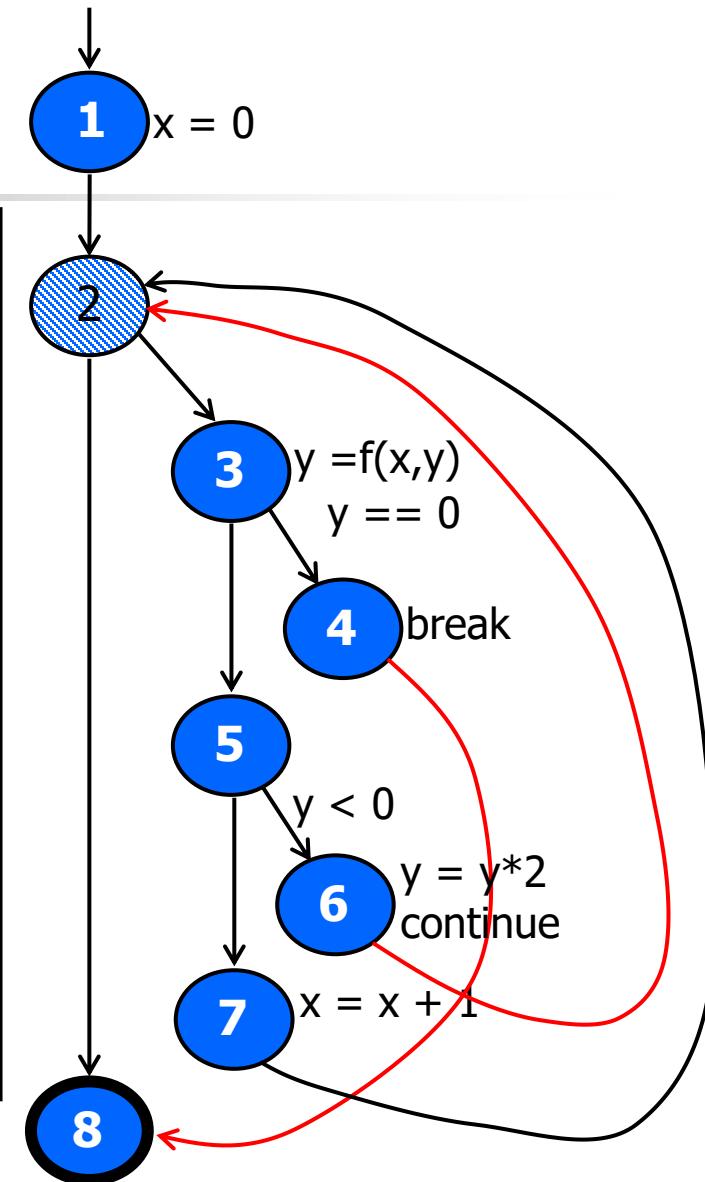
increments loop

CFG: do, break and continue

```
x = 0;  
do  
{  
    y = f (x, y);  
    x = x + 1;  
} while (x < y);  
println (y)
```

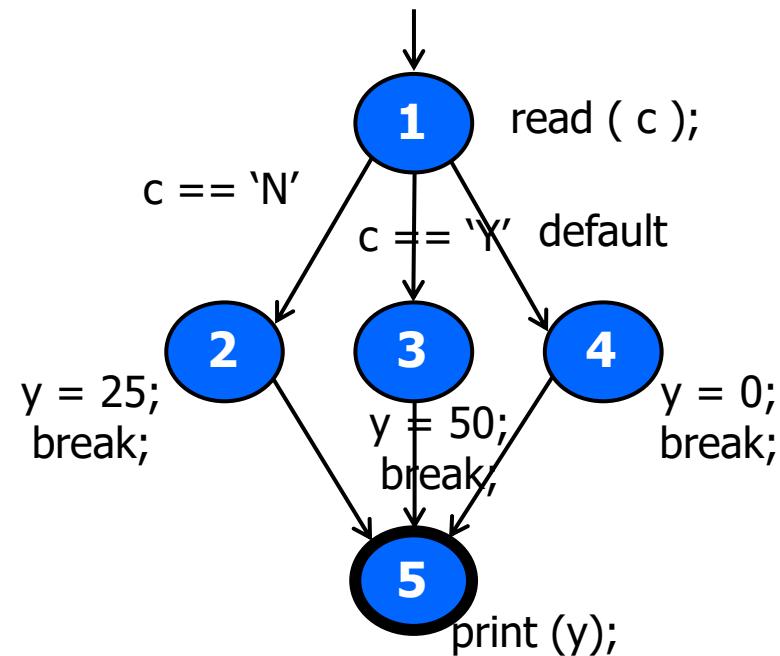


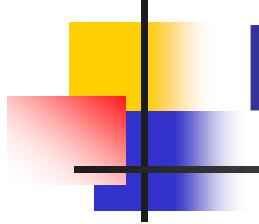
```
x = 0;  
while (x < y)  
{  
    y = f (x, y);  
    if (y == 0)  
    {  
        break;  
    } else if (y < 0)  
    {  
        y = y*2;  
        continue;  
    }  
    x = x + 1;  
}  
print (y);
```



CFG: case (switch)

```
read ( c );
switch ( c )
{
    case 'N':
        y = 25;
        break;
    case 'Y':
        y = 50;
        break;
    default:
        y = 0;
        break;
}
print (y);
```





Nodes in CFG

- If there is an edge from n_i to n_j
 - n_i is a **predecessor** of n_j
 - n_j is a **successor** of n_i
- For any node n
 - $\text{pred}(n)$: the set of predecessors of n
 - $\text{succ}(n)$: the set of successors of n
 - is a **predicate/branch node** if $\text{out-degree}(n) > 1$
 - is a **terminal/end node** if $\text{out-degree}(n) = 0$

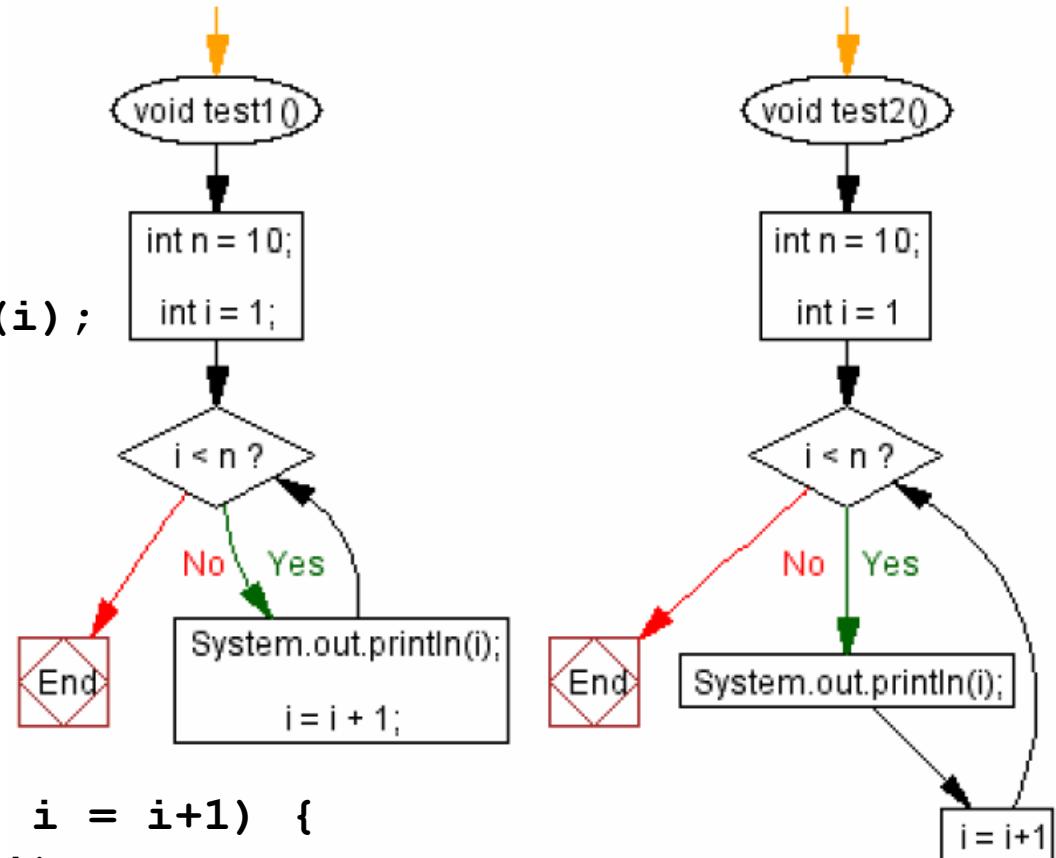
in/out-degree: the number of ingoing/outgoing edges to/from a node

Examples: Visustin CFG generator

Demo version available: <http://www.aivosto.com/visustin.html>

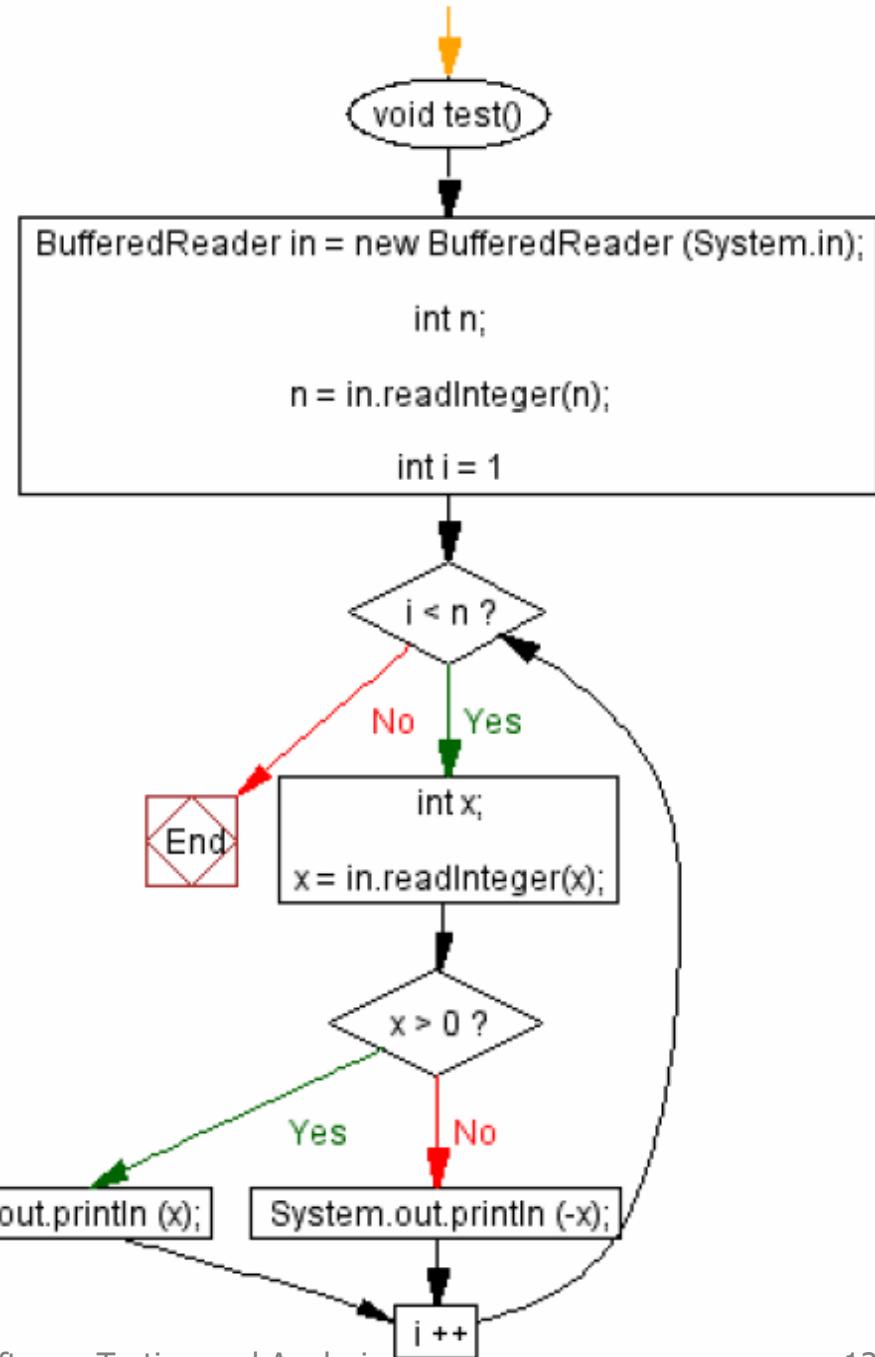
```
void test1() {  
    int n = 10;  
    int i = 1;  
    while ( i < n) {  
        System.out.println(i);  
        i = i + 1;  
    }  
}
```

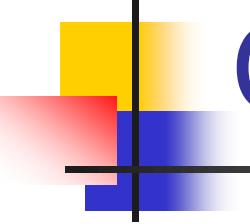
```
void test2() {  
    int n = 10;  
    for (int i = 1; i < n; i = i+1) {  
        System.out.println(i);  
    }  
}
```



Examples:

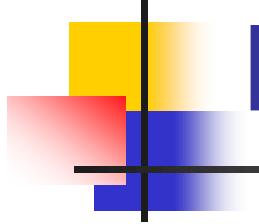
```
void test() {  
    BufferedReader in =  
        new BufferedReader (System.in);  
    int n;  
    n = in.readInteger();  
    for (int i = 1; i < n; i++) {  
        int x;  
        x = in.readInteger();  
        if ( x > 0) {  
            System.out.println (x);  
        }  
        else {  
            System.out.println (-x);  
        }  
    }  
}
```





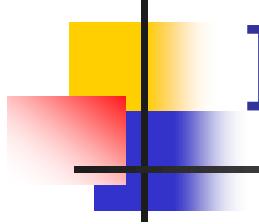
Other CFG Generator Tools

- Eclipse plugin for CFG Generator
 - <http://eclipsecfg.sourceforge.net/>
- GNU tools
 - <http://gcc.gnu.org/>
- Avrora tool for assembly language
 - <http://compilers.cs.ucla.edu/avrora/cfg.html>
- and many more available on the Web..



Dominance

- Node d of a CFG *dominates* node n if every path from the entry node of the graph to n passes through d ($d \text{ dom } n$)
 - $\text{Dom}(n)$: the set of dominators of node n
 - Every node dominates itself: $n \in \text{Dom}(n)$
 - Node d strictly dominates n if $d \in \text{Dom}(n)$ and $d \neq n$
 - Dominance-based loop recognition: entry of a loop dominates all nodes in the loop

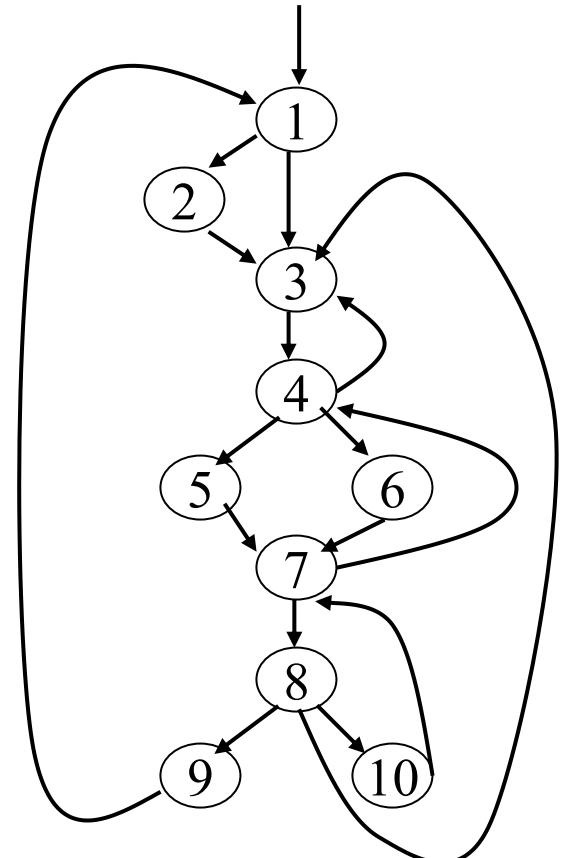


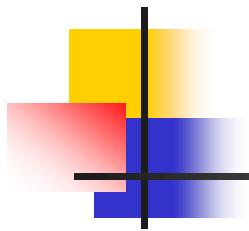
Immediate Dominator

- Each node n has a unique *immediate dominator* m which is the last dominator of n on any path from the entry to n ($m \text{ idom } n$), $m \neq n$
 - The immediate dominator m of n is the strict dominator of n that is closest to n

Dominator Example

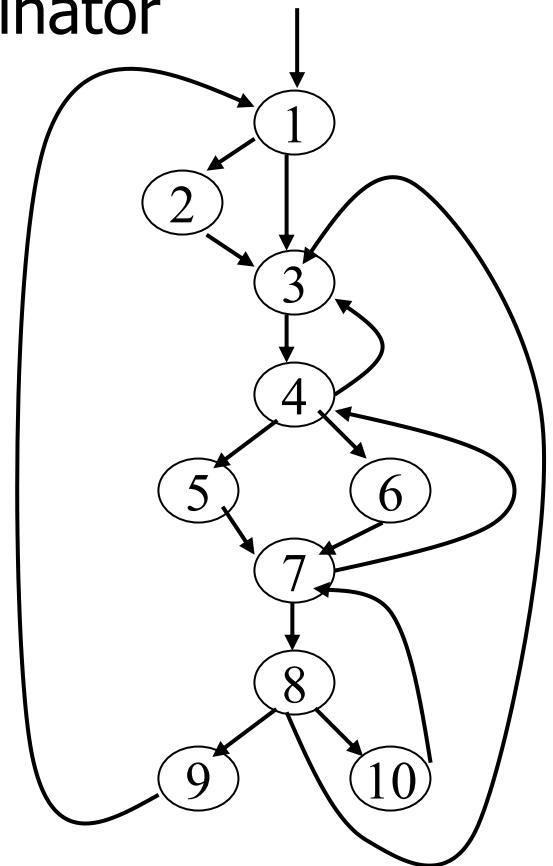
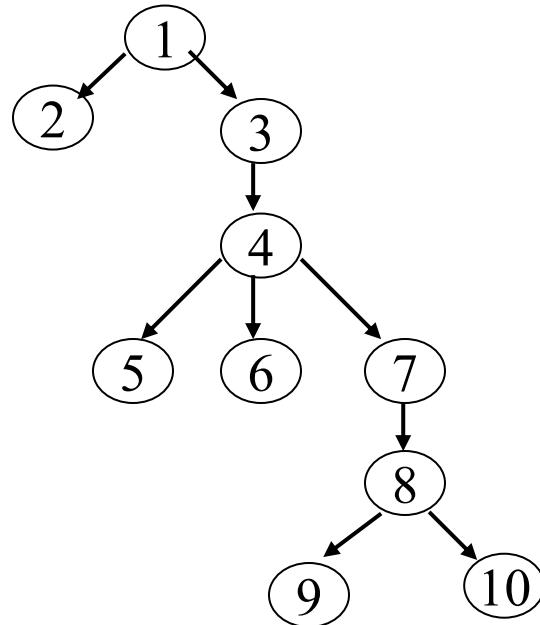
Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8



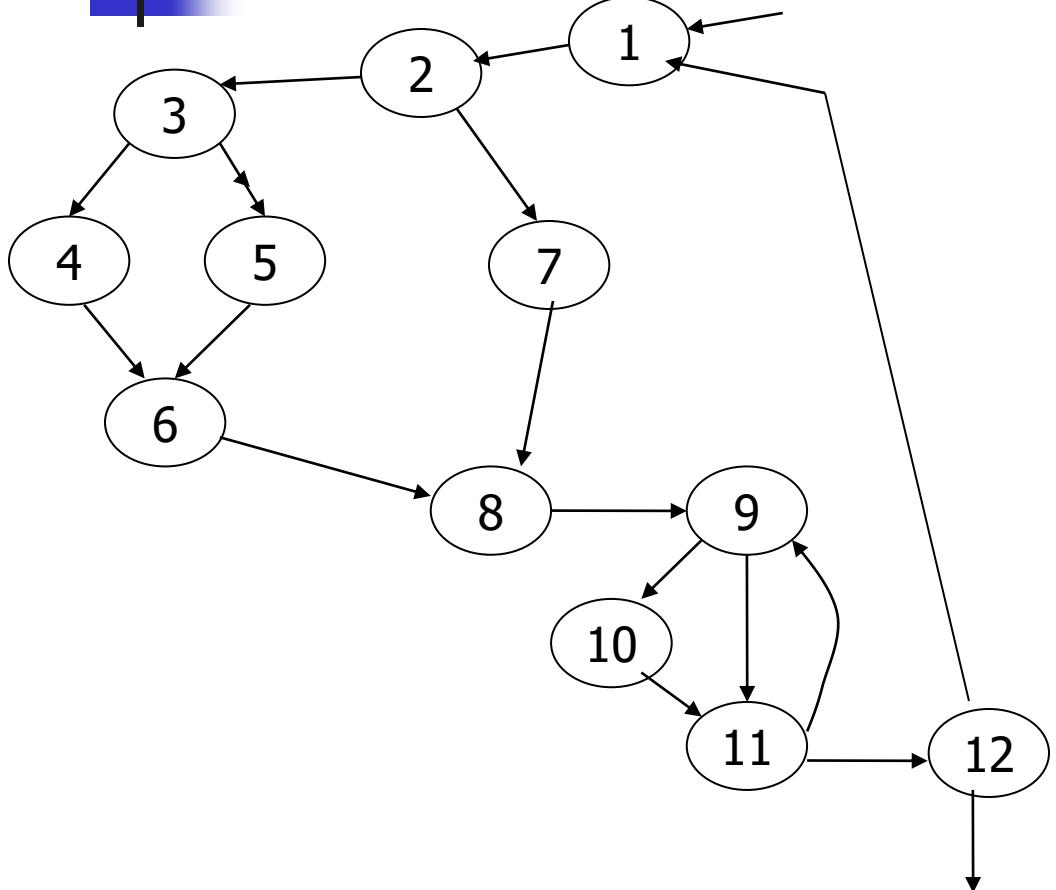


Dominator Trees

- A node's parent is its immediate dominator

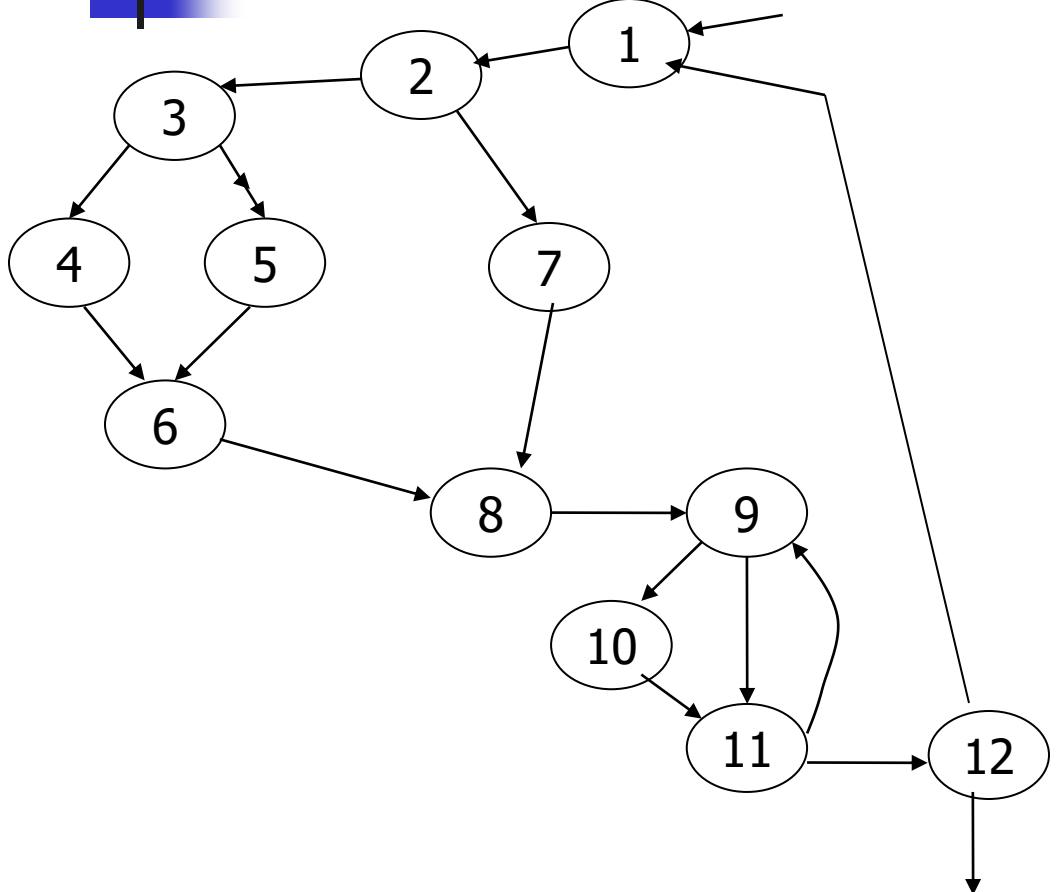


Exercise

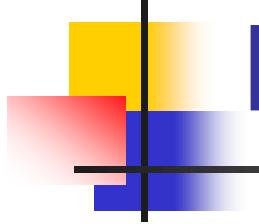


Block	Dom	IDom
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

Exercise



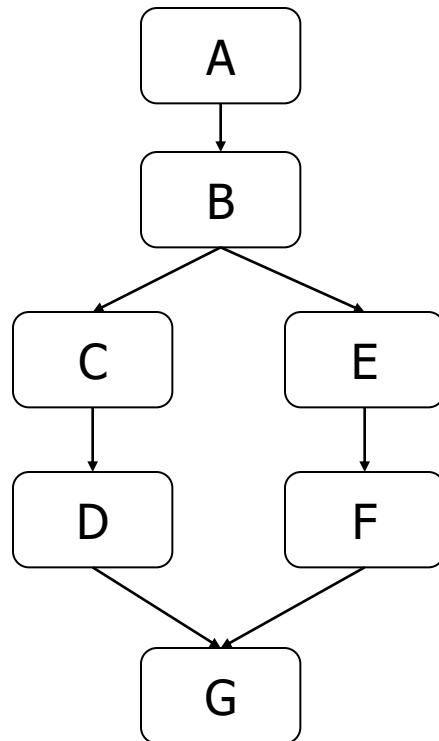
Block	Dom	IDom
1	1	-
2	1,2	1
3	1,2,3	2
4	1,2,3,4	3
5	1,2,3,5	3
6	1,2,3,6	3
7	1,2,7	2
8	1,2,8	2
9	1,2,8,9	8
10	1,2,8,9,10	9
11	1,2,8,9,11	9
12	1,2,8,9,11,12	11



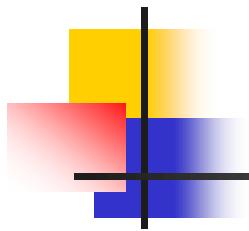
Post-Dominators

- **Post-dominators:** Calculated in the reverse of the CFG, using a special “exit” node as the root.

Example:



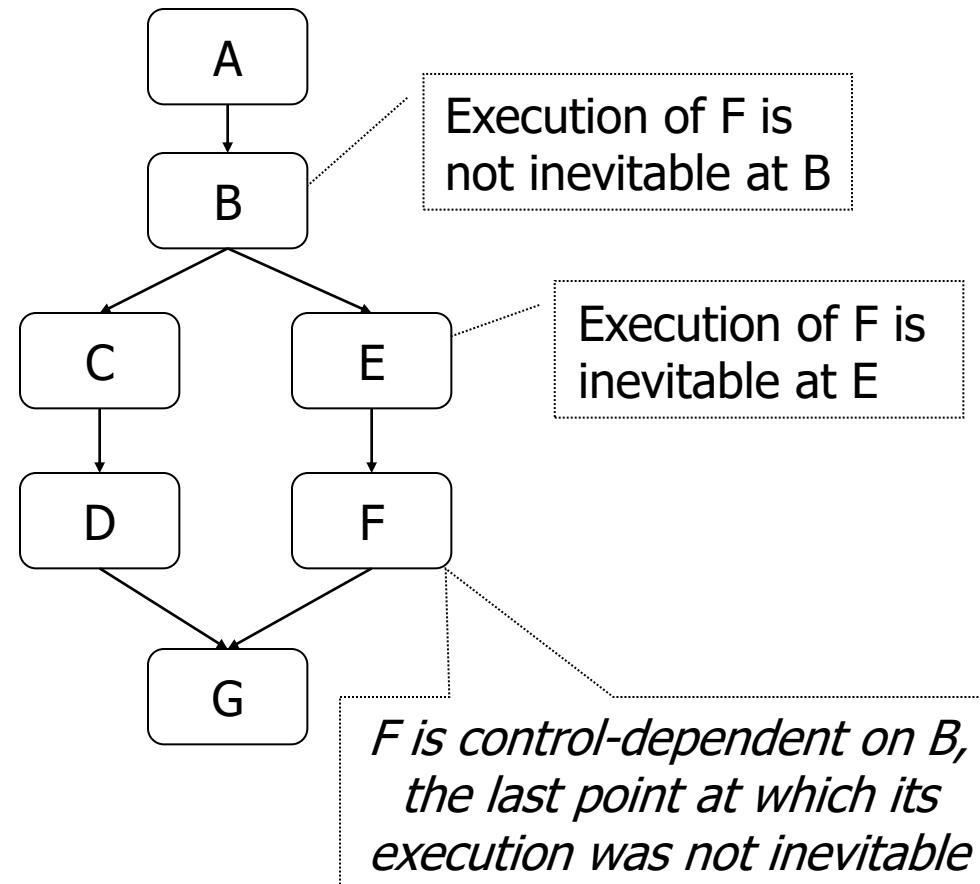
- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
 - C does *not* post-dominate B
- B is the immediate pre-dominator of G
 - F does *not* pre-dominate G

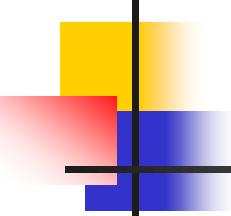


Control dependence

- Control dependence defined by post-dominators:
 - Consider again a node N that is (not always) reachable
 - There must be some node C with the following property:
 - C is a predicate node
 - C is not post-dominated by N
 - a successor of C in the CFG is post-dominated by N
 - Then, N is control-dependent on C .
- Intuitively, C was the last decision that controlled whether N executed

Control Dependence



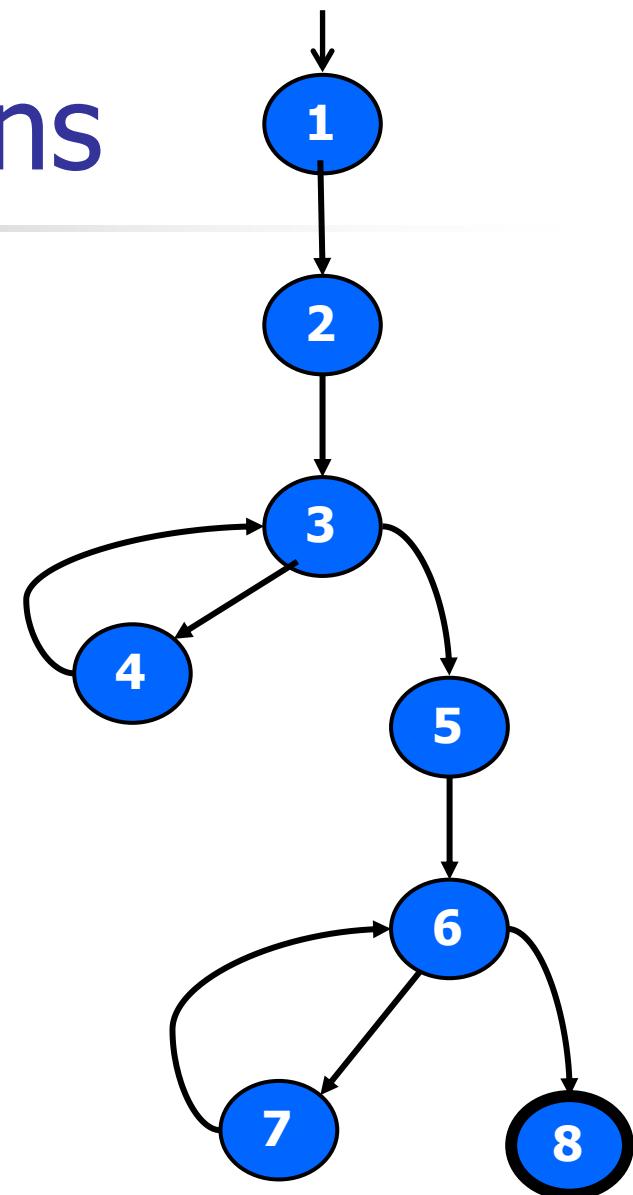


Coverage Criteria Based on Control Flow and any Graph Model in General

- Node Coverage (NC)
- Edge Coverage (EC)
 - ~ Transition Coverage
- Edge-Pair Coverage (EPC)
 - each reachable path of length up to 2
- Prime Path Coverage (PPC)
- Complete Path Coverage (CPC)
 - Not practical in general

Covering Transitions

Edge Coverage	
TR	Test Path
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	



Covering Transitions

Edge-Pair Coverage

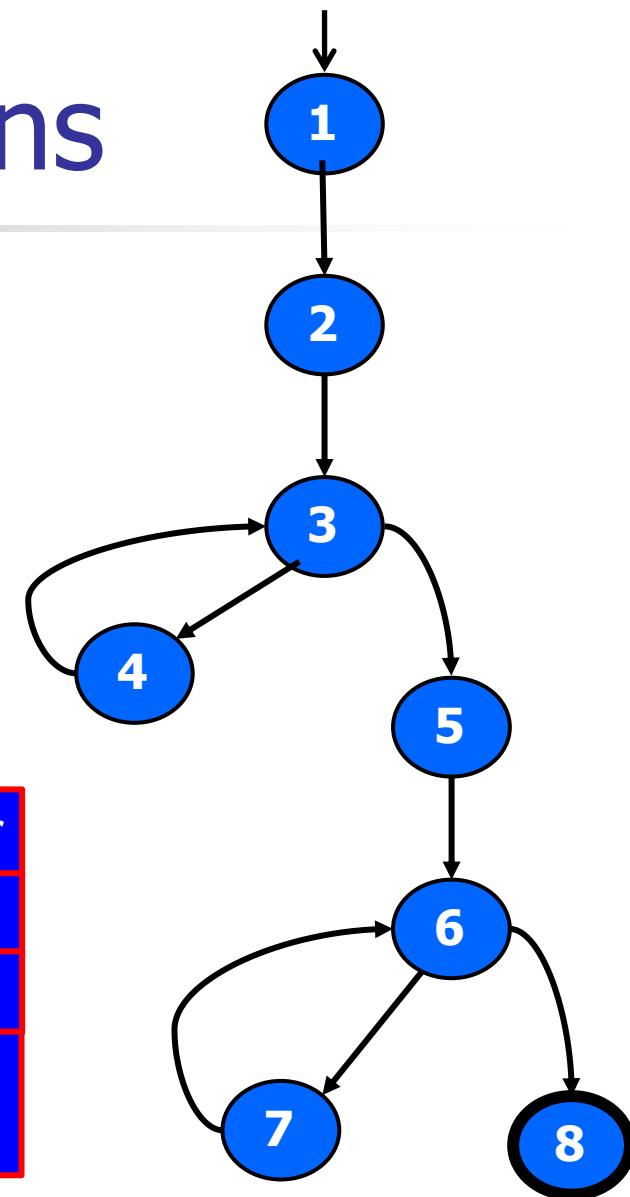
TR

- A. [1, 2, 3]
- B. [2, 3, 4]
- C. [2, 3, 5]
- D. [3, 4, 3]
- E. [3, 5, 6]
- F. [4, 3, 5]
- G. [5, 6, 7]
- H. [5, 6, 8]
- I. [6, 7, 6]
- J. [7, 6, 8]
- K. [4, 3, 4]
- L. [7, 6, 7]

Test Paths

- i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
- ii. [1, 2, 3, 5, 6, 8]
- iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

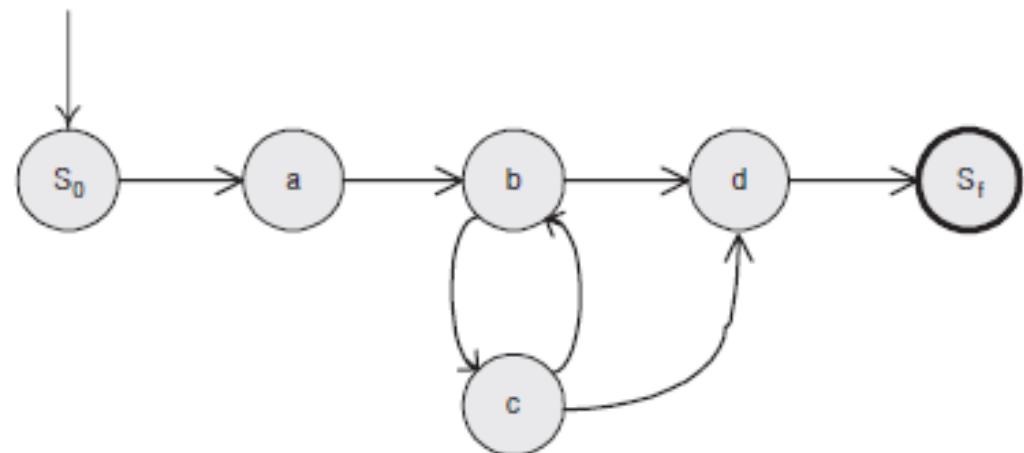
TP	TRs toured	<i>sidetrips</i>
i	A, B, D, E, F, G, I, J	C, H
ii	A, C , E, H	
iii	A, B, D, E, F, G, I, J, K , L	C, H



Some definitions..

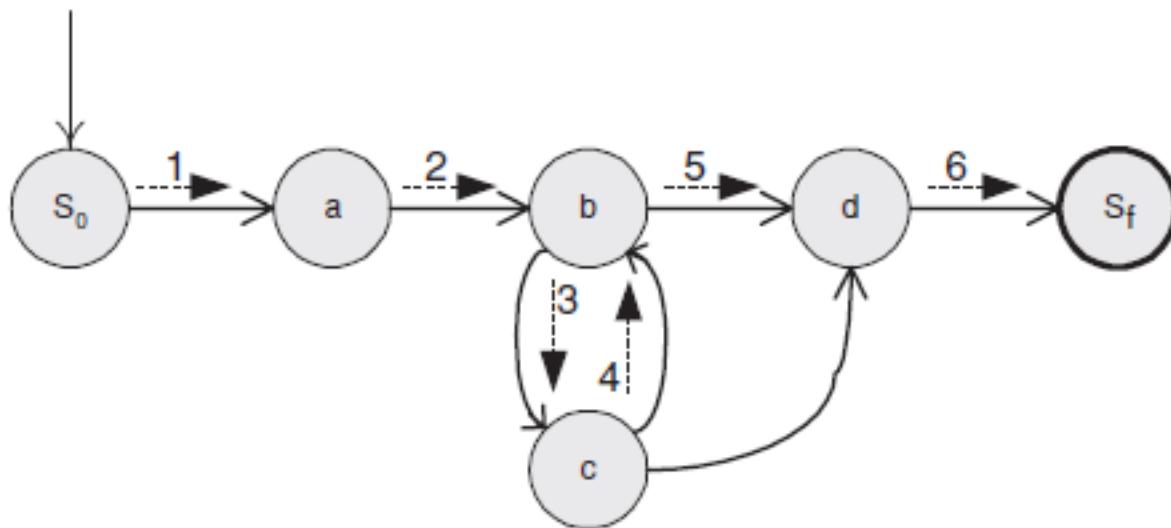
- For relaxing the test requirements

$q = [a, b, d]$ is a strict definition that does not confirm, e.g., $p = [a, b, c, b, d]$



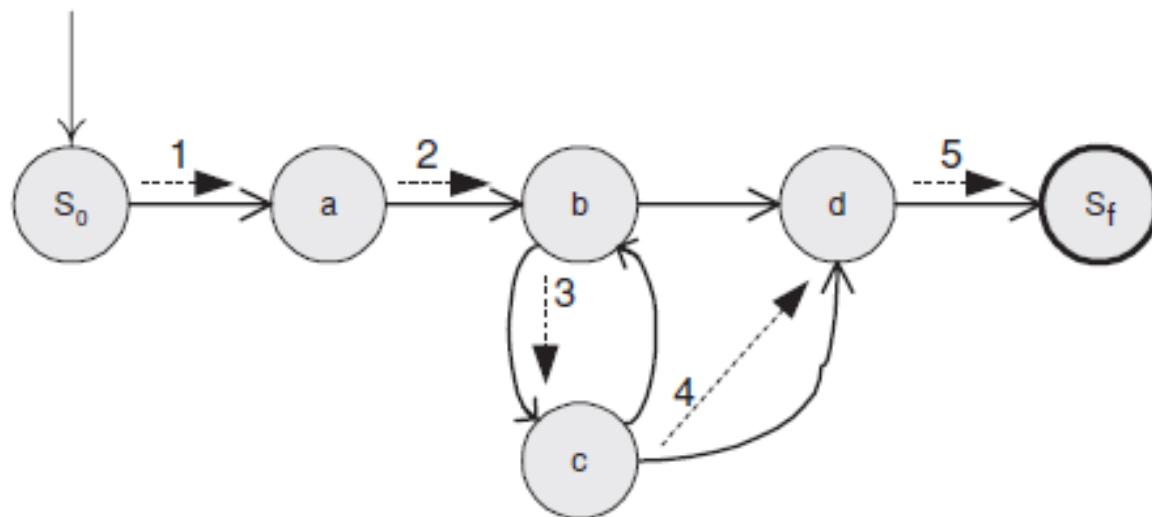
Some definitions..

- Sidetrip



Some definitions..

- Detour



Prime Path Coverage

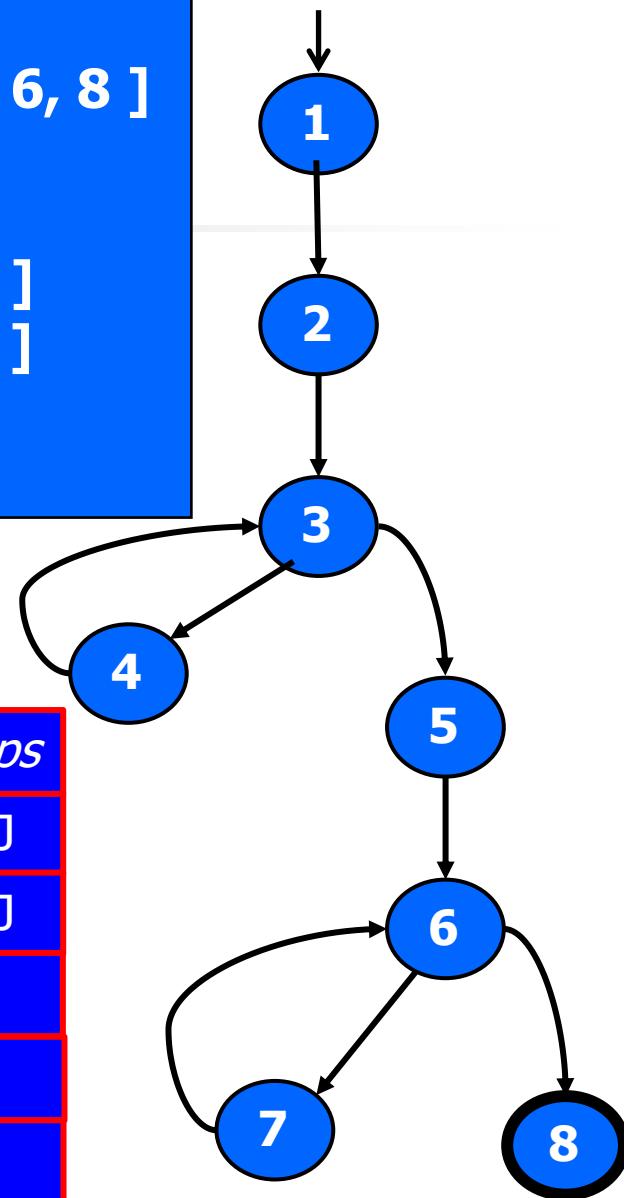
TR

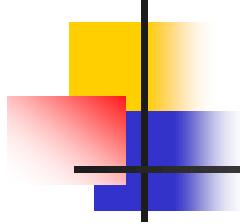
- A. [3, 4, 3]
- B. [4, 3, 4]
- C. [7, 6, 7]
- D. [7, 6, 8]
- E. [6, 7, 6]
- F. [1, 2, 3, 4]
- G. [4, 3, 5, 6, 7]
- H. [4, 3, 5, 6, 8]
- I. [1, 2, 3, 5, 6, 7]
- J. [1, 2, 3, 5, 6, 8]

Test Paths

- i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
- ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 8]
- iii. [1, 2, 3, 4, 3, 5, 6, 8]
- iv. [1, 2, 3, 5, 6, 7, 6, 8]
- v. [1, 2, 3, 5, 6, 8]

TP	TRs toured	<i>sidetrips</i>
i	A, D, E, F, G	H, I, J
ii	A, B , C , D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	





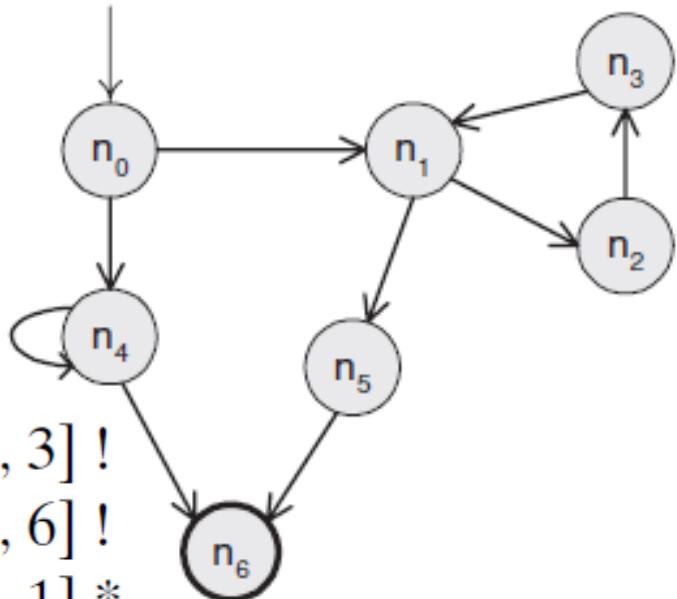
Prime Paths

- A **prime path** is a simple path that does not appear as a proper subpath of any other simple path
- Prime paths can be systematically discovered

Discovering Prime Paths

- 1) [0] **cannot be extended**
- 2) [1]
- 3) [2]
- 4) [3]
- 5) [4]
- 6) [5]
- 7) [6] !
- 8) [0, 1] **cycle**
- 9) [0, 4]
- 10) [1, 2]
- 11) [1, 5]
- 12) [2, 3]
- 13) [3, 1]
- 14) [4, 4] *
- 15) [4, 6] !
- 16) [5, 6] !
- 17) [0, 1, 2]
- 18) [0, 1, 5]
- 19) [0, 4, 6] !
- 20) [1, 2, 3]
- 21) [1, 5, 6] !
- 22) [2, 3, 1]
- 23) [3, 1, 2]
- 24) [3, 1, 5]

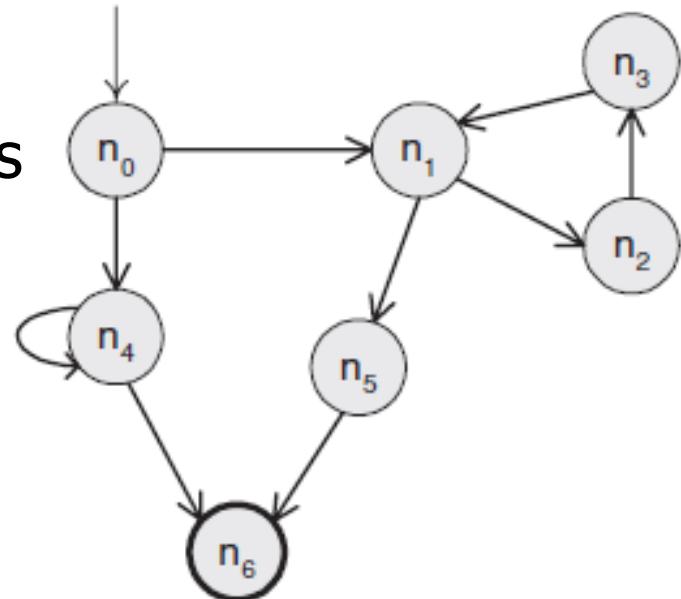
- 25) [0, 1, 2, 3] !
- 26) [0, 1, 5, 6] !
- 27) [1, 2, 3, 1] *
- 28) [2, 3, 1, 2] *
- 29) [2, 3, 1, 5]
- 30) [3, 1, 2, 3] *
- 31) [3, 1, 5, 6] !
- 32) [2, 3, 1, 5, 6] !

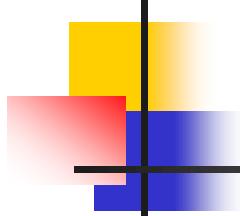


Discovering Prime Paths

- Eliminating the paths that are proper subpaths of other paths leads to 8 prime paths

- 14) [4, 4] *
- 19) [0, 4, 6] !
- 25) [0, 1, 2, 3] !
- 26) [0, 1, 5, 6] !
- 27) [1, 2, 3, 1] *
- 28) [2, 3, 1, 2] *
- 30) [3, 1, 2, 3] *
- 32) [2, 3, 1, 5, 6]!

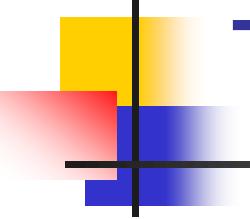




Recall: Coverage Criteria Based on CFG

and graphs in general..

- Node Coverage (NC)
- Edge Coverage (EC)
- Edge-Pair Coverage (EPC)
 - each reachable path of length up to 2
- Prime Path Coverage (PPC)
- Complete Path Coverage (CPC)

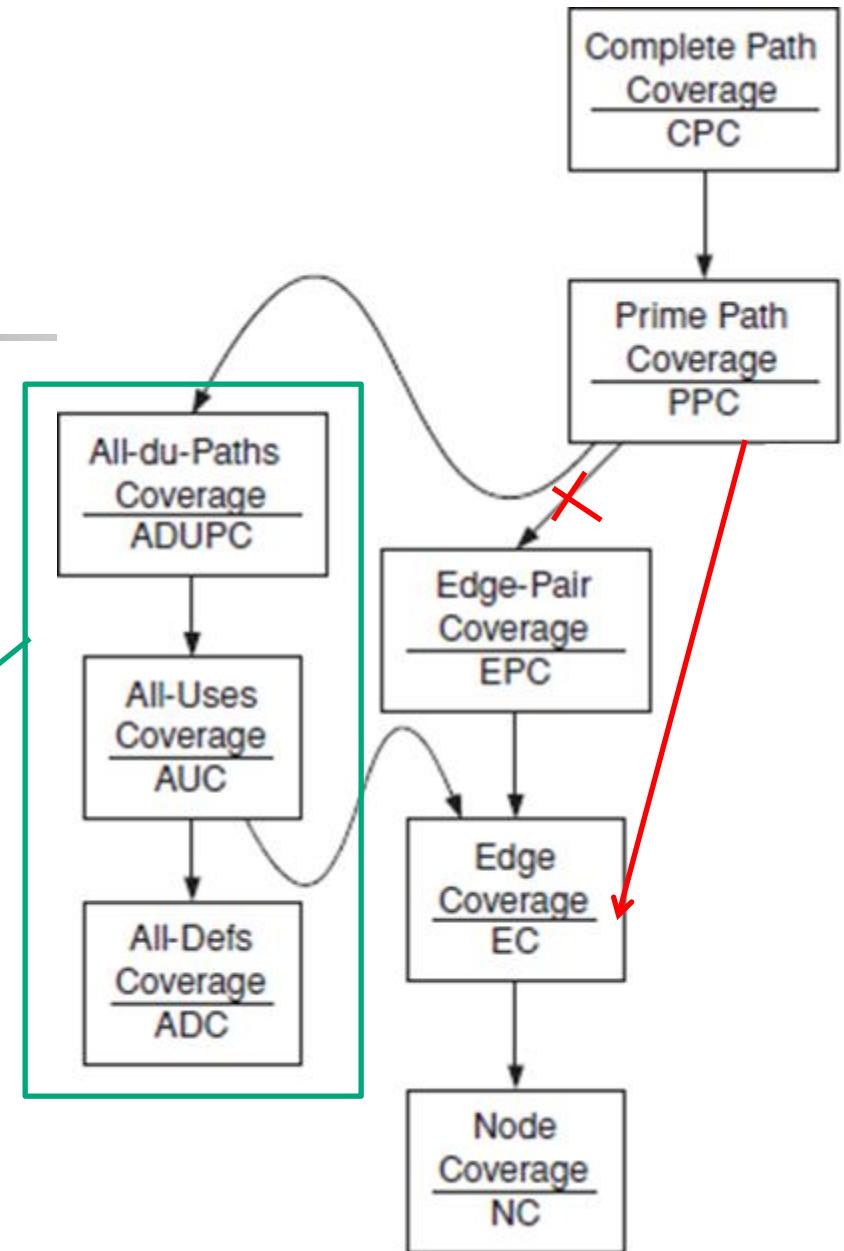


The *subsumes* relation

- *Test adequacy criterion A subsumes test adequacy criterion B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P*
- Example:
 - Exercising all program branches (branch coverage) *subsumes* exercising all program statements
- A common analytical comparison of closely related criteria
 - Useful for working from easier to harder levels of coverage, but not a direct indication of quality

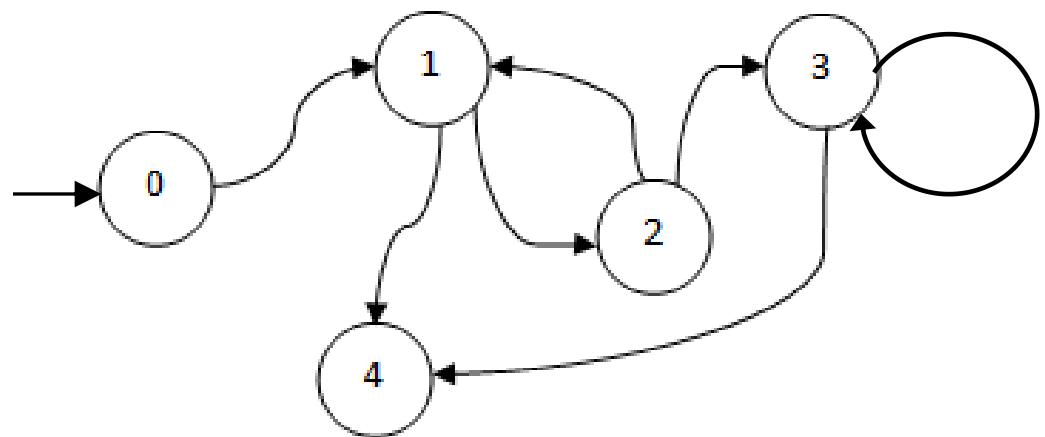
Subsumes Relations

to be discussed later



Counter Example

- Not possible to satisfy the test requirement [2,3,3] for edge-pair coverage with any prime path

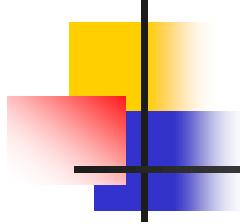


CS 575

Software Testing and Analysis

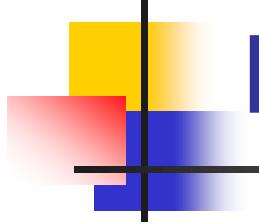
Test-Driven Development





Motivation

- *Common sense:*
 - **a good testing methodology** is necessary for the development of reliable software
- *Common practice:*
 - ad-hoc, unstructured
 - not repeatable
 - late



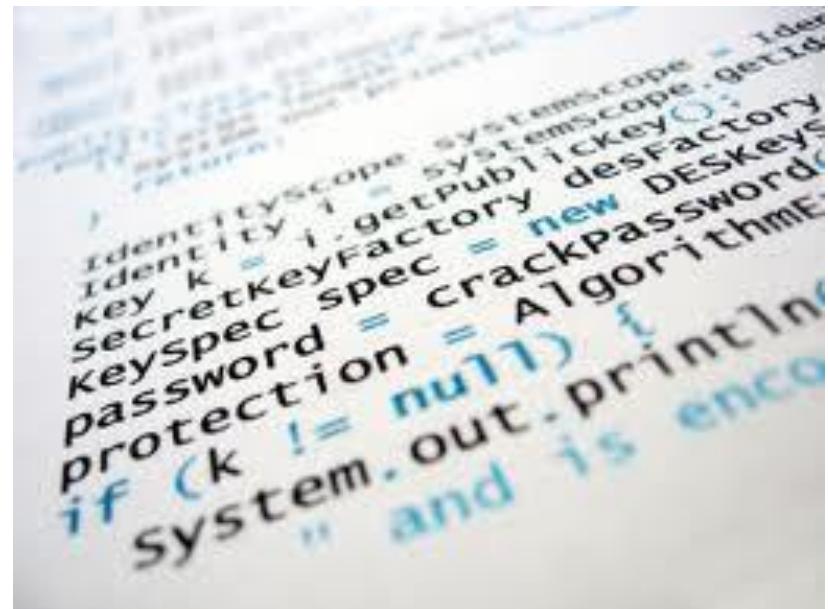
Motivation

- All the code must be tested to ensure reliability



Test-Driven Development (TDD)

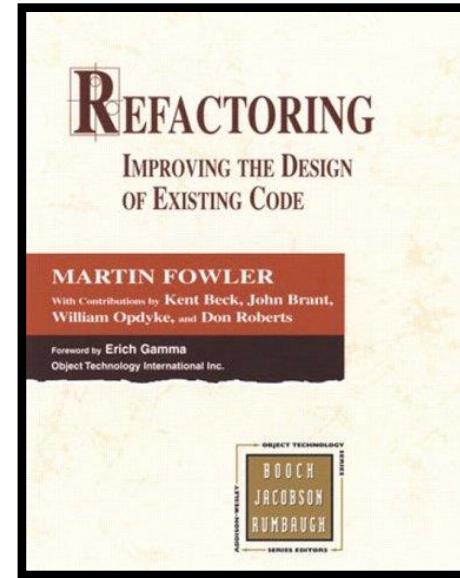
- A process with tiny steps
- main focus on
testing and **refactoring**
- Refactoring: restructuring code without changing its external behaviour ...



Refactoring Operations

- Extract Method
- Extract Class
- Extract Superclass
- Encapsulate Field
- Decompose Conditional
- ...

- Goal: increasing readability and maintainability of the code

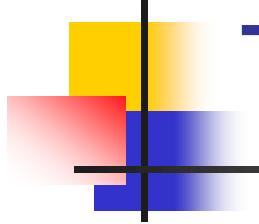


Refactoring: Improving the Design of Existing Code
Martin Fowler
Addison-Wesley, 1999
ISBN 0-201-48567-2



Origins..

- Devised in the context Agile methodologies
 - but can also be applied by itself, without Agile methods
- Agile development
 - eXtreme Programming (XP)
 - TDD
- Other methodologies, e.g., waterfall, spiral, etc.



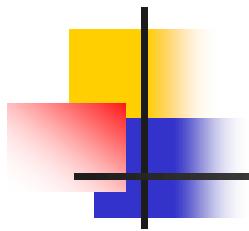
TDD Focus

- Implementation, code-level
 - (i.e., not architecture design, requirements engineering, etc.)
 - Developer = programmer
- Unit testing
 - (i.e., not integration testing, acceptance testing, etc.)
 - Testing internals of a class
 - Black-box testing for objects

Tool-support for TDD

- There exists frameworks for automating the test execution
 - a must for TDD!
- Examples:
 - JUnit, CppUnit, Nunit, DUnit, VBUnit, RUnit, PyUnit, Sunit, HtmlUnit, ...
 - More listed at www.xprogramming.com





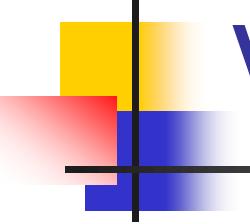
Basic principles

- **Before** writing the code,
think about what it will do
- Write tests for methods that
do not even exist yet



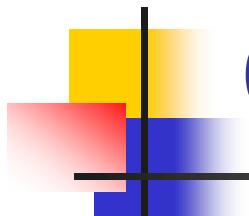
TDD Stages





Vital Properties

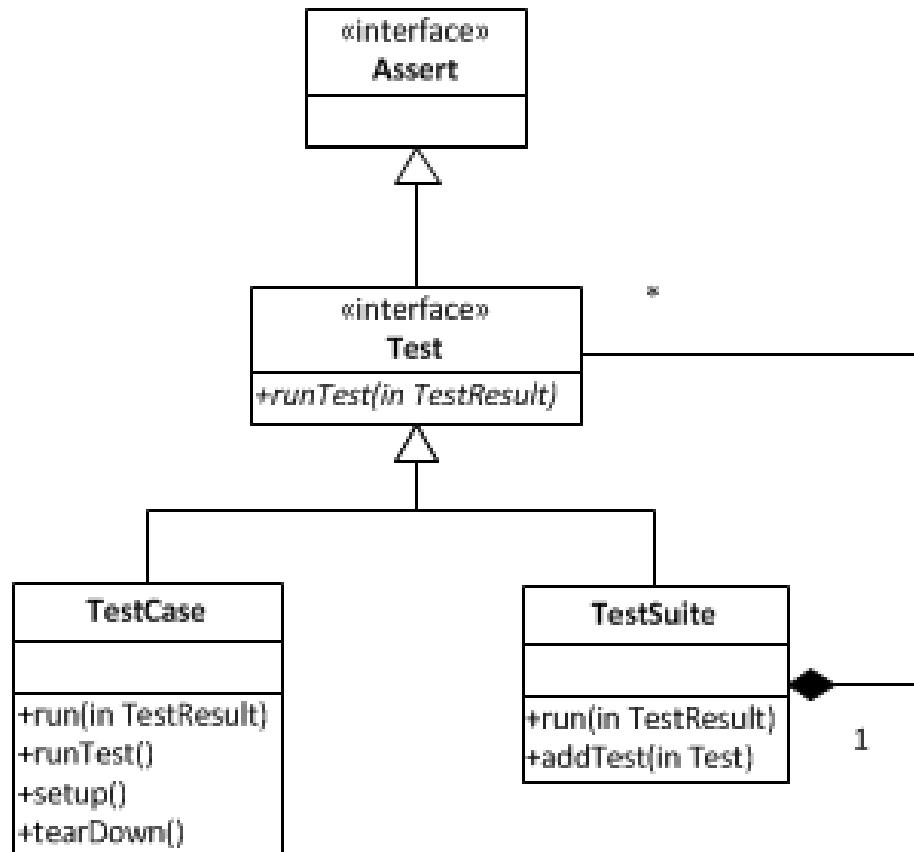
- Only **small** refactorings
 - makes it less likely to go wrong
 - system is kept fully working after each step
- Test: not an activity but a piece of **code**
 - not something you do, but something you write
- Test execution is **automated**
 - repeatedly executed after very small changes



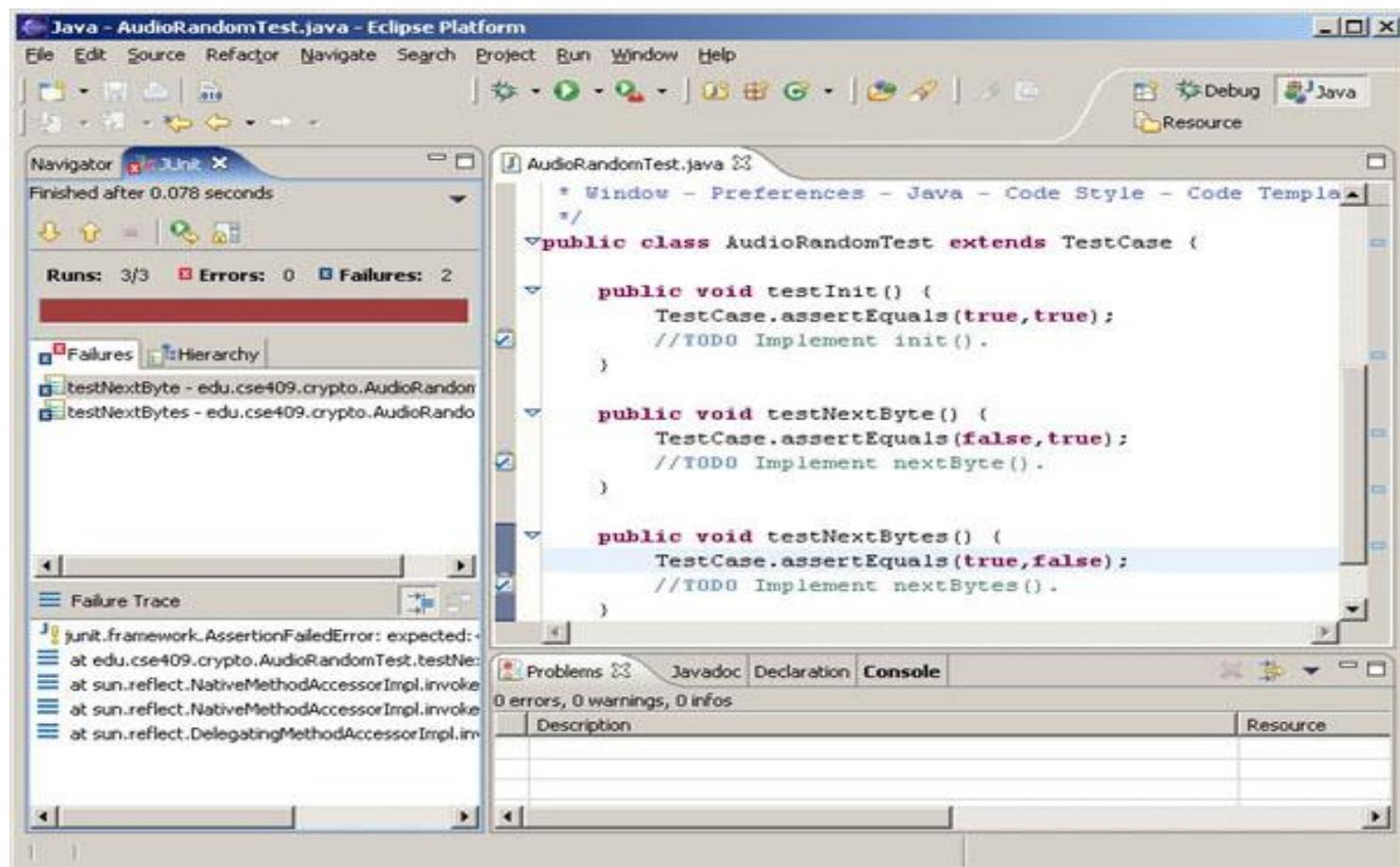
Case: Java + Eclipse + JUnit

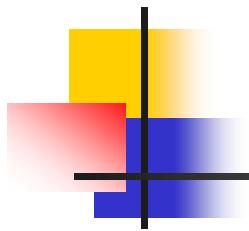


Junit Framework



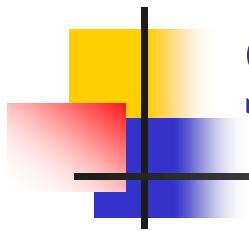
Junit for Eclipse





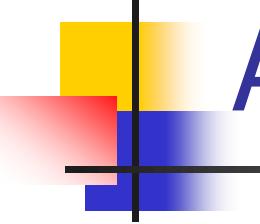
Demo..





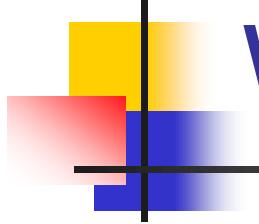
Structure of a Test Case class

- `setUp()` initialize variables, claim resources
- `tearDown()` release the resources
- `run()` run a *test case/suite*
- `testCase()` a test case



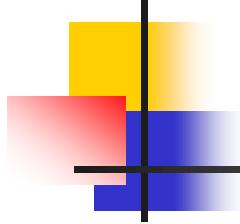
Assert Statements

- assertEquals(expected, actual)
- assertEquals(message, expected, actual)
- assertEquals(expected, actual, delta)
- assertEquals(message, expected, actual, delta)
- assertFalse(condition)
- assertFalse(message, condition)
- Assert(Not)Null(object)
- Assert(Not)Null(message, object)
- Assert(Not)Same(expected, actual)
- Assert(Not)Same(message, expected, actual)
- assertTrue(condition)
- assertTrue(message, condition)



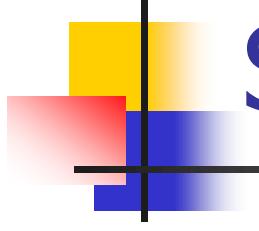
Why TDD?

- Programmers do not like testing
 - considered to be a boring task
- TDD **encourages** and **facilitates** programmers to maintain repeatable tests
 - Tests live alongside the code
 - Test execution is automated (push-button)
 - Test after every single change increases **confidence**
- ...



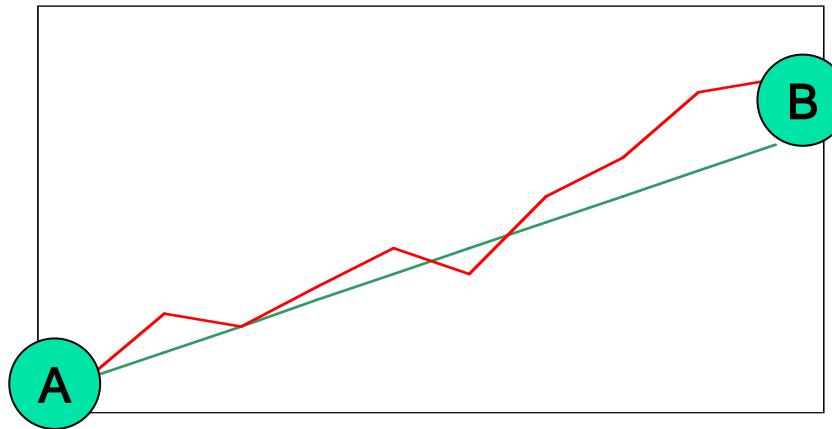
TDD ..

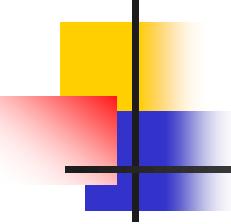
- shortens the programming feedback loop
 - in the order of minutes
- provides detailed specification (tests)
- promotes the development of high-quality code
- provides concrete evidence that code works
 - boosts confidence
- supports evolutionary development



Small steps

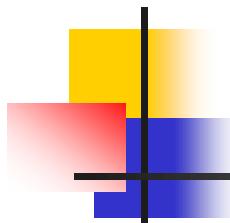
- TDD promotes “small steps”
 - Not to diverge too much from the path to the destination





Final notes

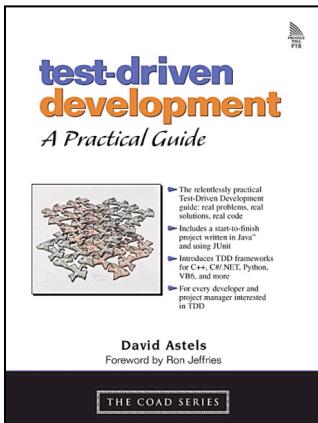
- TDD increases development speed
 - due to less time spent for debugging
- TDD decreases maintenance costs
 - due to increased modularity as a result of refactoring
- TDD does not replace traditional testing
 - just proposes a way of working
- TDD might require supporting techniques
 - creating Mock objects (stubs) and views
 - substituting visual elements during testing



Summary

- No code without tests
 - Tests verify the code, acting as documentation
- Tests dictate the code
 - Let the design emerge
- Testing and refactoring go hand-in-hand
 - Ensuring clean, modular code
- Elementary increments
 - Small, tiny, safe steps

Some Resources on TDD..

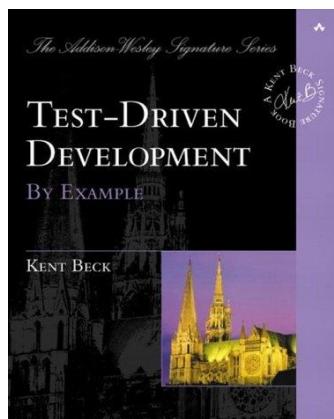


test-driven development: A Practical Guide

Dave Astels

Pearson Education, 2003

ISBN 0-13-101649-0

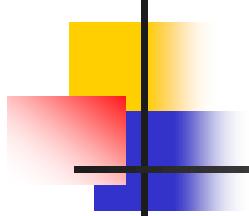


Test-Driven Development: By Example

Kent Beck

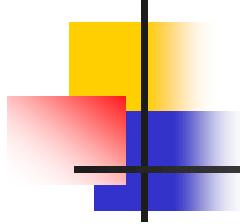
Addison-Wesley, 2003

ISBN 0-321-14653-0

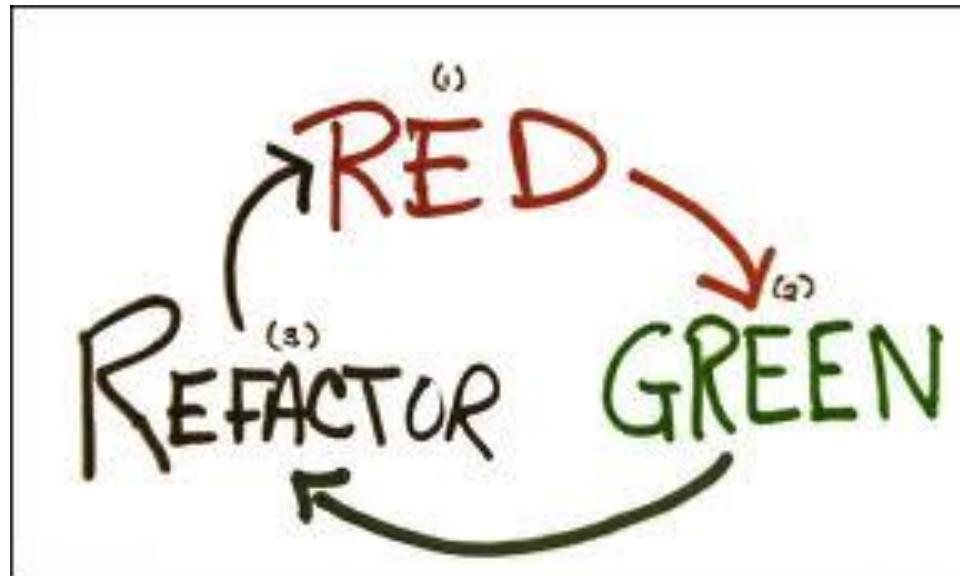


Introducing TDD in an organization...

- Start small and simple
- Adopt TDD
 - for new tasks
 - for the code that needs to be modified in small pieces
- Provide proof-of-concept results



Questions?



CS 575

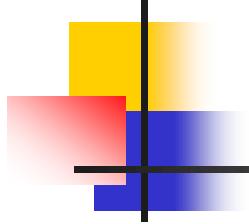
Software Testing and Analysis

Control Flow Data Analysis



(c) Slides partially adopted from the book/slides of

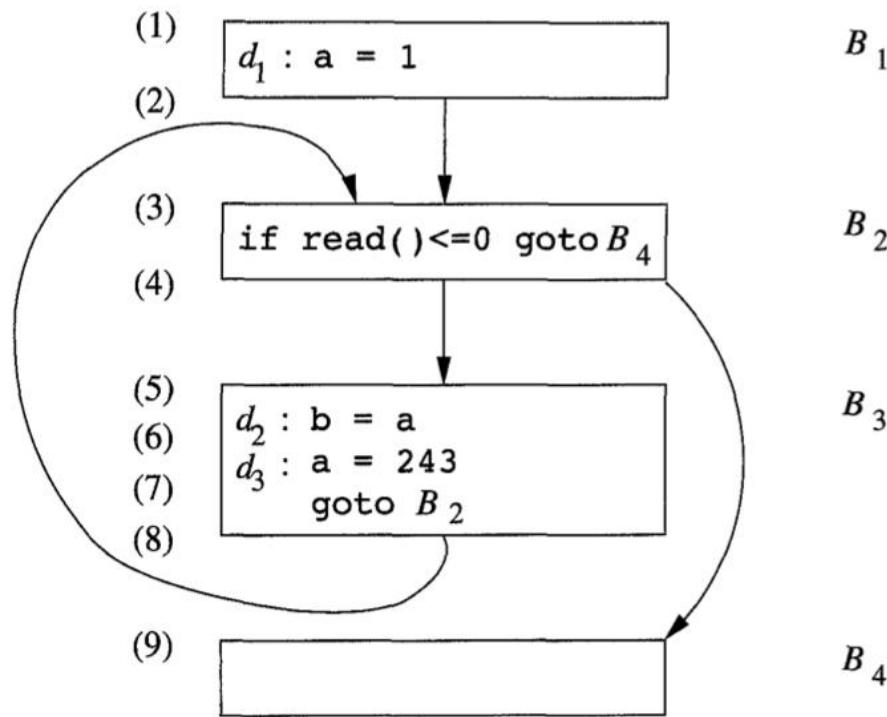
- P. Amman & J. Offut
- M. Pezze and M. Young
- Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman
- K.D. Cooper and L. Torczon
- B. Aktemur



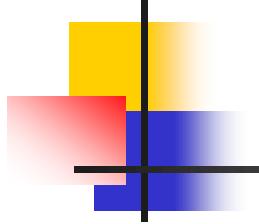
Software Testing and Analysis

- Revealing bugs
 - Null pointers, memory leaks, uninitialized variables, race conditions, buffer overflows..
- Code Optimization
- Defining Coverage Criteria
 - Based on data dependence:
 - Where does this value of x come from?
 - What would be affected by changing this?
 - ...

Data Flow Analysis: Example

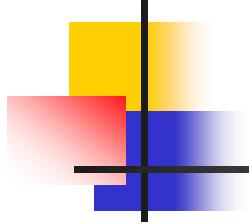


- The first time program point (5) is executed, the value of a is 1 due to definition d_1 .
- In subsequent iterations, d_3 reaches point (5) and the value of a is 243.
- At point (5), the value of a is one of $\{1, 243\}$.
- It may be defined by one of $\{d_1, d_3\}$.



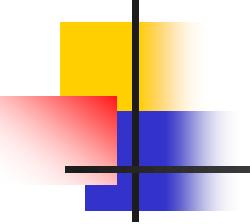
Def-Use Pairs

- A **def-use (du) pair** associates
 - a point in a program where a value is produced with
 - a point where it is used
- Extensively used for dataflow analysis in compilers
 - i.e., reaching definitions



Def (Definition)

- Where a variable gets a value
 - Variable declaration (often the special value “uninitialized”)
 - Variable initialization
 - Assignment
 - Values received by a parameter

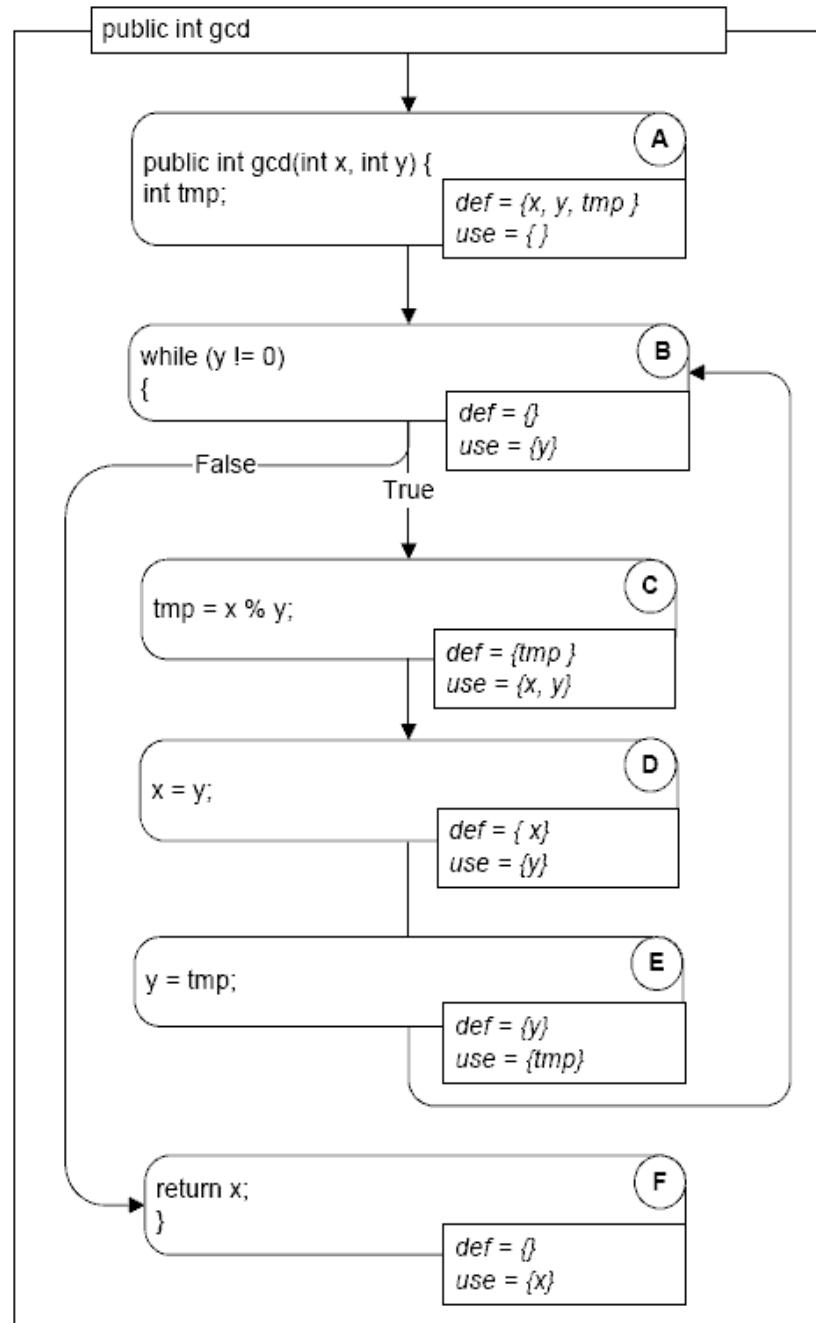


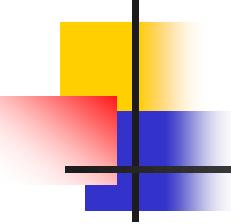
Use

- Extraction of a value from a variable
 - Expressions
 - Conditional statements
 - Parameter passing
 - Returns

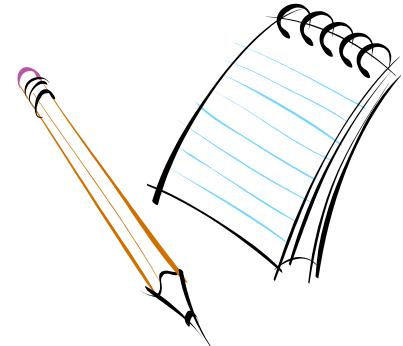
Def-Use Sets: Example

```
/** Euclid's algorithm */
public class GCD
{
    public int gcd(int x, int y) {
        int tmp;          // A: def x, y, tmp
        while (y != 0) {  // B: use y
            tmp = x % y; // C: def tmp; use x, y
            x = y;         // D: def x; use y
            y = tmp;        // E: def y; use tmp
        }
        return x;          // F: use x
    }
}
```





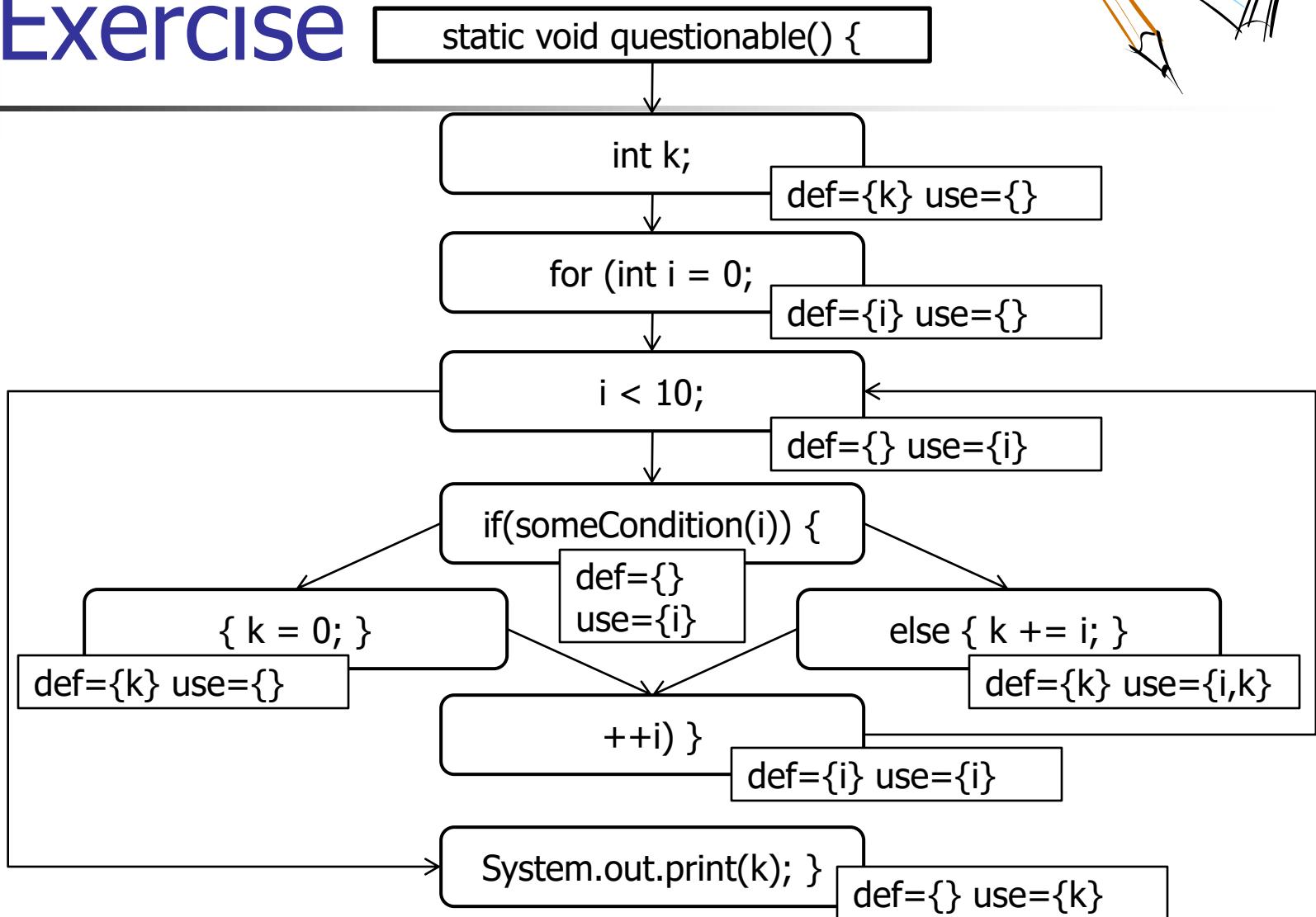
Exercise



- Draw the control flow graph for the method
- Annotate the nodes with *def* and *use* sets

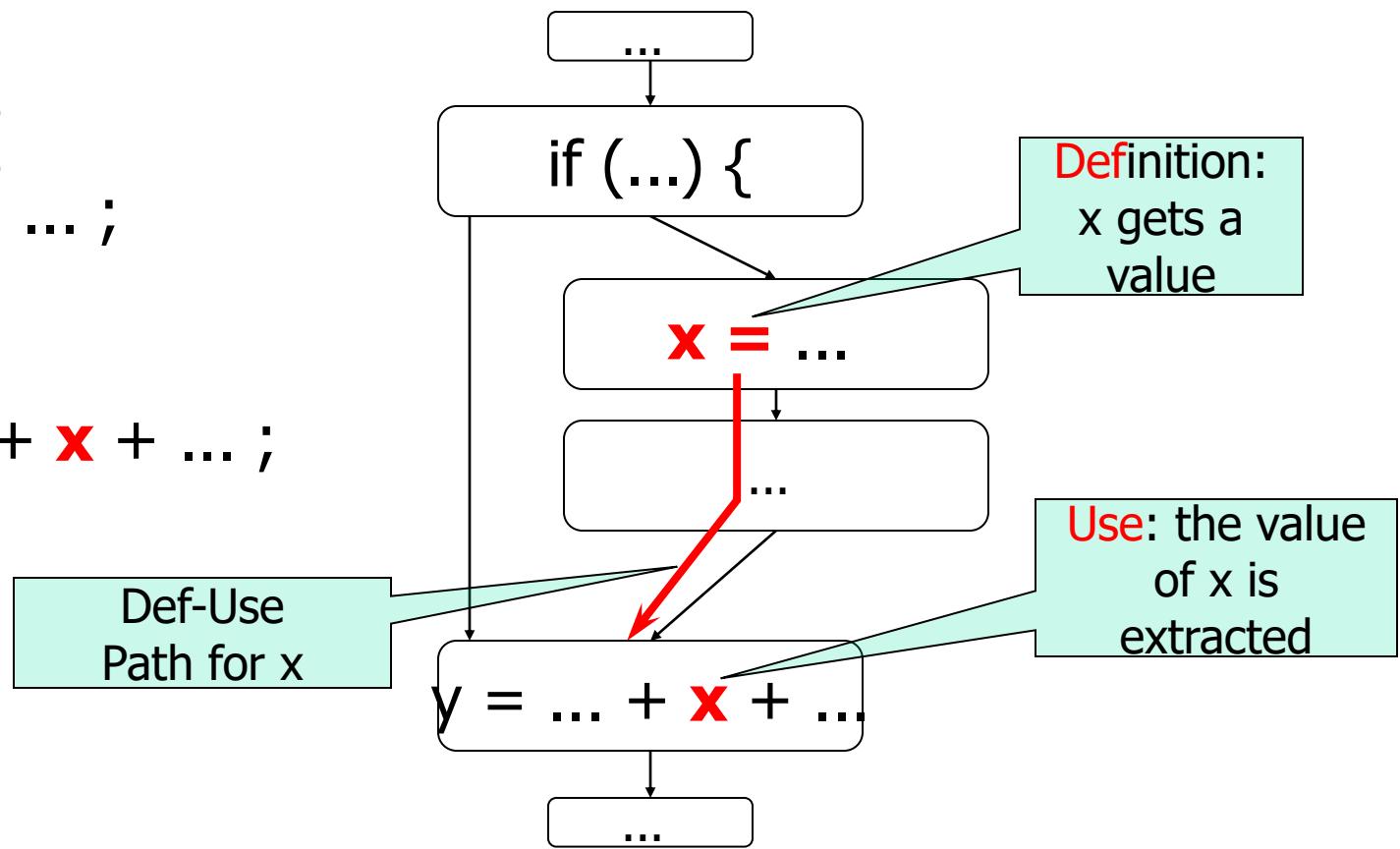
```
static void questionable() {  
    int k;  
    for (int i = 0; i < 10; ++i) {  
        if (someCondition(i)) {  
            k = 0;  
        } else {  
            k += i;  
        }  
    }  
    System.out.println(k);  
}
```

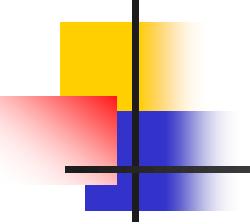
Exercise



Def-Use Pairs: Example

```
...  
if (...) {  
    x = ... ;  
}  
y = ... + x + ... ;
```





Killing (Overwriting) Definitions

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without* another definition of the variable in-between
 - in case of any overwriting, the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

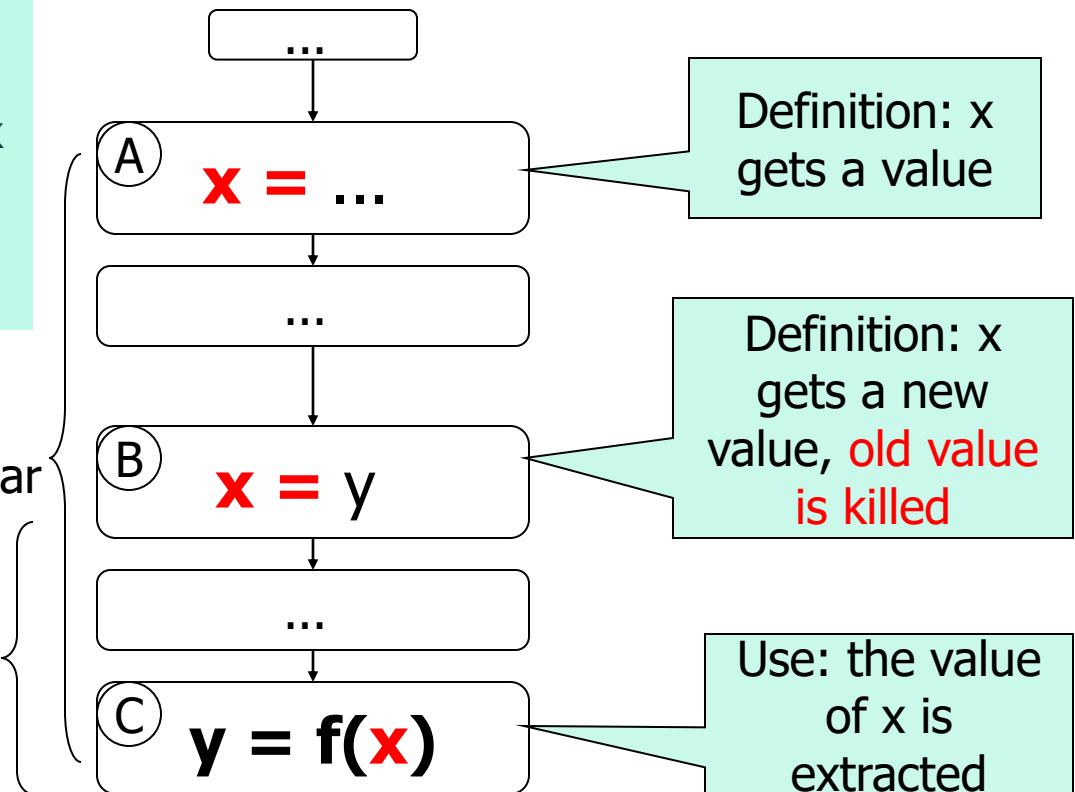
* *In fact, sometimes it is impossible to know for sure whether two definitions affect the same variable or storage location.*

Killing (Overwriting) Definitions: Example

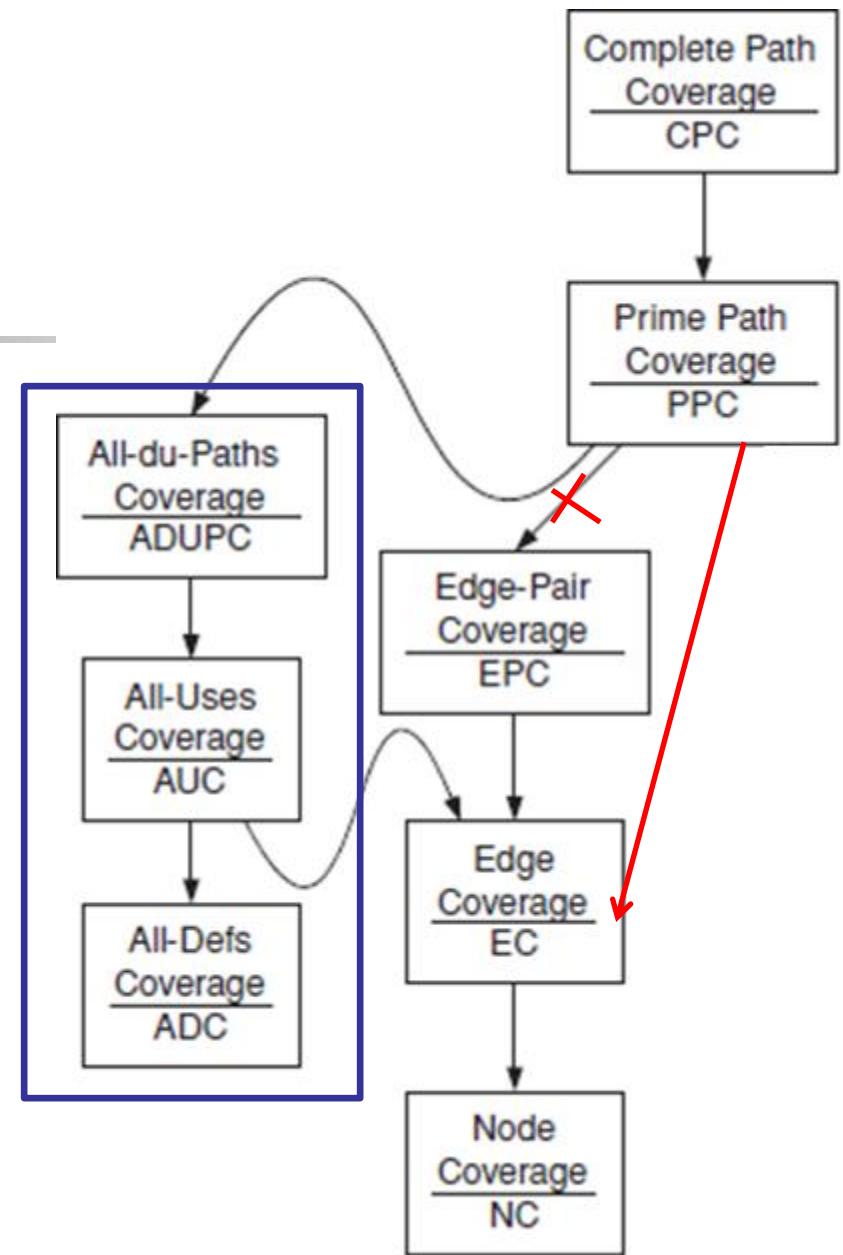
```
x = ... // A: def x  
q = ...  
x = y; // B: kill x, def x  
z = ...  
y = f(x); // C: use x
```

Path A..C is
not definition-clear

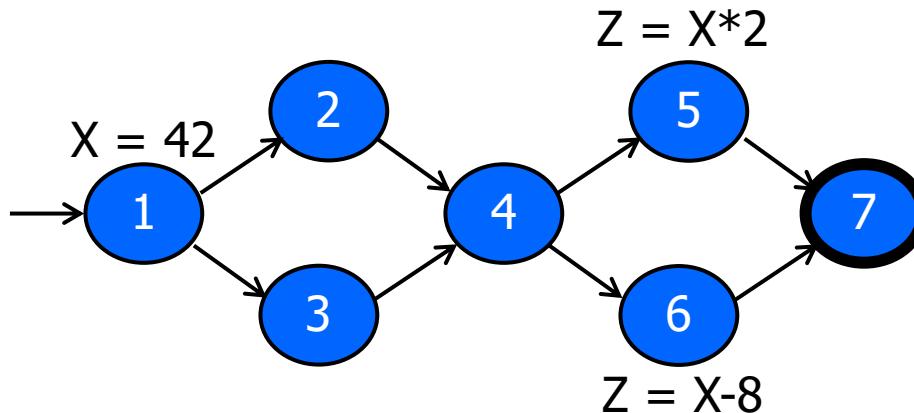
Path B..C is
definition-clear



Recall: Coverage Criteria Based on Data Flow



Data Flow Testing Example



All-defs for X

[1, 2, 4, 5]

All-uses for X

[1, 2, 4, 5]

[1, 2, 4, 6]

All-du-paths for X

[1, 2, 4, 5]

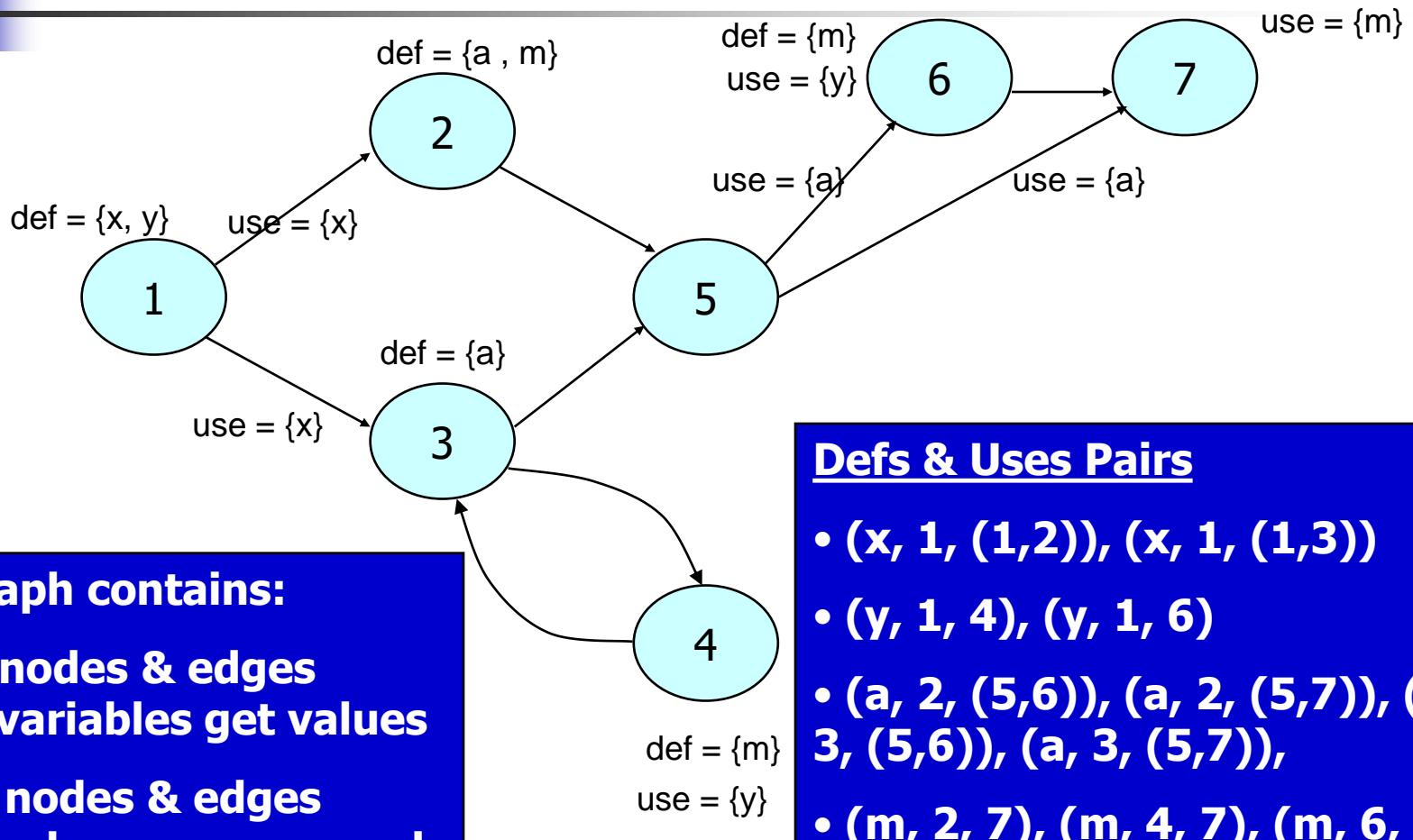
[1, 3, 4, 5]

[1, 2, 4, 6]

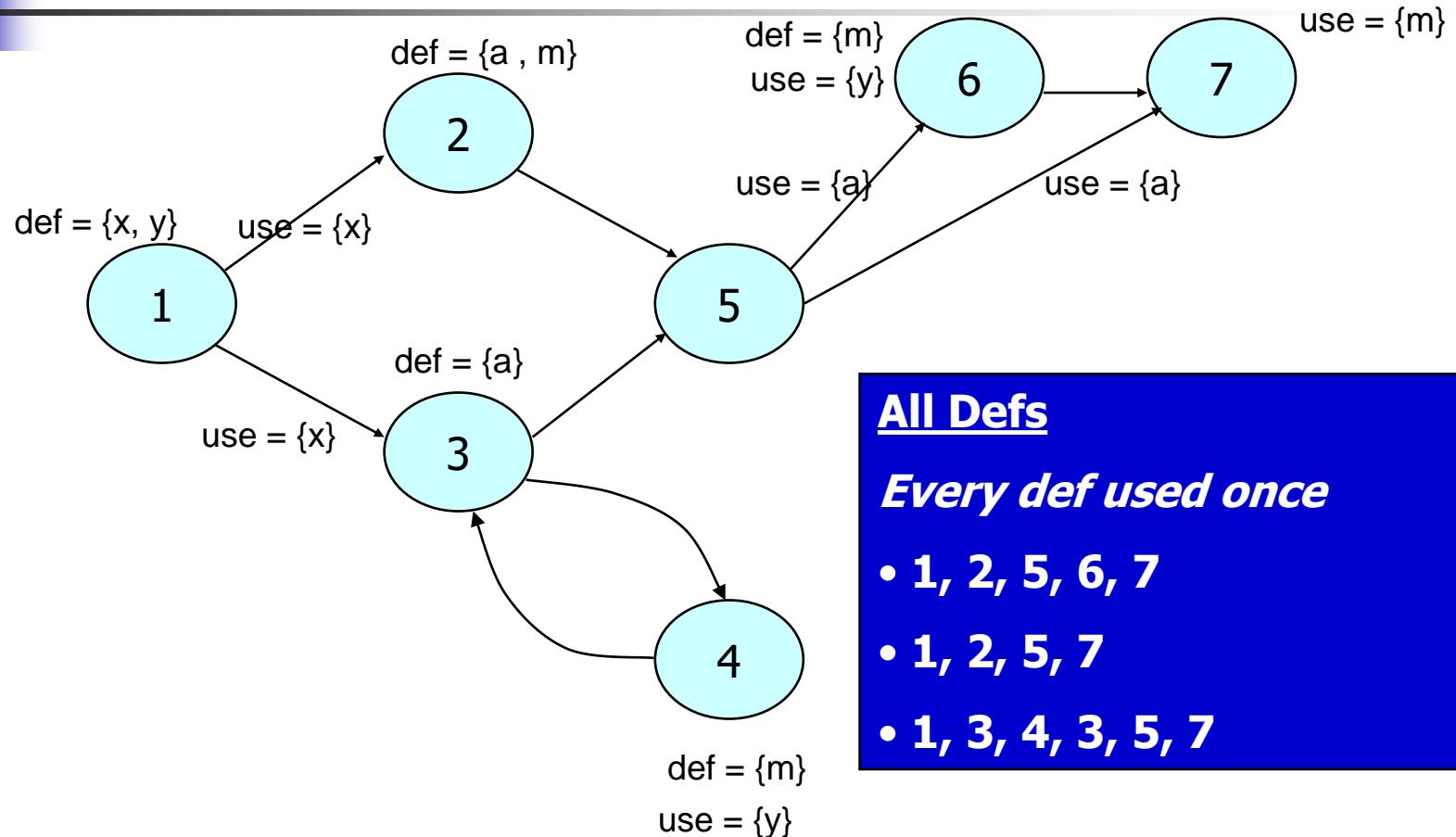
[1, 3, 4, 6]

Defs & Uses Pairs

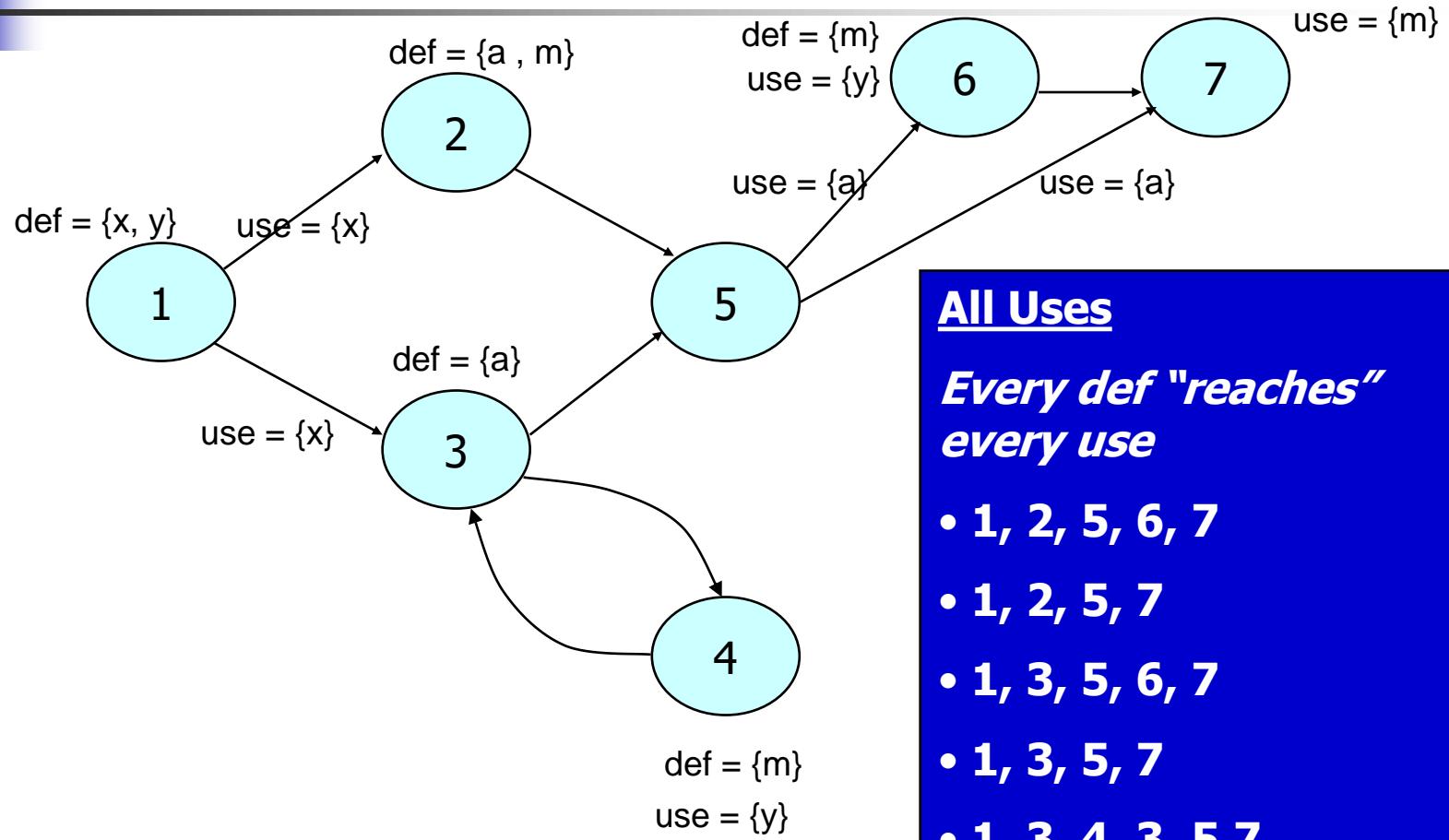
Coverage Criteria

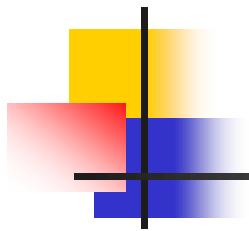


All-Defs Coverage Criteria



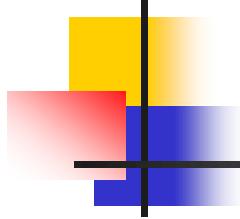
All-Uses Coverage Criteria



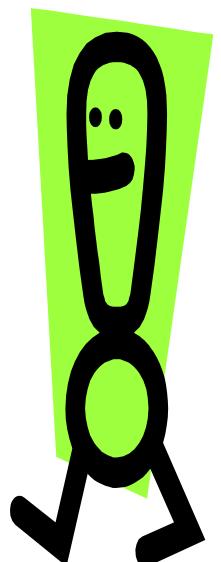


Calculating def-use pairs

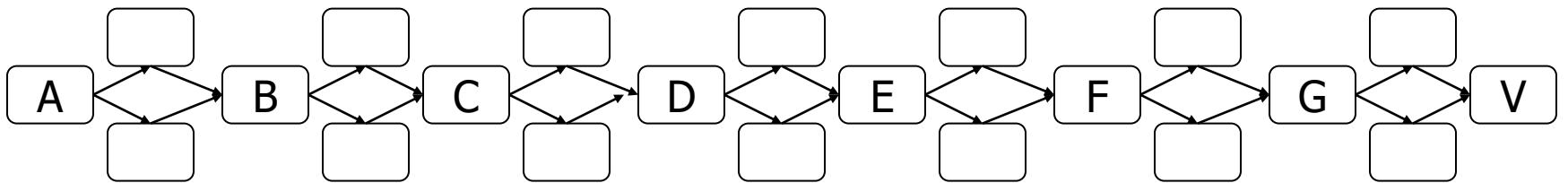
- Defined in terms of paths in the CFG:
 - There is an association (d,u) between a definition of variable v at d and a use of variable v at u if and only if
 - there is at least one control flow path from d to u
 - with no intervening definition of v
 - v_d **reaches** u (v_d is a **reaching definition** at u).
 - If a control flow path passes through another definition e of the same variable v , v_e **kills** v_d at that point.



Calculating def-use pairs

- 
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
 - Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

Example: Exponential number of paths (even without loops)



2 paths from A to B

4 from A to C

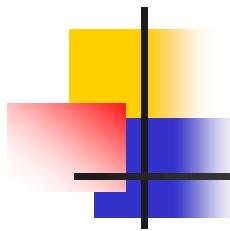
8 from A to D

16 from A to E

...

128 paths from A to V

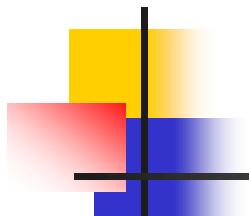
Tracing each path is not efficient, and we can do much better.



An Iterative Algorithm for Computing Reaching Definitions

- Based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node

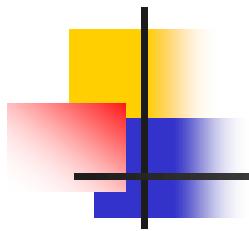




Propagation of Information Among Nodes of a CFG

- Suppose we are calculating the reaching definitions of node n , and there is an edge (p, n) from an immediate predecessor node p
 - If the predecessor node p can assign a value to variable v , then the definition v_p reaches n . We say the definition v_p is generated at p .
 - If a definition v_p of variable v reaches a predecessor node p , and if v is not redefined at that node, then the definition is propagated on from p to n .





Equations of node E ($y = \text{tmp}$)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
public class GCD {  
    public int gcd(int x, int y) {  
        int tmp;          // A: def x, y, tmp  
        while (y != 0) {  // B: use y  
            tmp = x % y; // C: def tmp; use x, y  
            x = y;         // D: def x; use y  
            y = tmp;        // E: def y; use tmp  
        }  
        return x;         // F: use x  
    }  
}
```

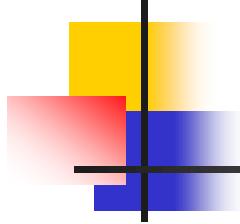
- $\text{Reach}(E) = \text{ReachOut}(D)$
- $\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$

Equations of node B (while ($y \neq 0$))

*This line has two predecessors:
Before the loop,
end of the loop*

```
public class GCD {  
    public int gcd(int x, int y) {  
        int tmp; // A: def x, y, tmp  
        while (y != 0) { // B: use y  
            tmp = x % y; // C: def tmp; use x, y  
            x = y; // D: def x; use y  
            y = tmp; // E: def y; use tmp  
        }  
        return x; // F: use x  
    }  
}
```

- $\text{Reach}(B) = \text{ReachOut}(A) \cup \text{ReachOut}(E)$
- $\text{ReachOut}(A) = \text{gen}(A) = \{x_A, y_A, \text{tmp}_A\}$
- $\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$



General equations for Reach analysis

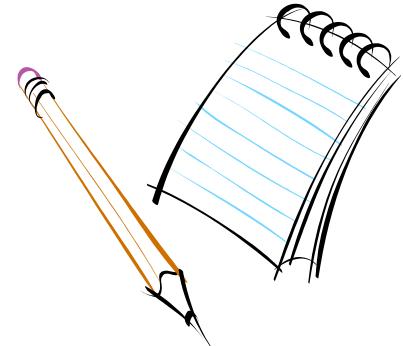
$$\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$$

$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

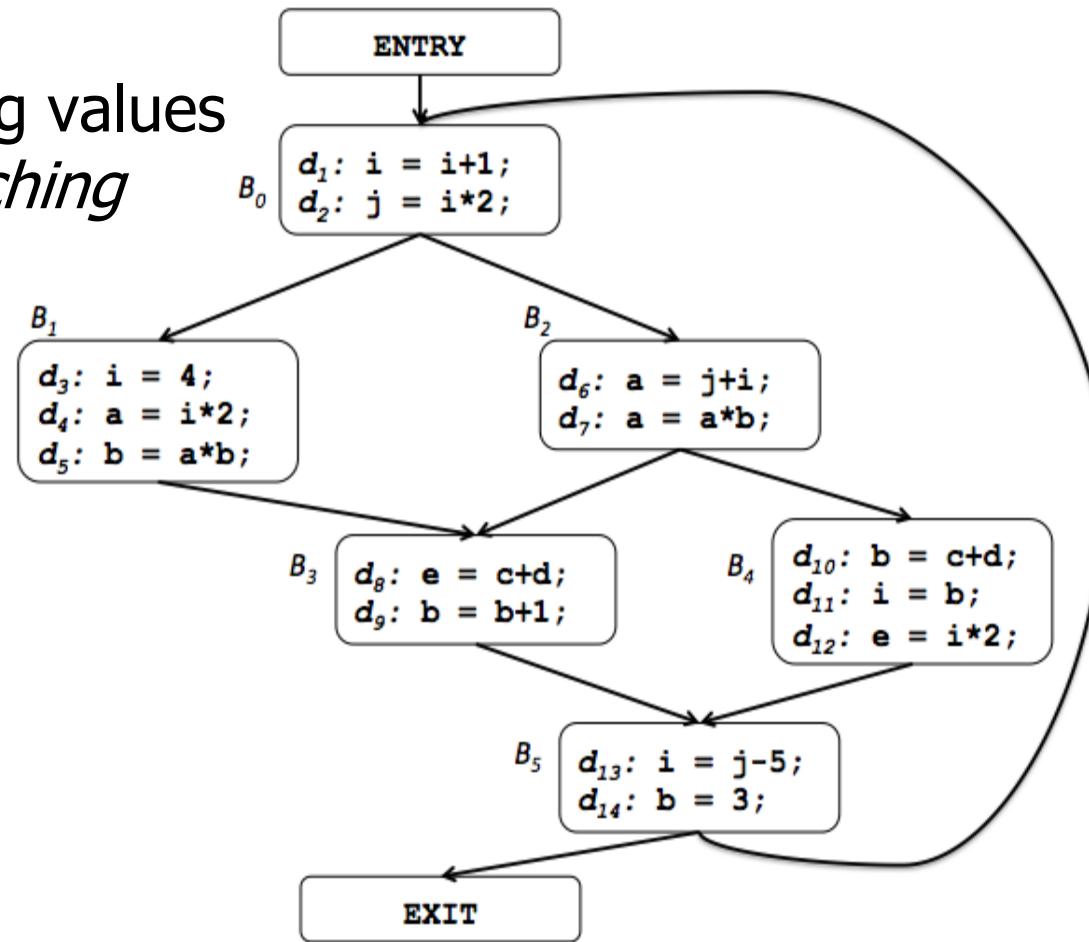
$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$

$\text{kill}(n) = \{ v_x \mid v \text{ is (re)defined or modified at } x, x \neq n \}$

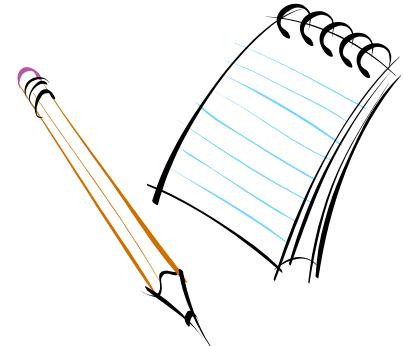
Exercise



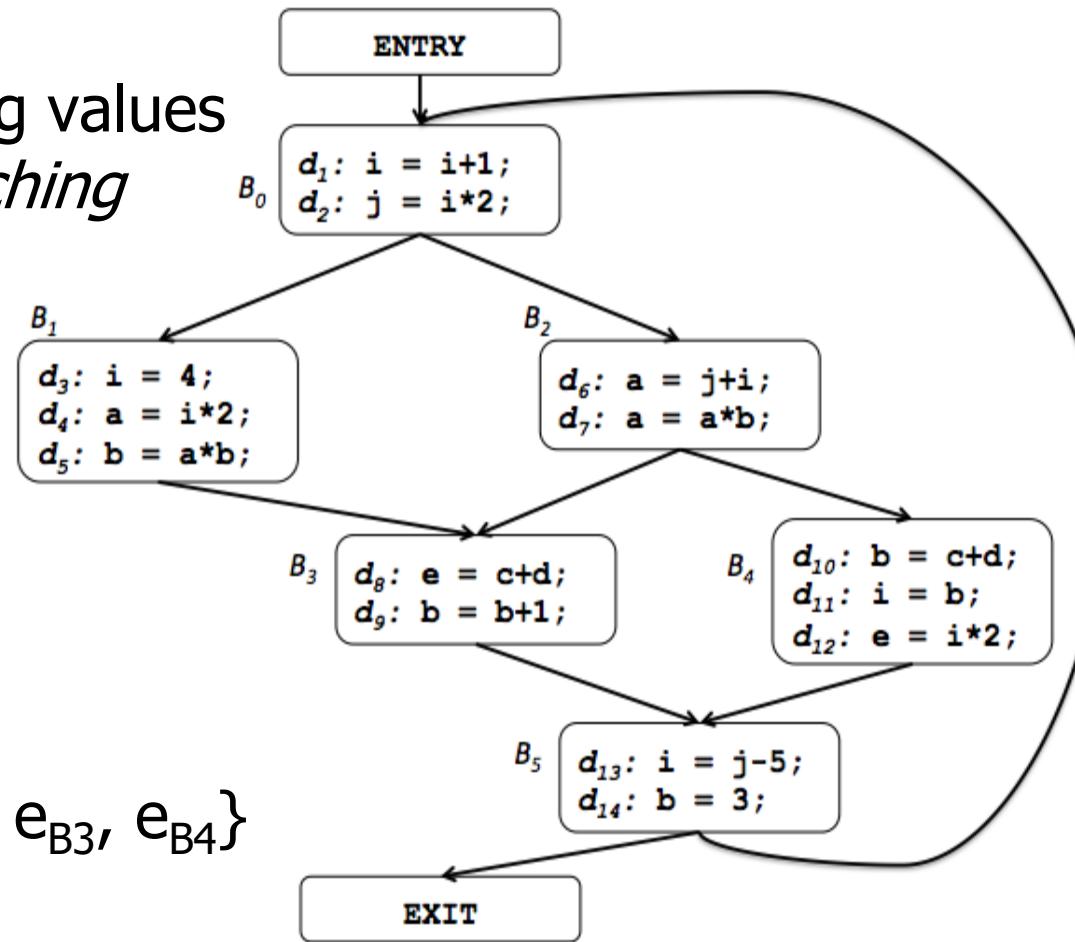
- What are the following values according to the *Reaching Definitions* analysis?
 - $\text{gen}(B_0)$
 - $\text{kill}(B_0)$
 - $\text{ReachOut}(B_0)$



Exercise



- What are the following values according to the *Reaching Definitions* analysis?
 - $\text{gen}(B_0) = \{i_{B0}, j_{B0}\}$
 - $\text{kill}(B_0) = \{i_{B1}, i_{B4}, i_{B5}\}$
 - $\text{ReachOut}(B_0) = \{i_{B0}, j_{B0}, a_{B1}, a_{B2}, b_{B5}, e_{B3}, e_{B4}\}$



Avail equations (available expressions)

$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

"all paths"
rather than
"any path"

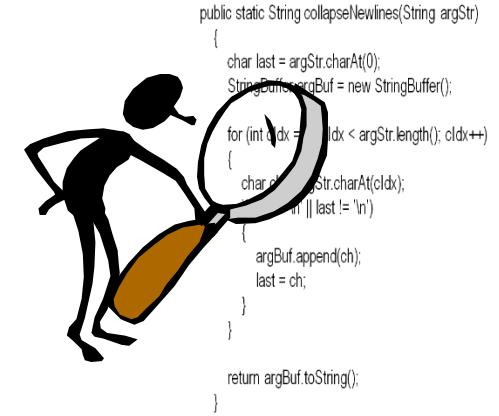
$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$\text{gen}(n) = \{ \text{exp} \mid \text{exp is computed at } n \}$

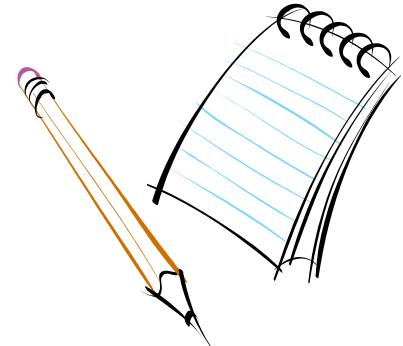
$\text{kill}(n) = \{ \text{exp} \mid \text{exp has variables assigned at } n \}$

Possible Application of Avail analysis

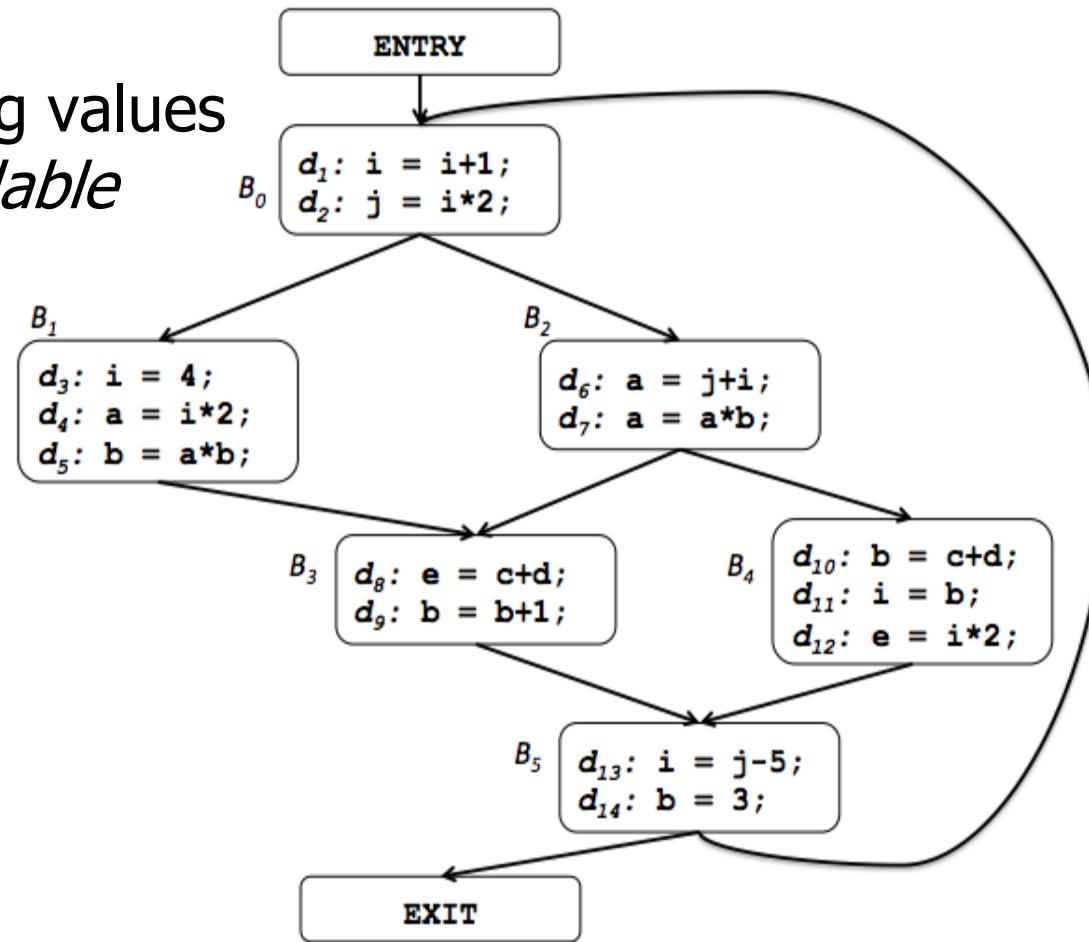
- Compiler Optimization
 - If an expression is available, do not recompute it
- Enforcing Variable Initialization
 - Java requires a variable to be initialized before use on all execution paths
 - **kill** sets are empty since there is no way to “uninitialize” a variable in Java



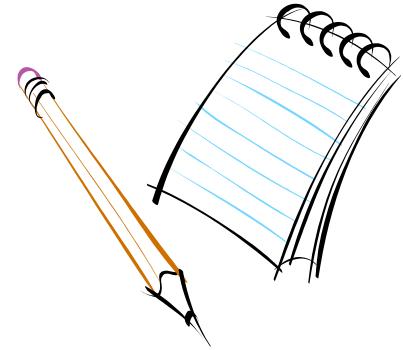
Exercise



- What are the following values according to the *Available Expressions* analysis?
 - $\text{gen}(B_0)$
 - $\text{kill}(B_0)$
 - $\text{Avail}(B_5)$

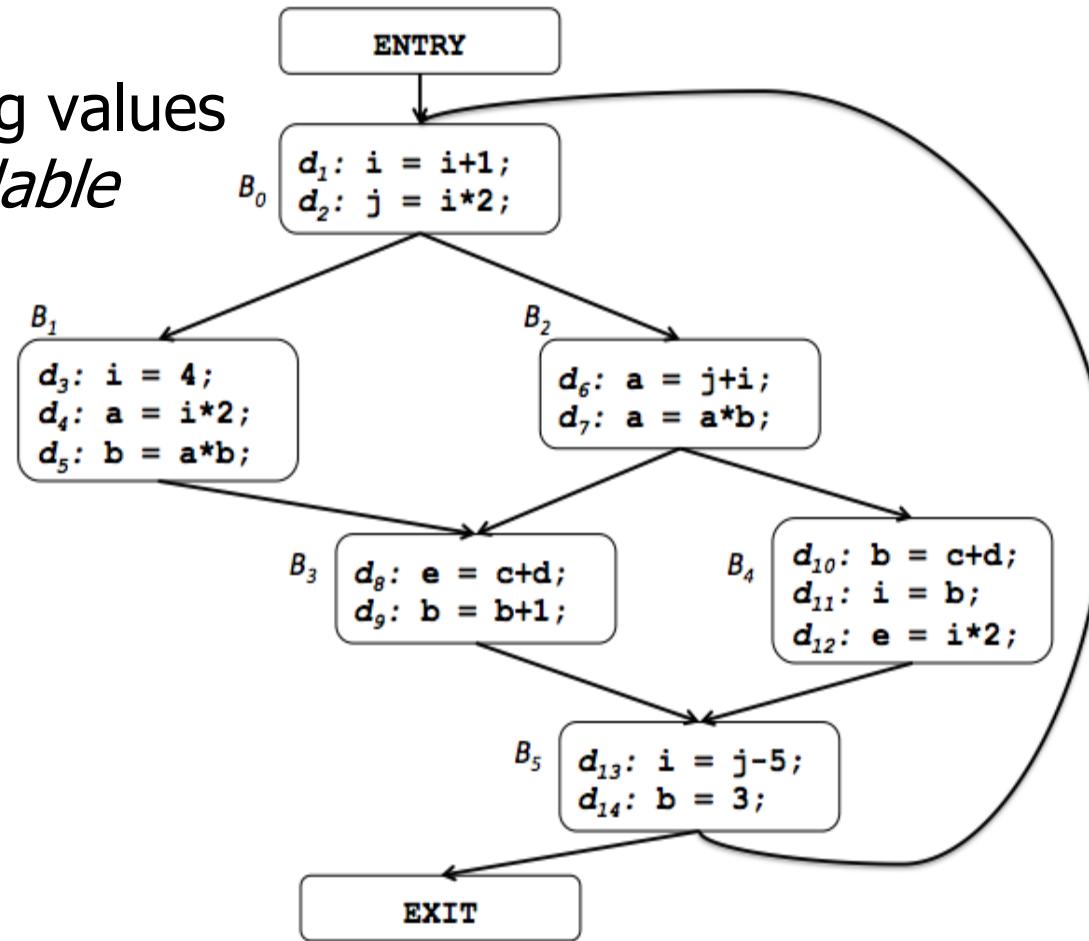


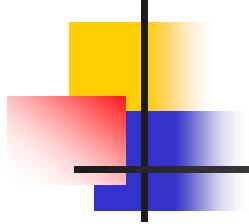
Exercise



- What are the following values according to the *Available Expressions* analysis?

- $\text{gen}(B_0) = \{i^2\}$
- $\text{kill}(B_0) = \{j+i, j-5, i+1\}$
- $\text{Avail}(B_5) = \{i^2, c+d\}$





Live variable equations

$$\text{Live}(n) = \cup \text{LiveOut}(m)$$

$m \in \text{succ}(n)$

**if the variable
might be used in
“any following path”**

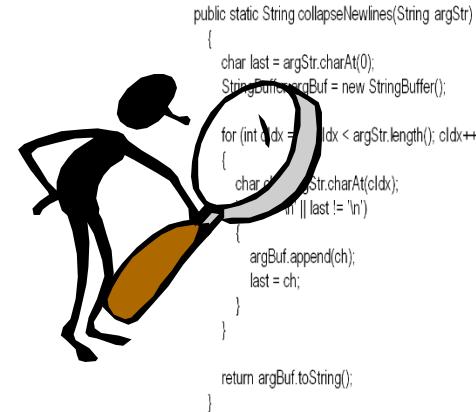
$$\text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v \mid v \text{ is used at } n \}$$

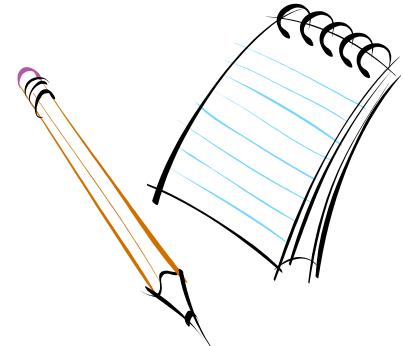
$$\text{kill}(n) = \{ v \mid v \text{ is modified at } n \}$$

Possible Application of Live Variable Analysis

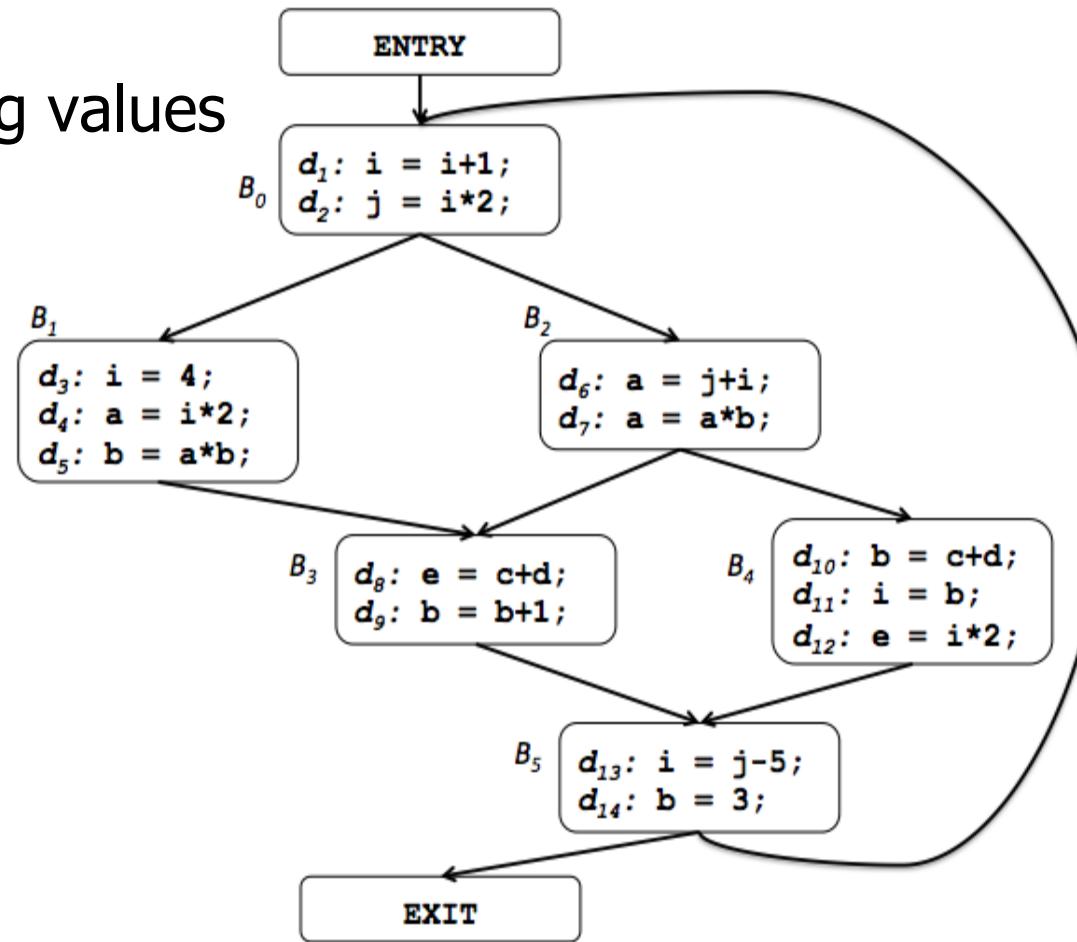
- Recognizing useless definitions
 - Often symptomatic for a fault, e.g., mispelling a variable name



Exercise

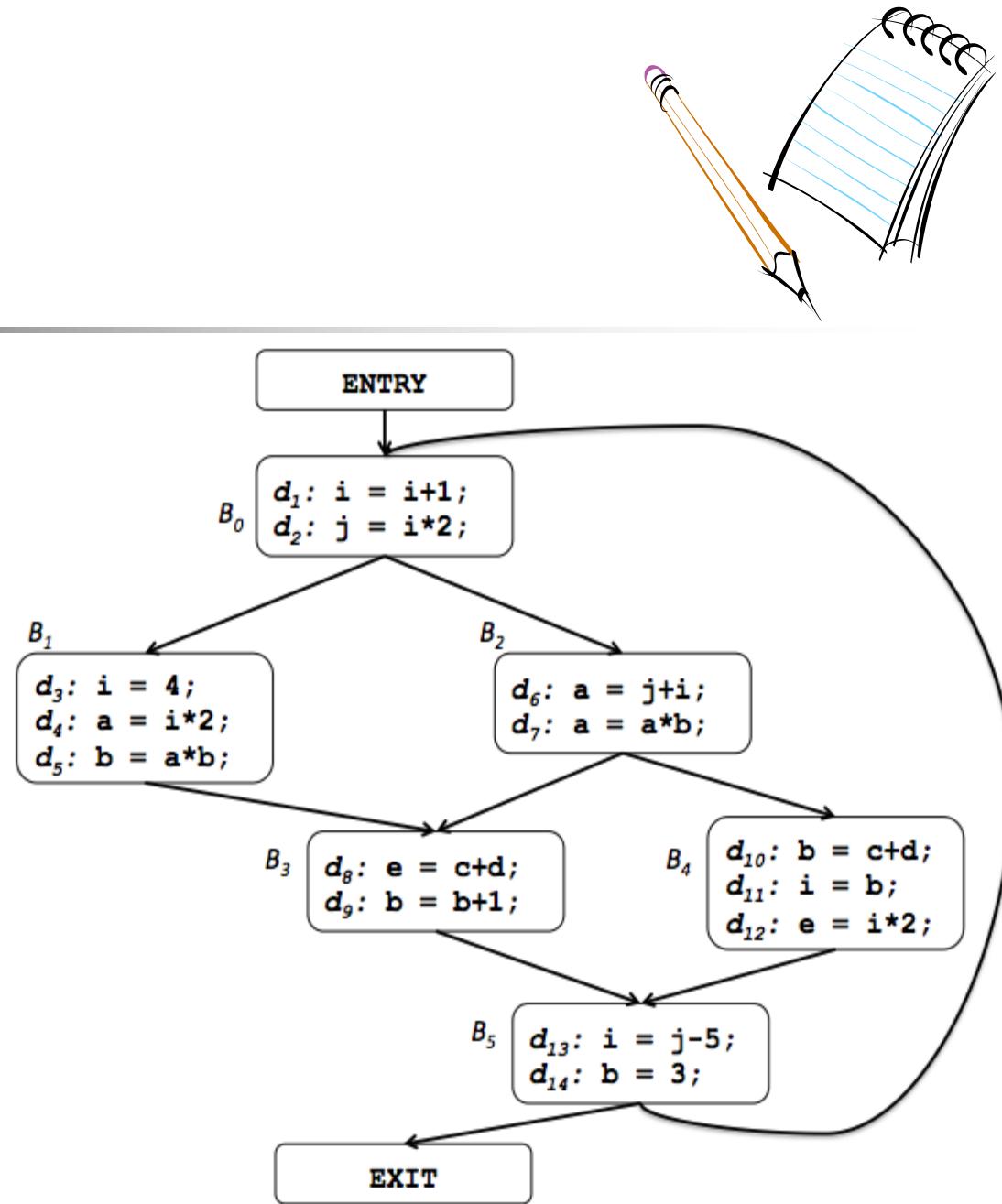


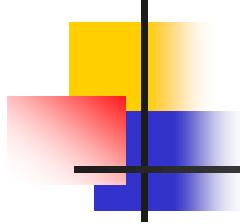
- What are the following values according to the *Live Variables* analysis?
 - $\text{gen}(B_0)$
 - $\text{kill}(B_0)$
 - $\text{Live}(B_0)$



Exercise

- What are the followir according to the *Live Variables* analysis?
 - $\text{gen}(B_0) = \{i\}$
 - $\text{kill}(B_0) = \{i, j\}$
 - $\text{Live}(B_0) = \{i, j, b, c, d\}$



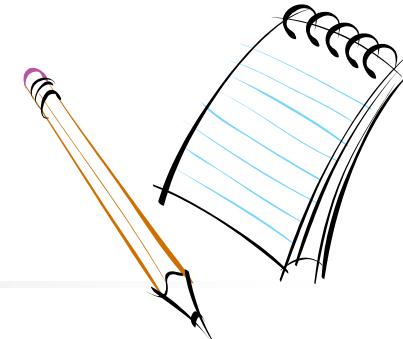


Classification of analyses

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

	Any-path (\cup)	All-paths (\cap)
Forward (pred)	Reach	Avail
Backward (succ)	Live	“inevitable”

Inevitability Definition



$$\text{Inev}(n) = \bigcap_{m \in \text{succ}(n)} \text{InevOut}(m)$$

all paths
backward analysis

$$\text{InevOut}(n) = (\text{Inev}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

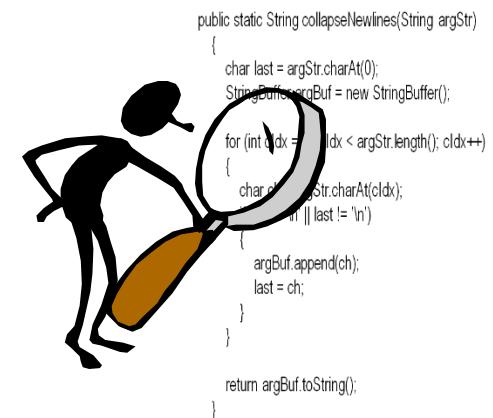
$$\text{gen}(n) = \{ v \mid v = q \}$$

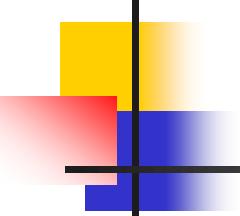
$$\text{kill}(n) = \{ \}$$

**Here, we are interested in
the accessibility of a node, not
(re)definition or modification
of variables**

“Inevitability” Analysis

- Example usage scenarios:
 - Ensuring that interrupts are reenabled after executing an interrupt-handling routine
 - Ensuring that files are closed after opening them
 - ...

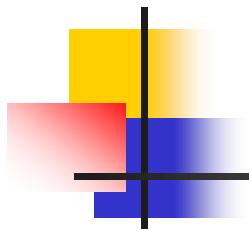




Iterative Solution of Dataflow Equations

- Initialize values (first estimate of answer)
 - For “any path” problems, first guess is “nothing” (empty set) at each node
 - For “all paths” problems, first guess is “everything” (set of all possible values = union of all “gen” sets)
- Repeat until nothing changes
 - Pick some node and recalculate (new estimate)

This will converge on a "fixed point" solution where every new calculation produces the same value as the previous guess.



Data flow analysis with arrays and pointers

- Arrays and pointers introduce uncertainty:
Do different expressions access the same storage?
 - $a[i]$ same as $a[k]$ when $i = k$
 - $a[i]$ same as $b[i]$ when $a = b$ (**aliasing**)
- The uncertainty is accommodated depending to the kind of analysis
 - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
 - All-path: vice versa

CS 575

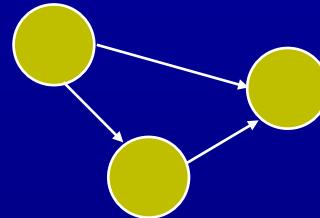
Software Testing and Analysis

Test Coverage and Logic

Slightly modified versions of the textbook slides by Ammann & Offutt

Types of models

- Graphs
 - E.g., prime path coverage
- Logical expressions
 - E.g, clause coverage
- Input space
 - E.g., pair-wise coverage
- Syntax-based (grammars)
 - E.g., production coverage



(not X or not Y) and A and B

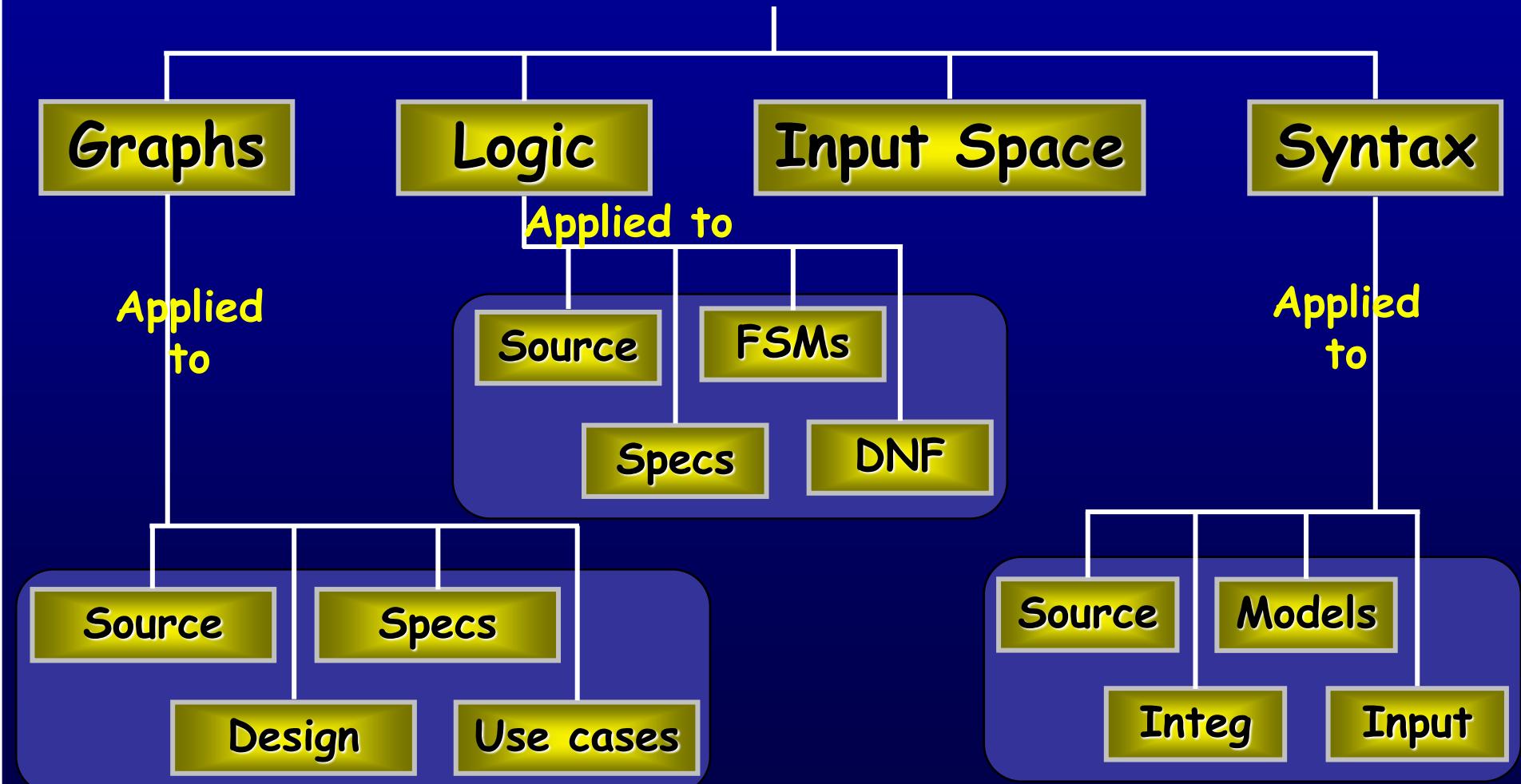
A: {0, 1, >1}

B: {600, 700, 800}

C: {swe, cs, isa, infs}

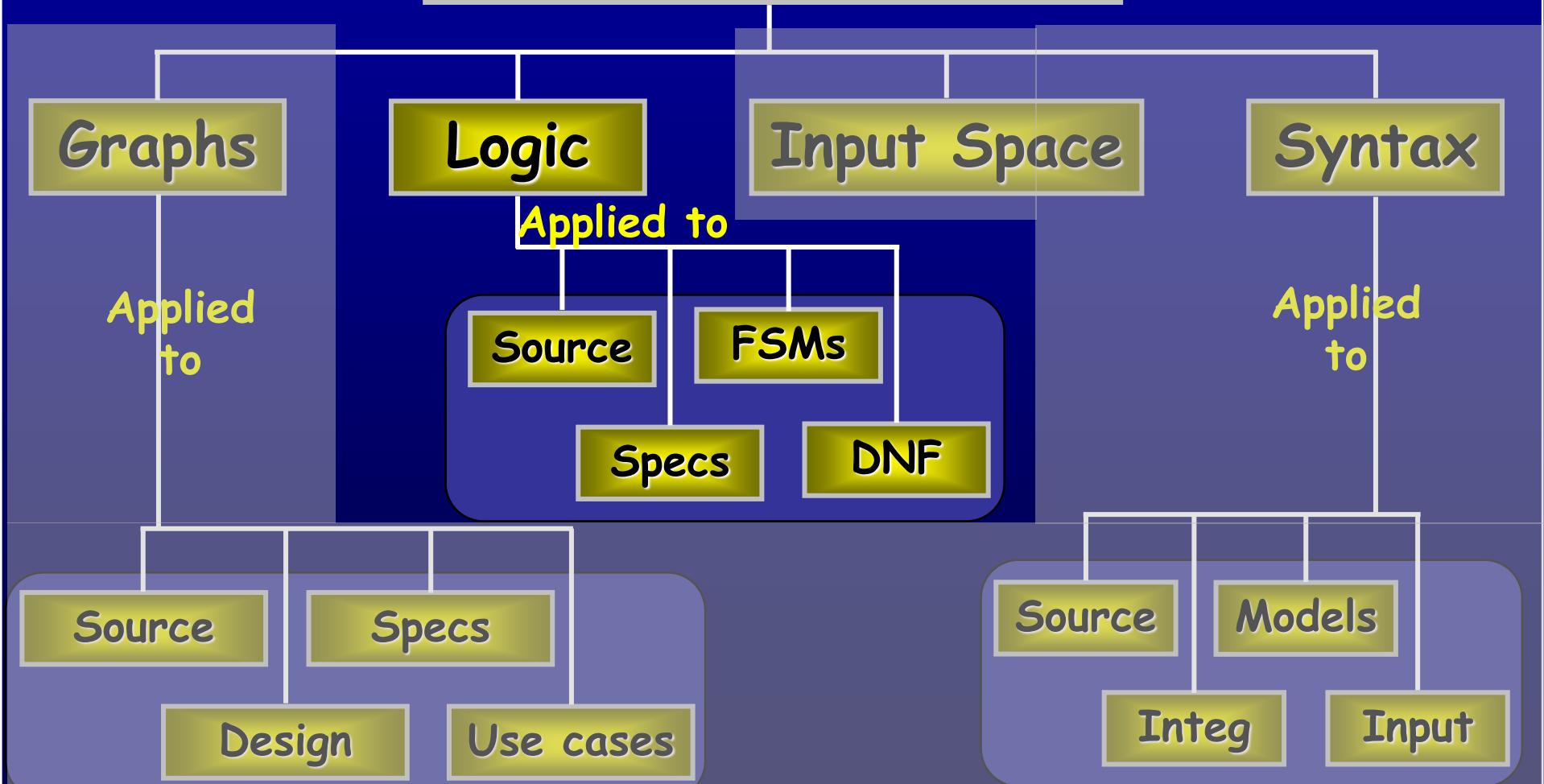
```
if (x > y)
    z = x - y;
else
    z = 2 * x;
```

Derivation of models



Ch. 3 : Logic Coverage

Four Structures for Modeling Software



Covering Logic Expressions (3.1)

- Logic expressions show up in many situations
- Covering logic expressions is required by the US Federal Aviation Administration for safety critical software
- Tests are intended to choose some subset of the total number of truth assignments to the expressions

Logic Predicates and Clauses

- A *predicate* is an expression that evaluates to a **boolean** value
- Predicates can contain
 - **boolean variables**
 - non-boolean variables that contain `>`, `<`, `==`, `>=`, `<=`, `!=`
 - boolean **function calls**
- Internal structure is created by logical operators
 - \neg – the *negation* operator
 - \wedge – the *and* operator
 - \vee – the *or* operator
 - \rightarrow – the *implication* operator
 - \oplus – the *exclusive or* operator
 - \leftrightarrow – the *equivalence* operator
- A *clause* is a predicate with no logical operators

Examples

- $(a < b) \vee f(z) \wedge D \wedge (m \geq n^*o)$
- Four clauses:
 - $(a < b)$ – relational expression
 - $f(z)$ – boolean-valued function
 - D – boolean variable
 - $(m \geq n^*o)$ – relational expression
- Sources of predicates
 - Decisions in programs
 - Guards in finite state machines
 - Decisions in UML activity graphs
 - Requirements, both formal and informal
 - SQL queries

Predicates Derived from Decisions in Programs

- Most predicates have only a few clauses
 - 88.5% have 1 clauses
 - 9.5% have 2 clauses
 - 1.35% have 3 clauses
 - Only .65% have 4 or more !

*from a study of 63 open source programs,
>400,000 predicates*

Testing and Covering Predicates

- We use predicates in testing as follows :
 - Developing a model of the software as one or more predicates
 - Requiring tests to satisfy some combination of clauses
- Abbreviations:
 - P is the set of predicates
 - p is a single predicate in P
 - C is the set of clauses in P
 - C_p is the set of clauses in predicate p
 - c is a single clause in C

Predicate and Clause Coverage

- The first (and simplest) two criteria require that each predicate and each clause be evaluated to both true and false

Predicate Coverage (PC) : For each p in P , TR contains two requirements: p evaluates to true, and p evaluates to false.

- When predicates come from conditions on edges, this is equivalent to edge coverage
- PC does not evaluate all the clauses, so ...

Clause Coverage (CC) : For each c in C , TR contains two requirements: c evaluates to true, and c evaluates to false.

Predicate Coverage Example

$$((a < b) \vee D) \wedge (m \geq n^*o)$$

predicate coverage

Predicate = true

$a = 5, b = 10, D = \text{true}, m = 1, n = 1, o = 1$
 $= (5 < 10) \vee \text{true} \wedge (1 \geq 1^*1)$
 $= \text{true} \vee \text{true} \wedge \text{TRUE}$
 $= \text{true}$

Predicate = false

$a = 10, b = 5, D = \text{false}, m = 1, n = 1, o = 1$
 $= (10 < 5) \vee \text{false} \wedge (1 \geq 1^*1)$
 $= \text{false} \vee \text{false} \wedge \text{TRUE}$
 $= \text{false}$

Clause Coverage Example

$$((a < b) \vee D) \wedge (m \geq n * o)$$

Clause coverage

<u>$(a < b) = \text{true}$</u>	<u>$(a < b) = \text{false}$</u>	<u>$D = \text{true}$</u>	<u>$D = \text{false}$</u>
$a = 5, b = 10$	$a = 10, b = 5$	$D = \text{true}$	$D = \text{false}$

<u>$m \geq n * o = \text{true}$</u>	<u>$m \geq n * o = \text{false}$</u>
$m = 1, n = 1, o = 1$	$m = 1, n = 2, o = 2$

true cases

false cases

Two tests

- 1) $a = 5, b = 10, D = \text{true}, m = 1, n = 1, o = 1$
- 2) $a = 10, b = 5, D = \text{false}, m = 1, n = 2, o = 2$

Problems with PC and CC

- PC does not fully exercise all the clauses, especially in the presence of short circuit evaluation
- CC does not always ensure PC
 - That is, we can satisfy CC without causing the predicate to be both true and false
 - This is definitely not what we want !
- The simplest solution is to test all combinations ...

Combinatorial Coverage

- CoC requires every possible combination
- Sometimes called Multiple Condition Coverage

Combinatorial Coverage (CoC) : For each p in P , TR has test requirements for the clauses in C_p to evaluate to each possible combination of truth values.

	$a < b$	D	$m \geq n^*o$	$((a < b) \vee D) \wedge (m \geq n^*o)$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Combinatorial Coverage

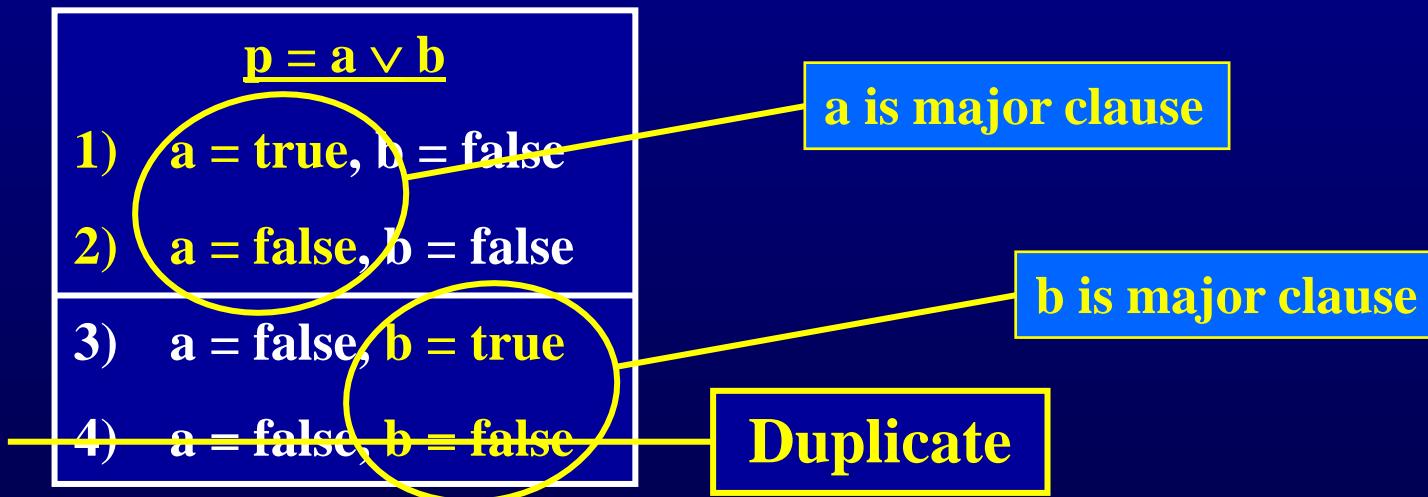
- This is simple, neat, clean, and comprehensive ...
- But quite expensive!
- 2^N tests, where N is the number of clauses
 - Impractical for predicates with more than 3 or 4 clauses
- The literature has lots of suggestions – some confusing
- The general idea is simple:

Test each clause independently from the other clauses

- Getting the details right is hard
- What exactly does “independently” mean ?
- The book presents this idea as “*making clauses active*” ...

Active Clause Coverage

Active Clause Coverage (ACC) : For each p in P and each major clause ci in Cp , choose minor clauses $cj, j \neq i$, so that ci determines p . TR has two requirements for each ci : ci evaluates to true and ci evaluates to false.



Active Clause Coverage

- This is a form of MCDC, which is required by the FAA for safety critical software
- $N+1$ tests are sufficient for coverage, where N is the number of clauses
- Ambiguity : Do the minor clauses have to have the same values when the major clause is true and false?

Resolving the Ambiguity

$$\underline{p = a \vee (b \wedge c)}$$

Major clause : a

a = true, b = false, c = true

a = false, b = false, **c = false**

Is this allowed ?

- Separate criteria defined to avoid ambiguity
 - Minor clauses do not need to be the same
 - General Active Clause Coverage (GACC)
 - Minor clauses do need to be the same
 - Restricted Active Clause Coverage (RACC)

Exercise

```
for(n = 0;  
    n < max_size && (c = getc( yyin )) != EOF && c != '\n';  
    n++)  
    buf[n] = (char) c;
```

- Devise a set of test cases that satisfy the GACC and RACC criteria with respect to the loop condition
- Hint: There exist 3 clauses. So, 4 test cases should be sufficient to satisfy GACC and RACC

Adopted from the textbook slides by Pezze & Young

Exercise

```
for(n = 0;  
    n < max_size && (c = getc( yyin )) != EOF && c != '\n';  
    n++)  
    buf[n] = (char) c;
```

- Entries marked with "-" normally can be either of true or false to satisfy GACC. However, we should set them to true for satisfying RACC.

Test Case	$n < \text{max_size}$	$(c = \text{getc}(yyin)) \neq \text{EOF}$	$c \neq '\backslash n'$	Outcome
(1)	<u>false</u>	-	-	false
(2)	true	<u>false</u>	-	false
(3)	true	true	<u>false</u>	false
(4)	<u>true</u>	true	<u>true</u>	true



Test Case	$n < \text{max_size}$	$(c = \text{getc}(yyin)) \neq \text{EOF}$	$c \neq '\backslash n'$	Outcome
(1)	<u>false</u>	true	true	false
(2)	true	<u>false</u>	true	false
(3)	true	true	<u>false</u>	false
(4)	true	true	<u>true</u>	true

Correlated Active Clause Coverage

Correlated Active Clause Coverage (CACC) : For each p in P and each major clause c_i in C_p , choose minor clauses $c_j, j \neq i$, so that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other, that is, it is required that $p(c_i = \text{true}) \neq p(c_i = \text{false})$.

- A more recent interpretation
- Implicitly allows minor clauses to have different values
- Explicitly satisfies (**subsumes**) predicate coverage

CACC and RACC

	a	b	c	$a \wedge (b \vee c)$
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

major clause

$P_a : b = \text{true or } c = \text{true}$

CACC can be satisfied by choosing any of rows 1, 2, 3 AND any of rows 5, 6, 7 – a total of nine pairs

	a	b	c	$a \wedge (b \vee c)$
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

RACC can only be satisfied by row pairs (1, 5), (2, 6), or (3, 7)

Only three pairs

Exercise by Paul Amman

- Devise a set of test cases that satisfy the CACC and RACC criteria with respect to the predicate

	a	b	c	P	P_a
1	T	T	T	T	T
2	T	T		T	T
3	T		T		T
4	T			T	
5		T	T	T	T
6		T			T
7			T	T	T
8				T	

$P = \bar{a}\bar{b} + a\bar{c} + \bar{a}c$

All pairs of p in Pa

- $Pa = T, a = T: 1, 2, 3$
- $Pa = T, a = F: 5, 6, 7$
- All Pairs:
 $(1,5) (2,5) (3,5)$
 $(1,6) (2,6) (3,6)$
 $(1,7) (2,7) (3,7)$
- Satisfies GACC

	a	b	c	P	P_a
1	T	T	T		T
2	T	T		T	T
3	T		T		T
4	T			T	
5		T	T	T	T
6		T			T
7			T	T	T
8				T	

$P = \bar{a}\bar{b} + a\bar{c} + \bar{a}c$

Pairs that satisfy CACC

- $P_a = T, a = T: 1, 2, 3$
- $P_a = T, a = F: 5, 6, 7$
- All Pairs where P changes:
~~(1,5) (2,5) (3,5)~~
~~(1,6) (2,6) (3,6)~~
~~(1,7) (2,7) (3,7)~~

	a	b	c	P	P_a
1	T	T	T		T
2	T	T		T	T
3	T		T		T
4	T			T	
5		T	T	T	T
6		T			T
7			T	T	T
8				T	

$P = \bar{a}\bar{b} + a\bar{c} + \bar{a}c$

Pairs that satisfy RACC

- $\bar{P}a = T, a = T: 1, 2, 3$
- $\bar{P}a = T, a = F: 5, 6, 7$
- All Pairs where minor clauses remain the same:

~~(1,5) (2,5) (3,5)~~

~~(1,6) (2,6) (3,6)~~

~~(1,7) (2,7) (3,7)~~

c: (5,6), (3,4)

b: (6,8)

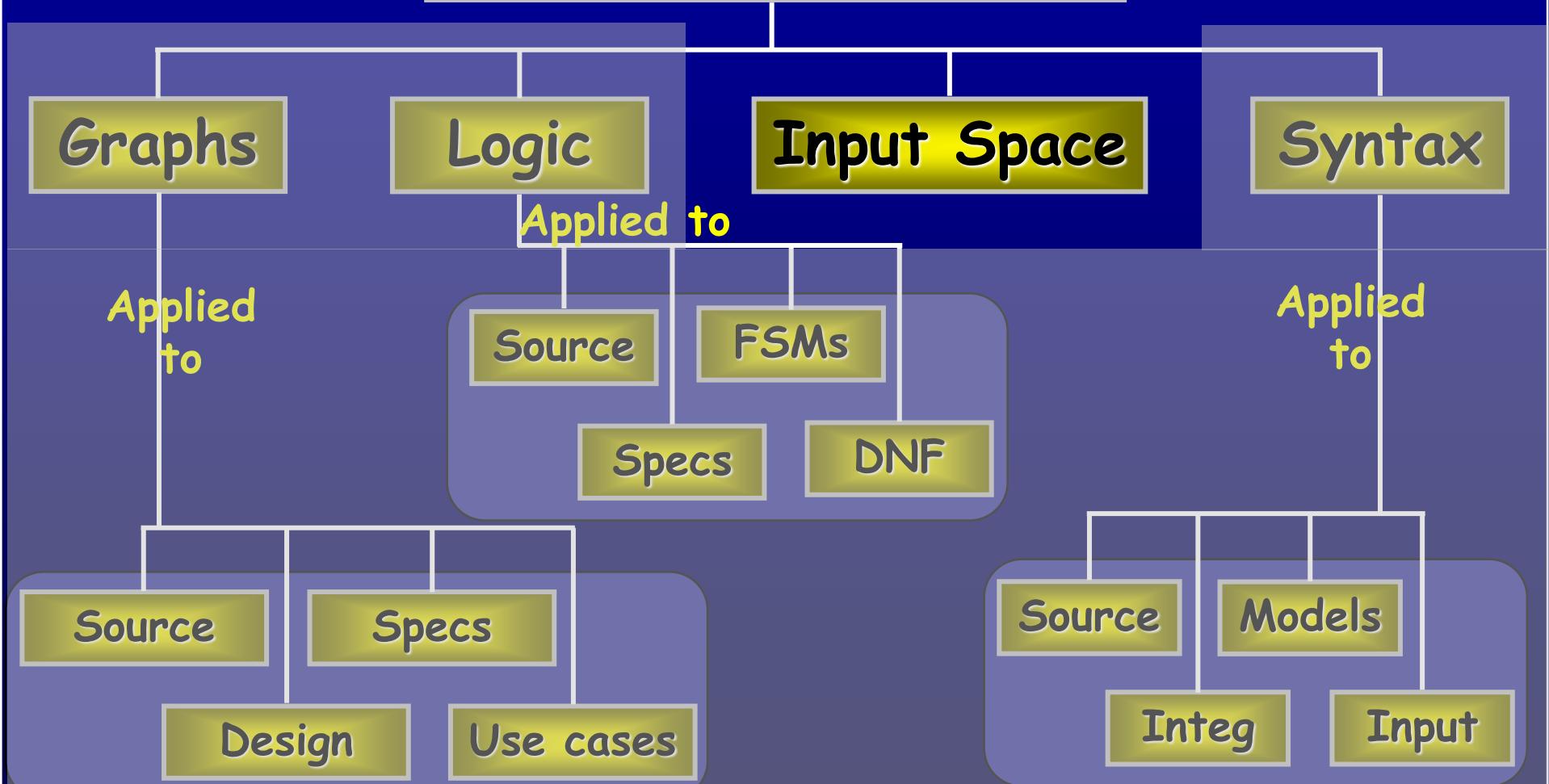
T: 6, 2, 5, 8

	a	b	c	P	$\bar{P}a$
1	T	T	T		T
2	T	T		T	T
3	T		T		T
4	T			T	
5		T	T	T	T
6		T			T
7			T	T	T
8				T	

$P = \bar{a}\bar{b} + a\bar{c} + \bar{a}c$

Ch. 4 : Input Space Coverage

Four Structures for Modeling Software



Input Domains

- The input domain to a program contains all the possible inputs to that program
- Testing is fundamentally about choosing finite sets of values from the input domain

The Source of Input Domain at different abstraction levels

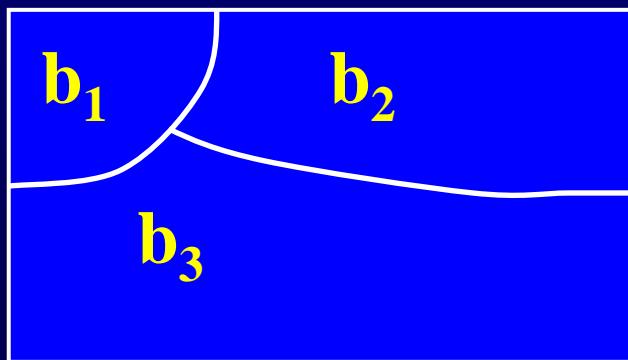
- System level
 - Number of students $\{ 0, 1, >1 \}$
 - Level of course $\{ 600, 700, 800 \}$
 - Major $\{ swe, cs, isa, infs \}$
- Unit level
 - Parameters $F (int X, int Y)$
 - Possible values $X: \{ <0, 0, 1, 2, >2 \}, Y : \{ 10, 20, 30 \}$
 - Tests $F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)$

Input Domain Partitioning

- Domain for each input parameter is partitioned into regions
- At least one value is chosen from each region

Partitioning Domains

- Domain D
- Partition scheme q of D
- The partition q defines a set of blocks, $Bq = b_1, b_2, \dots b_Q$
- The partition must satisfy two **properties** :
 1. blocks must be pairwise disjoint (no overlap)
 2. together the blocks cover the domain D (complete)



Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the “order of file F”

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

but ... something's fishy ...

What if the file is of length 1?

The file will be in all three blocks ...

That is, disjointness is not satisfied

Solution:

Each characteristic should address just one property

C1: File F sorted ascending

- c1.b1 = true
- c1.b2 = false

C2: File F sorted descending

- c2.b1 = true
- c2.b2 = false

Properties of Partitions

- If the partitions are not **complete** or **disjoint**, that means the partitions have not been considered carefully enough
- They should be reviewed carefully, like any design attempt
- Different **alternatives** should be considered

Exercise

- Consider the following characteristics and blocks for an input parameter, **car**:

- **Where Made**

- b1: North America
 - b2: Europe
 - b3: Asia

- **Energy Source**

- b1: Gas Only
 - b2: Electric Only
 - b3: Hybrid

- **Size**

- b1: 2-door
 - b2: 4-door
 - b3: hatch-back

What are the mistakes here?

How can they be corrected?

Exercise

– Where Made

- b1: North America
- b2: Europe
- b3: Asia

Not complete!

May be an additional block, "other"?

– Energy Source

- b1: Gas Only
- b2: Electric Only
- b3: Hybrid

– Size

- b1: 2-door
- b2: 4-door
- b3: hatch-back

Not disjoint!

A hatchback can be 2-door or 4-door

Either add more blocks such as
2-door+hatch-back and 4-door+hatch-back
or

Add more characteristics:

Side Doors; b1: 2, b2: 4

Hatch-back; b1: True, b2: False

Input Domain Modeling

- Step 1: Identify the input domain
- Step 2: Identify equivalence classes (partitioning)
- Two basic approaches
 - Interface Based
 - Functionality Based

Interface Based Input Domain Modeling

- Each parameter is considered in isolation
- Each characteristic is related to a single parameter
- Simple, can be automated 
- Ignores parameter interactions/relations
- Can lead to incomplete domain modeling 

Functionality Based Input Domain Modeling

- Use semantics and domain knowledge
- Characteristics correspond to intended functionality
- Modeling is based on requirements; can start early
- Takes parameter relationships into account
- Hard to identify characteristics
- Hard to translate values to concrete test cases



Example

- **public boolean findElement(List list, Object element)**
 - If list or element is null, throw NullPointerException
 - Else if element is in the list, return true
 - Else, return false
- **Interface-based characteristics for list:**
 - list is null
 - b1 = True
 - b2 = False
 - list is empty
 - b1 = True
 - b2 = False
- **Functionality-based characteristics:**
 - Number of occurrences of element in list
 - b1 = 0
 - b2 = 1
 - b3 = more than 1
 - Element occurs first in list
 - b1 = True
 - b2 = False

Choosing Blocks and Values

- A creative design step not to unnecessarily increase the number of test cases and capture all faults at the same time
- General strategy for indentifying values
 - Partition valid values for different part of the functionality
 - Check for completeness (missing partitions)
 - Check for disjointness (overlapping partitions)
- Selection
 - Valid values
 - Boundaries
 - Normal use
 - Invalid values

Running Example

- **TriTyp program**
 - Input: Side1, Side2, Side3
 - **3 integer values that represent the lengths of the sides of a triangle**
 - Output: the category of triangle
 - **Scalene**
 - **Isosceles**
 - **Equilateral**
 - **Invalid**

TriTyp: Interface-Based Modeling

- Relation of a variable with respect to some special value, 0

First Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3
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

- A maximum of $3*3*3 = 27$ tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests ...

TriTyp: Interface-Based Modeling

Second Characterization of TriTyp Inputs

Characteristic	b_1	b_2	b_3	b_4
$q_1 = \text{"Refinement of } q_1\text{"}$	greater than 1	equal to 1	equal to 0	less than 0
$q_2 = \text{"Refinement of } q_2\text{"}$	greater than 1	equal to 1	equal to 0	less than 0
$q_3 = \text{"Refinement of } q_3\text{"}$	greater than 1	equal to 1	equal to 0	less than 0

- A maximum of $4*4*4 = 64$ tests
- Complete because the inputs are integers (0 .. 1)

Possible values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side 1	2	1	0	-1

Test boundary conditions

TriTyp: Functionality-Based Modeling

- First two characterizations are based on syntax—parameters and their type
- A **semantic** level characterization could use the fact that the three integers represent a triangle

Geometric Characterization of TriTyp Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles	equilateral	invalid

- Oops ... something's fishy ... equilateral is also isosceles !
- We need to **refine** the example to make characteristics valid

Correct Geometric Characterization of TriTyp Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles, not equilateral	equilateral	invalid

TriTyp: Functionality-Based Modeling

- Values for this partitioning can be chosen as

Possible values for geometric partition q_1

Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

TriTyp: Functionality-Based Modeling

- A **different approach** would be to break the geometric characterization into four separate characteristics

Four Characteristics for *TriTyp*

Characteristic	b_1	b_2
----------------	-------	-------

$q_1 = \text{"Scalene"}$	True	False
$q_2 = \text{"Isosceles"}$	True	False
$q_3 = \text{"Equilateral"}$	True	False
$q_4 = \text{"Valid"}$	True	False

- Use constraints to ensure that
 - Equilateral = True implies Isosceles = True
 - Valid = False implies Scalene = Isosceles = Equilateral = False

Choosing Combinations of Values

- Once characteristics and partitions are defined, the next step is to **choose test values**
- We use **criteria** – to choose **effective** subsets
- The most obvious criterion is to choose all combinations

All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.

- Number of tests is the product of the number of blocks in each characteristic : $\prod_{i=1}^Q (B_i)$
- The second characterization of TriTyp results in $4*4*4 = 64$ tests
 - Too many ?

Input Space Partitioning (ISP)

Coverage Criteria – All Combinations

- Consider the “second characterization” of TriTyp as given before:

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Refinement of q_1 ”	greater than 1	equal to 1	equal to 0	less than 0
q_2 = “Refinement of q_2 ”	greater than 1	equal to 1	equal to 0	less than 0
q_3 = “Refinement of q_3 ”	greater than 1	equal to 1	equal to 0	less than 0

- For convenience, we relabel the blocks:

Characteristic	b_1	b_2	b_3	b_4
A	A1	A2	A3	A4
B	B1	B2	B3	B4
C	C1	C2	C3	C4

ISP Criteria – ACoC Tests

A1 B1 C1	A2 B1 C1	A3 B1 C1	A4 B1 C1
A1 B1 C2	A2 B1 C2	A3 B1 C2	A4 B1 C2
A1 B1 C3	A2 B1 C3	A3 B1 C3	A4 B1 C3
A1 B1 C4	A2 B1 C4	A3 B1 C4	A4 B1 C4

A1 B2 C1	A2 B2 C1	A3 B2 C1	A4 B2 C1
A1 B2 C2	A2 B2 C2	A3 B2 C2	A4 B2 C2
A1 B2 C3	A2 B2 C3	A3 B2 C3	A4 B2 C3
A1 B2 C4	A2 B2 C4	A3 B2 C4	A4 B2 C4

A1 B3 C1	A2 B3 C1	A3 B3 C1	A4 B3 C1
A1 B3 C2	A2 B3 C2	A3 B3 C2	A4 B3 C2
A1 B3 C3	A2 B3 C3	A3 B3 C3	A4 B3 C3
A1 B3 C4	A2 B3 C4	A3 B3 C4	A4 B3 C4

A1 B4 C1	A2 B4 C1	A3 B4 C1	A4 B4 C1
A1 B4 C2	A2 B4 C2	A3 B4 C2	A4 B4 C2
A1 B4 C3	A2 B4 C3	A3 B4 C3	A4 B4 C3
A1 B4 C4	A2 B4 C4	A3 B4 C4	A4 B4 C4

ACoC yields
 $4*4*4 = 64$ tests
for TriTyp!

This is almost certainly more than we need

Only 8 are valid
(all sides greater than zero)

ISP Criteria – Each Choice

- 64 tests for TriTyp is almost certainly way too many
- One criterion comes from the idea that we should try at least one value from each block

Each Choice Coverage (ECC) : One value from each block for each characteristic must be used in at least one test case.

- Number of tests is the number of blocks in the largest characteristic : $\text{Max}_{i=1}^Q(B_i)$

For TriTyp : A1, B1, C1

A2, B2, C2

A3, B3, C3

A4, B4, C4

Substituting values: 2, 2, 2

I, I, I

0, 0, 0

-I, -I, -I

ISP Criteria – Pair-Wise

- Each choice yields few tests—**cheap** but maybe ineffective
- Another approach **combines** values with other values

Pair-Wise Coverage (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

- Number of tests is at least the product of two largest characteristics $(\text{Max}_{i=1}^Q (B_i)) * (\text{Max}_{j=1, j \neq i}^Q (B_j))$

For TriTyp : A1, B1, C1	A1, B2, C2	A1, B3, C3	A1, B4, C4
A2, B1, C2	A2, B2, C3	A2, B3, C4	A2, B4, C1
A3, B1, C3	A3, B2, C4	A3, B3, C1	A3, B4, C2
A4, B1, C4	A4, B2, C1	A4, B3, C2	A4, B4, C3

ISP Criteria –T-Wise

- A natural extension is to require combinations of t values instead of 2

t-Wise Coverage (TWC) :A value from each block for each group of t characteristics must be combined.

- Number of tests is at least the product of t largest characteristics
- If all characteristics are the same size, the formula is

$$(\text{Max}_{i=1}^Q(B_i))^t$$

- If t is the number of characteristics Q , then all combinations
- That is ... $Q\text{-WC} = ACoC$
- t -wise is expensive and benefits are not clear

ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are **important**
- This uses **domain knowledge** of the program

Base Choice Coverage (BCC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

- Number of tests is one base test + one test for each other block $1 + \sum_{i=1}^Q (B_i - 1)$

For TriTyp : Base AI,BI,CI AI,BI,C2 AI,B2,CI A2,BI,CI
 AI,BI,C3 AI,B3,CI A3,BI,CI
 AI,BI,C4 AI,B4,CI A4,BI,CI

Base Choice Notes

- The base test must be **feasible**
 - That is, all base choices must be **compatible**
- **Base choices** can be
 - Most likely from an end-use point of view
 - Simplest
 - Smallest
 - First in some ordering
- **Happy path** tests often make good base choices
- The base choice is a **crucial design** decision
 - Test designers should **document** why the choices were made

ISP Criteria – Multiple Base Choice

- We sometimes have **more than one logical base choice**

Multiple Base Choice Coverage (MBCC) : At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

- If **M** base tests and m_i base choices for each characteristic:

$$M + \sum_{i=1}^Q (M * (B_i - m_i))$$

ISP Criteria – Multiple Base Choice

- We sometimes have more than one logical base choice

Multiple Base Choice Coverage (MBCC) : At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

For TriTyp : Bases

A1, B1, C1 A1, B1, C3 A1, B3, C1 A3, B1, C1

A1, B1, C4 A1, B4, C1 A4, B1, C1

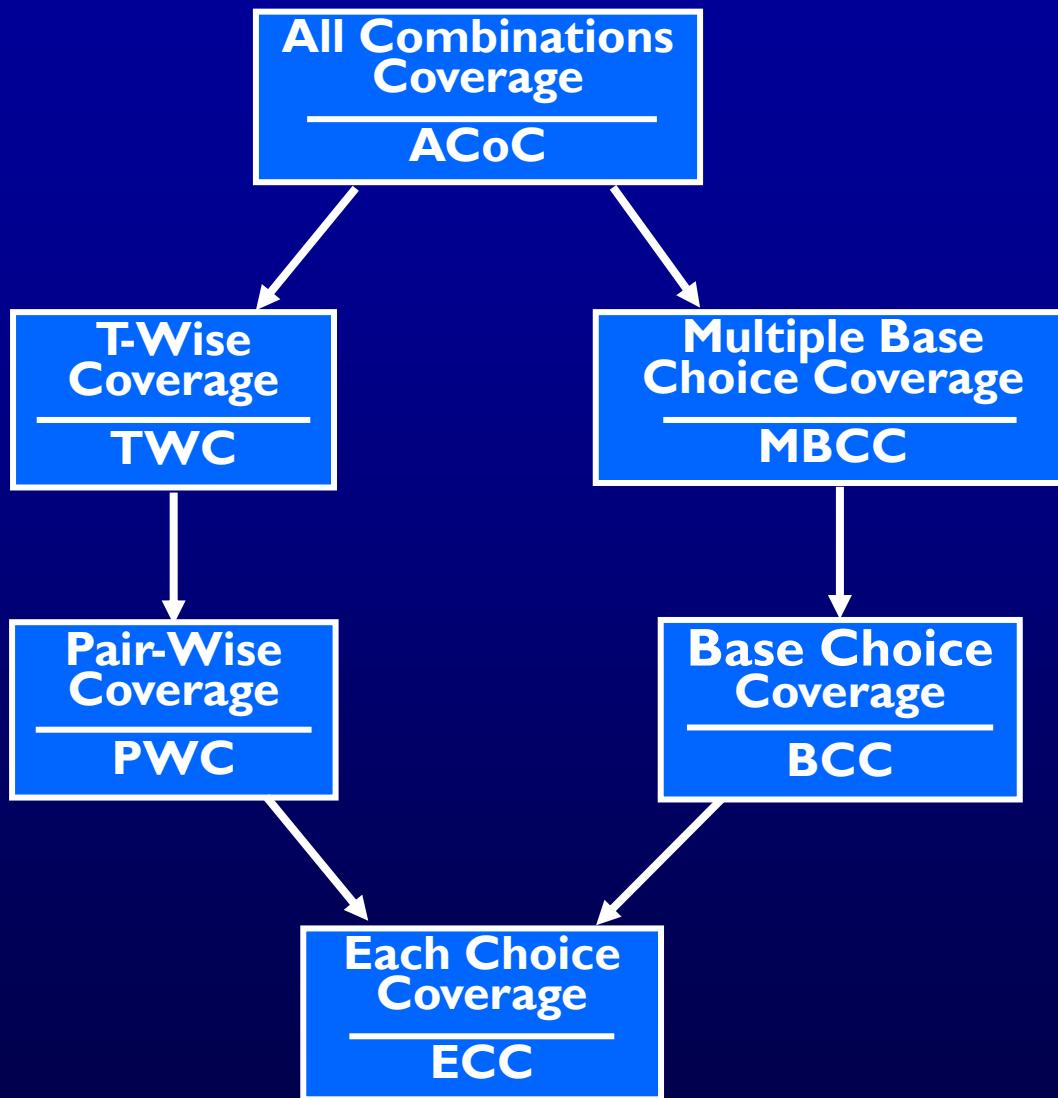
A1, B1, C2 A1, B2, C1 A2, B1, C1

A2, B2, C2 A2, B2, C3 A2, B3, C2 A3, B2, C2

A2, B2, C4 A2, B4, C2 A4, B2, C2

A2, B2, C1 A2, B1, C2 A1, B2, C2

ISP Coverage Criteria Subsumption



Exercise 1.1

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- Provide test cases that satisfy Each Choice coverage

Exercise 1.1

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- 4 test cases can satisfy Each Choice coverage

<i>V1</i>	<i>V2</i>	<i>Op</i>
-2	-2	+
0	0	-
2	2	×
2	2	÷

Exercise 1.2

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- Provide test cases that satisfy **Base Choice coverage**
 - Assume base choices as
 - Value 1: > 0
 - Value 2: > 0
 - Operation: +

Exercise 1.2

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- 8 test cases can satisfy
Base Choice coverage
 - Assuming base choices as
 - Value 1 > 0
 - Value 2 > 0
 - Operation = +

<i>V1</i>	<i>V2</i>	<i>Op</i>
2	2	+
-2	2	+
0	2	+
2	-2	+
2	0	+
2	2	-
2	2	×
2	2	÷

Exercise 1.3

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- How many test cases are needed to satisfy the All Combinations (ACoC) criterion?

Exercise 1.3

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- How many test cases are needed to satisfy the All Combinations (ACoC) criterion?

$$- 3 * 3 * 4 = 36$$

Exercise 1.4

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- Provide test cases that satisfy Pair-Wise coverage

Exercise 1.4

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- $4 * 3 = 12$ test cases are required according to the previously introduced formula

$$(\text{Max}_{i=1}^Q(B_i)) * (\text{Max}_{j=1, j \neq i}^Q(B_j))$$

Exercise 1.4

- Consider the following characteristics and blocks

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	-	×	÷

- 12 test cases can satisfy Pair-Wise coverage

-2 -2 +,
0 0 +,
2 2 +,

0 -2 -,
2 0 -,
-2 2 -,

2 -2 ×,
-2 0 ×,
0 2 ×,

-2 -2 ÷,
0 0 ÷,
2 2 ÷,

Constraints Among Characteristics

(6.3)

- Some combinations of blocks are **infeasible**
 - “less than zero” and “scalene” ... not possible at the same time
- These are represented as **constraints among blocks**
- Two general types of constraints
 - A block from one characteristic **cannot be combined with a specific block from another**
 - A block from one characteristic can **ONLY BE** combined with a specific block from another characteristic
- Handling constraints depends on the criterion used
 - **ACC, PWC, TWC** : Drop the infeasible pairs
 - **BCC, MBCC** : Change a value to another non-base choice to find a feasible combination

Example Handling Constraints

- Sorting an array

- Input : variable length array of arbitrary type
- Outputs : sorted array, largest value, smallest value

Blocks from other characteristics are irrelevant

Characteristics

- Length
- Type of elements
- Max value
- Min value
- Position of elements
- Position of values

Partitions:

- Len { 0, 1, 2..100, 101..MAXINT }
- Type { int, char, string, other }
- Max { ≤0, 1, >1, 'a', 'Z', 'b', ..., 'Y' }
- Min { ... }
- Max Pos { 1, 2 .. Len-1, Len }
- Min Pos { 1, 2 .. Len-1, Len }

Blocks must be combined

Blocks must be combined

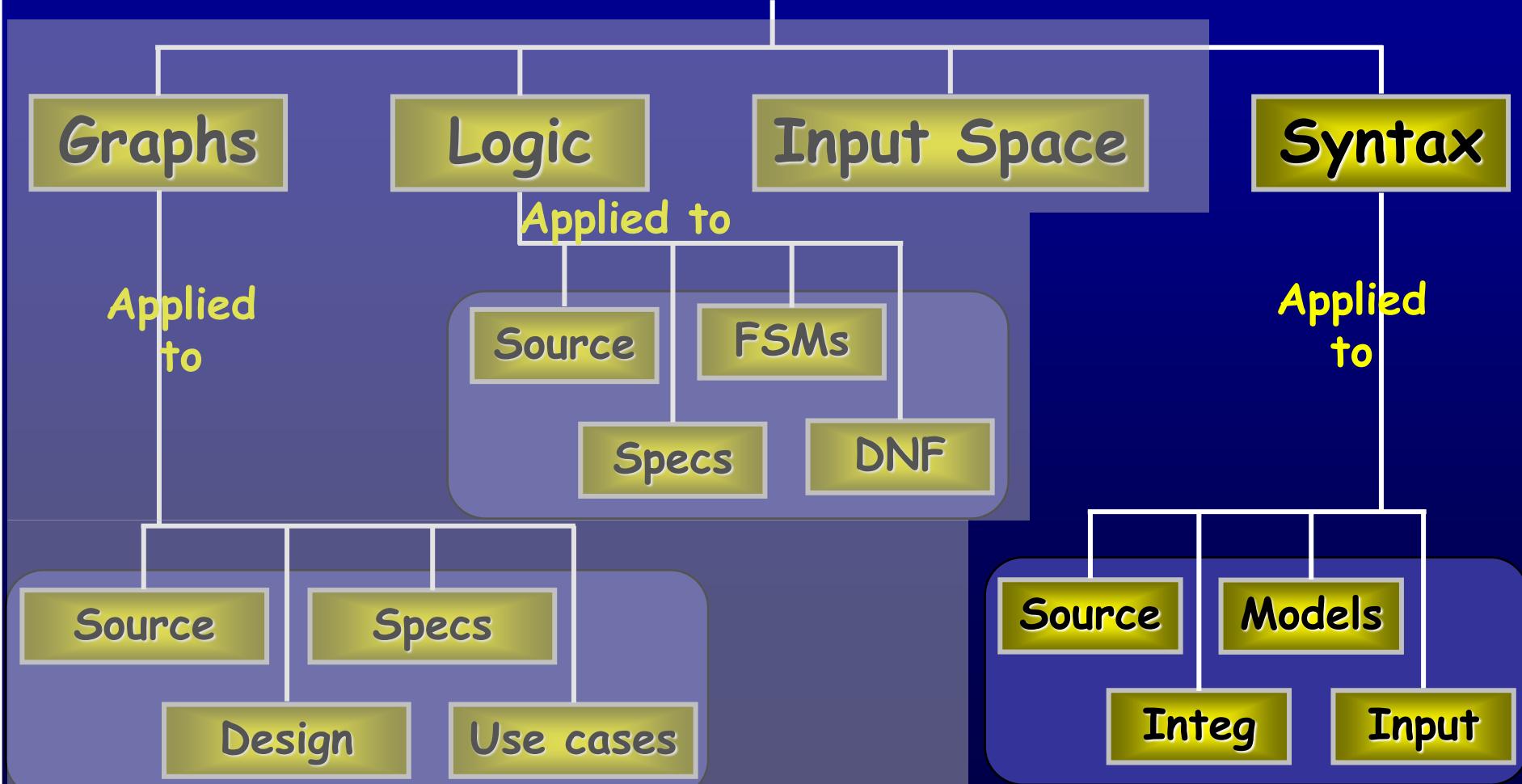
Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used in practice

Ch. 5 : Syntax Coverage

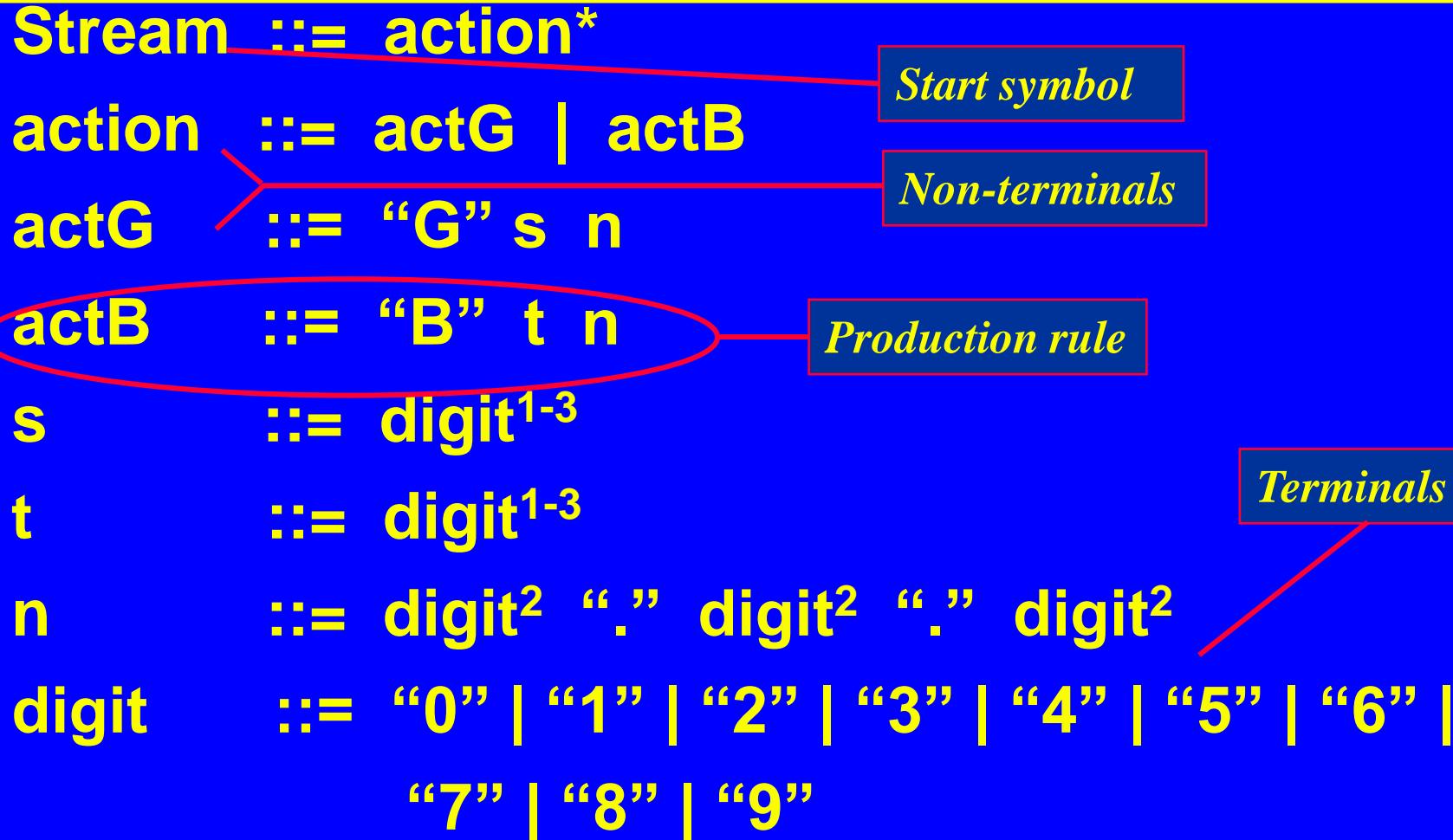
Four Structures for Modeling Software



Using the Syntax to Generate Tests (5.1)

- Lots of software artifacts follow strict syntax rules
- The syntax is often expressed as some sort of grammar such as BNF
- Syntactic descriptions can come from many sources
 - Programs
 - Integration elements
 - Design documents
 - Input descriptions
- Tests are created with two general goals
 - Cover the syntax in some way
 - Violate the syntax (invalid tests)

BNF Grammars



Test Cases from Grammar

- A string that satisfies the derivation rules is said to be “*in the grammar*”
- A test case is a **sequence of strings** that satisfy the regular expression
- Suppose ‘s’, ‘t’ and ‘n’ are numbers

G 20 08.01.90

B 16 06.27.94

G 15 11.21.94

B 07 01.09.03

Could be one test with four parts,
four separate tests, . . .

Exercise 1

- Consider the grammar provided for Canadian postal codes (taken from the lecture materials of Neal R. Wagner)
- Which of the following strings are invalid? Why?
 - K1N 6N5
 - B3D 1Z7
 - M5W 2E4
 - X0A 1A1
 - A4A CE3

Postalcode ::= ForwardSortationArea Space LocalDeliveryUnit

ForwardSortationArea ::= Letter Digit Letter

LocalDeliveryUnit ::= Digit Letter Digit

Space ::= “ ”

**Letter ::= “A” | “B” | “C” | “E” | “G” | “H” | “J” | “K” | “L” | “M” | “N” | “P”
| “R” | “S” | “T” | “V” | “W” | “X” | “Y” | “Z”**

Digit ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

Exercise 1

- Consider the grammar provided for Canadian postal codes (taken from the lecture materials of Neal R. Wagner)
- Which of the following strings are invalid? Why?
 - K1N 6N5: OK
 - B3D 1Z7: “D” does not exist in the list of terminals
 - M5W 2E4: OK
 - X0A 1A1: OK
 - A4A CE3: Second part cannot start with a letter

Postalcode ::= ForwardSortationArea Space LocalDeliveryUnit

ForwardSortationArea ::= Letter Digit Letter

LocalDeliveryUnit ::= Digit Letter Digit

Space ::= “ ”

**Letter ::= “A” | “B” | “C” | “E” | “G” | “H” | “J” | “K” | “L” | “M” | “N” | “P”
| “R” | “S” | “T” | “V” | “W” | “X” | “Y” | “Z”**

Digit ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

Syntax-based Coverage Criteria

- The most common and straightforward use every terminal and every production at least once

Terminal Symbol Coverage (TSC) : TR contains each terminal symbol t in the grammar G .

Production Coverage (PDC) : TR contains each production p in the grammar G .

- PDC subsumes TSC

Exercise 2

- Provide a set of test inputs based on the grammar below such that
 - TSC is satisfied
 - PDC is satisfied

```
Stream ::= action*
action ::= actG | actB
actG   ::= "G" s n
actB   ::= "B" t n
s       ::= digit1-3
t       ::= digit1-3
n       ::= digit2 "." digit2 "." digit2
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |
          "7" | "8" | "9"
```

Exercise 2

- Provide a set of test inputs based on the grammar below such that
 - TSC is satisfied (There are 13 Terminal symbols: G, B, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, all of which should take place in the test cases)
 - PDC is satisfied (There are 18 productions, all of which must be exercised)
 - Note that the ‘|’ symbol adds productions, so "action" has two productions and "digit" has 10.

```
Stream ::= action*
action ::= actG | actB
actG   ::= "G" s n
actB   ::= "B" t n
s       ::= digit1-3
t       ::= digit1-3
n       ::= digit2 “.” digit2 “.” digit2
digit  ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” |
          “7” | “8” | “9”
```

Mutation Testing

- Grammars describe both valid and invalid strings
- Both types can be produced as mutants
- A mutant is a variation of a valid string
 - Mutants may be valid or invalid strings
- Mutation is based on “mutation operators” and “ground strings”

Mutation Testing

- **Ground string**: A string in the grammar
 - The term “ground” is used as a reference to algebraic ground terms
- **Mutation Operator** : A rule that specifies syntactic variations of strings generated from a grammar
- **Mutant** : The result of one application of a mutation operator
 - A mutant is a string

Mutants and Ground Strings

- The key to mutation testing is the **design** of the mutation operators
 - Well designed **operators** lead to powerful testing
- Sometimes **mutant strings** are based on ground strings
- Sometimes they are derived directly **from the grammar**
 - **Ground** strings are used for **valid** tests
 - **Invalid** tests do not need ground strings

Valid Mutants

Ground Strings

G 20 08.01.90

B 16 06.27.94

Mutants

B 20 08.01.90

B 45 06.27.94

Invalid Mutants

7 20 08.01.90

B 134 06.27.1

Syntax-based Coverage Criteria

- When creating invalid strings, we just apply the operators
- This results in two simple criteria
- It makes sense to either use every operator once or every production once

Mutation Operator Coverage (MOC) : For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.

Mutation Production Coverage (MPC) : For each mutation operator, TR contains several requirements, to create one mutated string m that includes every production that can be mutated by that operator.

Example

Stream ::= action*

action ::= actG | actB

actG ::= "G" s n

actB ::= "B" t n

s ::= digit¹⁻³

t ::= digit¹⁻³

n ::= digit² "." digit² "." digit²

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Grammar

Ground String

G 20 08.01.90

B 16 06.27.94

Mutation Operators

- Exchange *actG* and *actB*
- Replace digits with other digits

Mutants using MOC

B 20 08.01.90

B 19 06.27.94

Mutants using MPC

G 20 08.01.90 G 16 06.27.94

G 20 08.01.90 B 11 06.27.94

G 30 08.01.90 B 13 06.27.94

G 40 08.01.90 B 14 06.27.94

G 50 08.01.90 B 15 06.27.94

...

...

Mutation Testing

- The number of test requirements for mutation depends on two things
 - The syntax of the artifact being mutated
 - The mutation operators
- Mutation testing is very difficult to apply **by hand**
- Mutation testing is very effective – considered the “**gold standard**” of testing
- Mutation testing is often used to **evaluate other criteria**

Introduction to Software Testing Chapter 9.2 Program-based Grammars

Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

Applying Syntax-based Testing to Programs

- Syntax-based criteria originated with programs and have been used mostly with programs
- BNF criteria are most commonly used to test compilers
- Mutation testing criteria are most commonly used for unit testing and integration testing of classes

BNF Testing for Compilers (9.2.1)

- Testing compilers is very complicated
 - Millions of correct programs !
 - Compilers must recognize and reject incorrect programs
- BNF criteria can be used to generate programs to test all language features that compilers must process
- This is a very specialized application and not discussed in detail

Program-based Grammars (9.2.2)

- The original and most widely known application of syntax-based testing is to **modify programs**
- **Operators** modify a **ground string** (program under test) to create **mutant programs**
- Mutant programs must compile correctly (**valid strings**)
- Mutants are **not tests**, but used to find tests
- Once mutants are defined, **tests** must be found to cause mutants to fail when executed
- This is called “**killing mutants**”

Killing Mutants

Given a mutant $m \in M$ for a ground string program P and a test t , t is said to **kill m if and only if the output of t on P is different from the output of t on m .**

- If mutation operators are designed well, the resulting tests will be very powerful
- Different operators must be defined for different programming languages and different goals
- Testers can keep adding tests until all mutants have been killed
 - *Dead mutant* : A test case has killed it
 - *Stillborn mutant* : Syntactically illegal
 - *Trivial mutant* : Almost every test can kill it
 - *Equivalent mutant* : No test can kill it (same behavior as original)

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    if (B < A)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

6 mutants

Each represents a separate program

With Embedded Mutants

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    Δ 1 minVal = B;
    if (B < A)
    Δ 2 if (B > A)
    Δ 3 if (B < minVal)
    {
        minVal = B;
        Δ 4 Bomb ();
        Δ 5 minVal = A;
        Δ 6 minVal = failOnZero (B);
    }
    return (minVal);
} // end Min
```

Replace one variable with another

Replaces operator

Immediate runtime failure ... if reached

Immediate runtime failure if $B==0$, else does nothing

Syntax-Based Coverage Criteria

Mutation Coverage (MC) : For each $m \in M$, TR contains exactly one requirement, to kill m .

- The RIPR model:
 - *Reachability* : The test causes the **faulty statement** to be reached (in mutation – the **mutated statement**)
 - *Infection* : The test causes the faulty statement to result in an **incorrect state**
 - *Propagation* : The incorrect state **propagates** to **incorrect output**
 - *Revealability* : The tester must **observe** part of the **incorrect output**
- The RIPR model leads to two variants of mutation coverage ...

Syntax-Based Coverage Criteria

I) Strongly Killing Mutants:

Given a mutant $m \in M$ for a program P and a test t , t is said to **strongly kill** m if and only if the **output** of t on P is different from the output of t on m

2) Weakly Killing Mutants:

Given a mutant $m \in M$ that modifies a location l in a program P , and a test t , t is said to **weakly kill** m if and only if the **state** of the execution of P on t is different from the state of the execution of m on t immediately after l

- Weakly killing satisfies **reachability** and **infection**, but not **propagation**

Weak Mutation

Weak Mutation Coverage (WMC) : For each $m \in M$, TR contains exactly one requirement, to weakly kill m .

- “Weak mutation” is so named because it is easier to kill mutants under this assumption
- Weak mutation also requires less analysis
- A few mutants can be killed under weak mutation but not under strong mutation (no propagation)
- Studies have found that test sets that weakly kill all mutants also strongly kill most mutants

Weak Mutation Example

Mutant 1 in the Min() example is:

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
 $\Delta 1$  minVal = B;
    if (B < A)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

With one or two partners :

1. Find a test that **weakly kills** the mutant, but not strongly
2. Generalize : What **must be true** to weakly kill the mutant, but not strongly?
3. Try to write down the **conditions** needed to (i) **reach** the mutated statement, (ii) **infect** the program state, and (iii) **propagate** to output

Weak Mutation Example

```
minVal = A;  
Δ 1 minVal = B;  
if (B < A)  
    minVal = B;
```

- I. Find a test that **weakly kills** the mutant, but not strongly

A = 5, B = 3

2. Generalize : What **must be true** to weakly kill the mutant, but not strongly?

B < A // minVal is set to B on for both

3. RIP **conditions**

Reachability : true // we always reach

Infection : $A \neq B$ // minVal has a different value

Propagation : $(B < A) = \text{false}$ // Take a different branch

Equivalent Mutation Example

Mutant 3 in the Min() example is equivalent:

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    if (B < A)
        Δ 3 if (B < minVal)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

With one or two partners

1. Convince yourselves that this mutant is equivalent

2. Briefly explain why

3. Try to prove the equivalence

Hint : Think about what must be true to kill the mutant

Equivalent Mutation Example

```
minVal = A;
```

```
if (B < A)
```

Δ 3 if (B < minVal)

1. Convince yourselves that this mutant is equivalent
2. Briefly explain why

A and minVal have the same value at the mutated statement

3. Try to prove the equivalence

Hint : Think about what must be true to kill the mutant

Infection : $(B < A) \neq (B < \text{minVal})$

Previous statement : $\text{minVal} = A$

Substitute : $(B < A) \neq (B < A)$

Contradiction ... therefore, equivalent

Strong Versus Weak Mutation

```
1 boolean isEven (int X)
2 {
3     if (X < 0)
4         X = 0 - X;
Δ 4     X = 0;
5     if (double) (X/2) == ((double) X) / 2.0
6         return (true);
7     else
8         return (false);
9 }
```

Reachability : $X < 0$

Infection : $X \neq 0$

$(X = -6)$ will kill mutant 4 under weak mutation

Propagation :

$((double) ((0-X)/2) == ((double) 0-X) / 2.0)$

$\neq ((double) (0/2) == ((double) 0) / 2.0)$

That is, X is not even ...

Thus $(X = -6)$ does not kill the mutant under strong mutation

Why Mutation Works

Fundamental Premise of Mutation Testing

If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

- This is not an absolute !
- The mutants guide the tester to an effective set of tests
- A very challenging problem :
 - Find a fault and a set of mutation-adequate tests that do not find the fault
- Of course, this depends on the mutation operators ...

Designing Mutation Operators

- At the **method level**, mutation operators for different programming languages are similar
- Mutation operators do one of **two things** :
 - Mimic typical programmer **mistakes** (incorrect variable name)
 - Encourage common test **heuristics** (cause expressions to be 0)
- Researchers design lots of operators, then experimentally **select** the most useful

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, \dots\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an **effective set of mutation operators**

Mutation Operators for Java

1. ABS — Absolute Value Insertion
2. AOR — Arithmetic Operator Replacement
3. ROR — Relational Operator Replacement
4. COR — Conditional Operator Replacement
5. SOR — Shift Operator Replacement
6. LOR — Logical Operator Replacement
7. ASR — Assignment Operator Replacement
8. UOI — Unary Operator Insertion
9. UOD — Unary Operator Deletion
10. SVR — Scalar Variable Replacement
- II. BSR — Bomb Statement Replacement

Full
definitions ...

Mutation Operators for Java

1. ABS — Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions `abs()`, `negAbs()`, and `failOnZero()`.

Examples:

$a = m * (o + p);$

$\Delta 1 \quad a = abs(m * (o + p));$

$\Delta 2 \quad a = m * abs((o + p));$

$\Delta 3 \quad a = failOnZero(m * (o + p));$

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators `+`, `-`, `*`, `/`, and `%` is replaced by each of the other operators. In addition, each is replaced by the special mutation operators `leftOp`, and `rightOp`.

Examples:

$a = m * (o + p);$

$\Delta 1 \quad a = m + (o + p);$

$\Delta 2 \quad a = m * (o * p);$

$\Delta 3 \quad a = m \leftarrow leftOp(o + p);$

Mutation Operators for Java (2)

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators ($<$, \leq , $>$, \geq , $=$, \neq) is replaced by each of the other operators and by *falseOp* and *trueOp*.

Examples:

```
if (X <= Y)
Δ1 if (X > Y)
Δ2 if (X < Y)
Δ3 if (X falseOp Y) // always returns false
```

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - $\&\&$, or - $\|$, and with no conditional evaluation - $\&$, or with no conditional evaluation - $|$, not equivalent - \wedge) is replaced by each of the other operators; in addition, each is replaced by *falseOp*, *trueOp*, *leftOp*, and *rightOp*.

Examples:

```
if (X <= Y && a > 0)
Δ1 if (X <= Y || a > 0)
Δ2 if (X <= Y leftOp a > 0) // returns result of left clause
```

Mutation Operators for Java (4)

5. SOR — Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.

Examples:

```
byte b = (byte) 16;  
b = b >> 2;  
Δ1 b = b << 2;  
Δ2 b = b /leftOp 2; // result is b
```

6. LOR — Logical Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by *leftOp* and *rightOp*.

Examples:

```
int a = 60;    int b = 13;  
int c = a & b;  
Δ1 int c = a | b;  
Δ2 int c = a /rightOp b; // result is b
```

Mutation Operators for Java (5)

7.ASR — Assignment Operator Replacement:

Each occurrence of one of the assignment operators ($+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $|=$, $\wedge=$, $<<=$, $>>=$, $>>>=$) is replaced by each of the other operators.

Examples:

$a = m * (o + p);$
Δ1 $a += m * (o + p);$
Δ2 $a *= m * (o + p);$

8.UOI — Unary Operator Insertion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

Examples:

$a = m * (o + p);$
Δ1 $a = m * -(o + p);$
Δ2 $a = -(m * (o + p));$

Mutation Operators for Java (6)

9. UOD — *Unary Operator Deletion:*

Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Examples:

- if !(X <= Y && !Z)
- Δ1 if (X > Y && !Z)
- Δ2 if !(X < Y && Z)

10. SVR — *Scalar Variable Replacement:*

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

Examples:

- a = m * (o + p);
- Δ 1 a = o * (o + p);
- Δ 2 a = m * (m + p);
- Δ 3 a = m * (o + o);
- Δ 4 p = m * (o + p);

Mutation Operators for Java (7)

II. BSR — *Bomb Statement Replacement*:

Each statement is replaced by a special `Bomb()` function.

Example:

`a = m * (o + p);`

$\Delta 1$ `Bomb() // Raises exception when reached`

Mutation Operators

- Mutation operators exist for several languages
 - Several programming languages (*Fortran, Lisp, Ada, C, C++, Java*)
 - Specification languages (*SMV, Z, Object-Z, algebraic specs*)
 - Modeling languages (*Statecharts, activity diagrams*)
 - Input grammars (*XML, SQL, HTML*)

Summary : Subsuming Other Criteria

- Mutation is widely considered the strongest test criterion
 - And most expensive !
 - By far the most test requirements (each mutant)
 - Usually the most tests
- Mutation subsumes other criteria by including specific mutation operators
- Subsumption can only be defined for weak mutation – other criteria only impose local requirements
 - Node coverage, Edge coverage, Clause coverage
 - General active clause coverage: Yes–Requirement on single tests
 - Correlated active clause coverage: No–Requirement on test pairs
 - All-defs data flow coverage

Introduction to Software Testing

Chapter 9.3

Integration and Object- Oriented Testing

Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

Integration and OO Testing

Integration Testing

Testing connections among separate program units

- In Java, testing the way **classes**, **packages** and **components** are connected
 - “*Component*” is used as a generic term
- This tests **features** that are unique to object-oriented programming languages
 - Inheritance, polymorphism and dynamic binding
- Integration testing is often based on **couplings** – the explicit and implicit relationships among software components

Integration Mutation (9.3.2)

- Faults related to component integration often depend on a mismatch of assumptions
 - Callee thought a list was sorted, caller did not
 - Callee thought all fields were initialized, caller only initialized some of the fields
 - Caller sent values in kilometers, callee thought they were miles
- Integration mutation focuses on mutating the connections between components
 - Sometimes called “*interface mutation*”
 - Both caller and callee methods are considered

Four Types of Mutation Operators

- Change a **calling** method by **modifying values** that are sent to a **called** method
- Change a **calling** method by **modifying the call**
- Change a **called** method by **modifying values** that enter and **leave** a method
 - Includes parameters as well as variables from higher scopes (class level, package, public, etc.)
- Change a **called** method by **modifying return statements** from the method

Five Integration Mutation Operators

1. IPVR — *Integration Parameter Variable Replacement*

Each parameter in a method call is replaced by each other variable in the scope of the method call that is of compatible type

- This operator replaces primitive type variables as well as object.

Example

```
MyObject a, b;  
    . . .  
callMethod (a);  
△ callMethod (b);
```

Five Integration Mutation Operators (2)

2. I_{UOI} — *Integration Unary Operator Insertion*

Each expression in a method call is modified by inserting all possible unary operators in front and behind it

- The unary operators vary by language and type

Example

```
callMethod (a);  
△ callMethod (a++);  
△ callMethod (++a);  
△ callMethod (a--);  
△ callMethod (--a);
```

Five Integration Mutation Operators (3)

3. IPEX — Integration Parameter Exchange

Each parameter in a method call is exchanged with each parameter of compatible types in that method call

- $\text{max}(a, b)$ is mutated to $\text{max}(b, a)$

Example

Max (a, b);
Δ **Max (b, a);**

Five Integration Mutation Operators (4)

4. IMCD — *Integration Method Call Deletion*

Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value

- Method calls that return objects are replaced with calls to “new ()”

Example

```
X = Max (a, b);  
Δ X = new Integer (0);
```

Five Integration Mutation Operators (5)

5. IREM — *Integration Return Expression Modification*

Each expression in each return statement in a method is modified by applying the UOI and AOR operators

Example

```
int myMethod ()  
{  
    return a + b;  
    Δ return ++a + b;  
    Δ return a - b;  
}
```

Object-Oriented Mutation

Testing Levels

intra-method

inter-method

intra-class

inter-class

integration mutation operators

- These five operators can be applied to **non-OO** languages
 - C, Pascal, Ada, Fortran, ...
- They do **not support** object oriented features
 - Inheritance, polymorphism, dynamic binding
- Two other language features that are often lumped with OO features are **information hiding (encapsulation)** and **overloading**
- Even experienced programmers often get encapsulation and access control wrong

Encapsulation, Information Hiding and Access Control

- *Encapsulation* : An abstraction mechanism to implement information hiding, which is a design technique that attempts to protect parts of the design from parts of the implementation
 - Objects can restrict access to their member variables and methods
- Java provides four access levels (C++ & C# are similar)
 - private
 - protected
 - public
 - default (also called package)
- Often **not used correctly** or understood, especially for programmers who are not well educated in **design**

Access Control in Java

Specifier	Same class	Same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

- Most class variables should be **private**
- **Public** variables should seldom be used
- **Protected** variables are particularly **dangerous** – future programmers can accidentally override (by using the same name) or accidentally use (by mis-typing a similar name)
 - They should be called “unprotected”

OO Language Features (Java)

- Method overriding

Allows a method in a subclass to have the same name, arguments and result type as a method in its parent

- Variable hiding

Achieved by defining a variable in a child class that has the same name and type of an inherited variable

- Class constructors

Not inherited in the same way other methods are – must be explicitly called

- Each object has ...

- A *declared type* : *Parent P*;
- An *actual type* : *P = new Child ()*; or assignment : *P = Pold*;
- Declared and actual types allow uses of the same name to reference **different variables with different types**

OO Language Feature Terms

- *Polymorphic attribute*
 - An object reference that can take on various types
 - Type the object reference takes on during execution can change
- *Polymorphic method*
 - Can accept parameters of different types because it has a parameter that is declared of type Object
- *Overloading*
 - Using the same name for different constructors or methods in the same class
- *Overriding*
 - A child class declares an object or method with a name that is already declared in an ancestor class
 - Easily confused with overloading because the two mechanisms have similar names and semantics
 - Overloading is in the same class, overriding is between a class and a descendant

More OO Language Feature Terms

- Members associated with a class are called **class** or **instance variables** and methods
 - **Static methods** can operate only on static variables; not instance variables
 - **Instance variables** are declared at the class level and are available to objects
- 20 object-oriented mutation operators **defined for Java – muJava**
- Broken into 4 general categories

Class Mutation Operators for Java

(1) Encapsulation

AMC

(2) Inheritance

IHI, IHD, IOD, IOP, IOR, ISI, ISD, IPC

(3) Polymorphism

PNC, PMD, PPD, PCI, PCD, PCC, PRV, OMR, OMD, OAC

(4) Java-Specific

JTI, JTD, JSI, JSD, JID, JDC

OO Mutation Operators— *Encapsulation*

I.AMC — Access Modifier Change

The access level for each instance variable and method is changed to other access levels

Example

point	
	private int x;
Δ1	public int x;
Δ2	protected int x;
Δ3	int x;

Class Mutation Operators for Java

(1) Encapsulation

(2) Inheritance

IHI, IHD, IOD, IOP, IOR, ISI, ISD, IPC

PNC, PMD, PPD, PCI, PCD, PCC, PRV, OMR, OMD, OAC

(4) Java-Specific

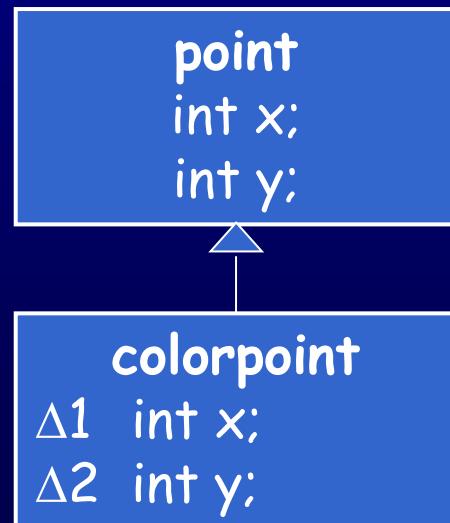
JTI, JTD, JSI, JSD, JID, JDC

OO Mutation Operators— *Inheritance*

2. IVI — *Hiding Variable Insertion*

A declaration is added to hide the declaration of each variable declared in an ancestor

Example

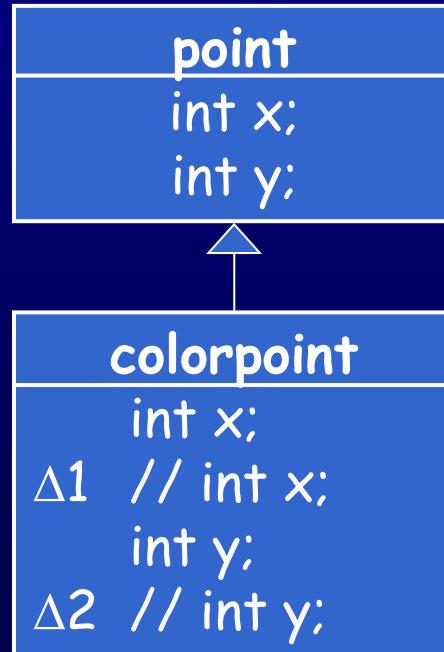


OO Mutation Operators— *Inheritance*

3. IVD — *Hiding Variable Deletion*

Each declaration of an overriding or hiding variable is deleted

Example

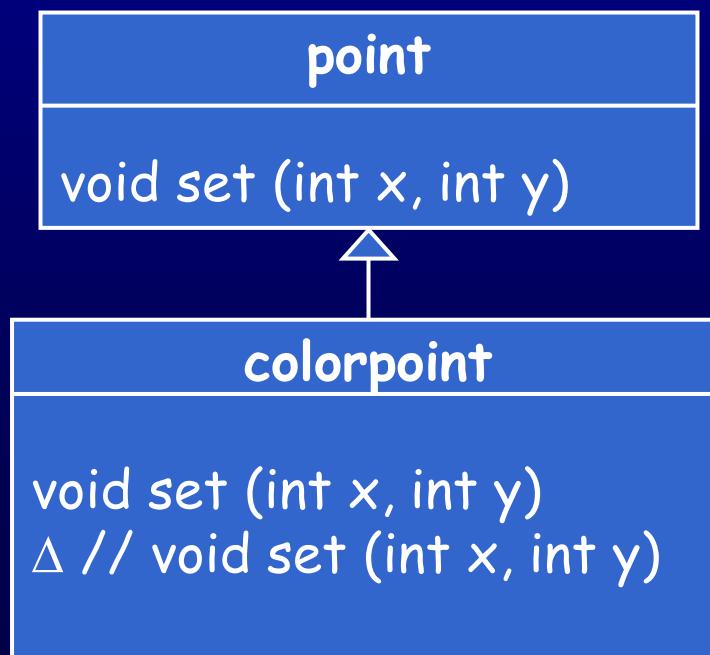


OO Mutation Operators— *Inheritance*

4. IOD — *Overriding Method Deletion*

Each entire declaration of an overriding method is deleted

Example

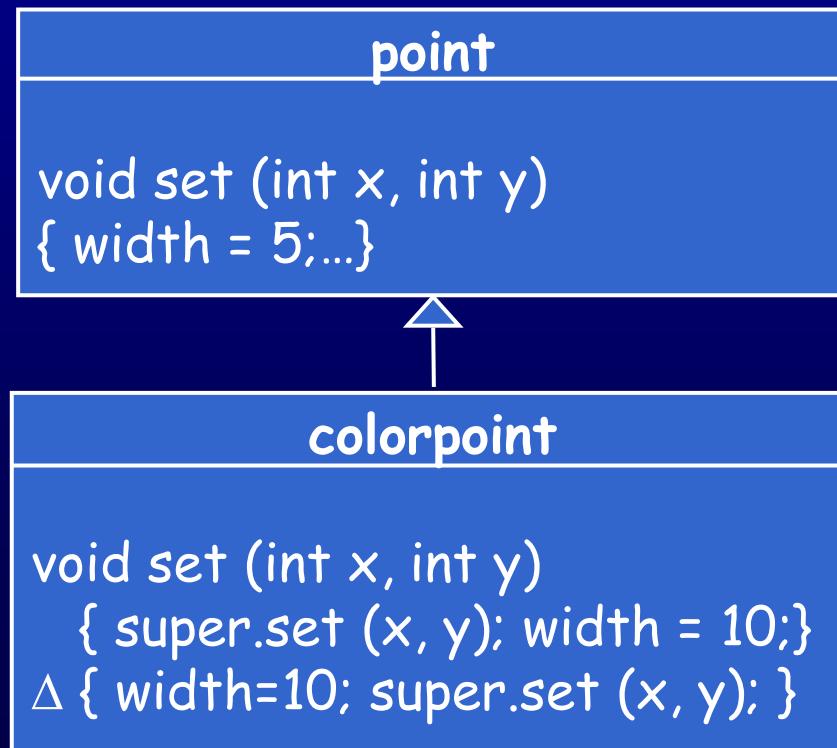


OO Mutation Operators— *Inheritance*

5. IOP — Overridden Method Calling Position Change

Each call to an overridden method is moved to the first and last statements of the method and up and down one statement

Example



OO Mutation Operators— *Inheritance*

6. IOR — Overridden Method Rename

Renames the parent's versions of methods that are overridden in a subclass so that the overriding does not affect the parent's method

Example

```
point
...
void set (int x, int y)
Δ void setP (int x, int y)
...
void setDimension (int d)
{
    ...
    set (x, y);
    Δ setP (x, y);
    ...
}
```

```
point p;
p = new colorpoint ();
...
p.set (1, 2);
p.setDimension (3);
```

colorpoint

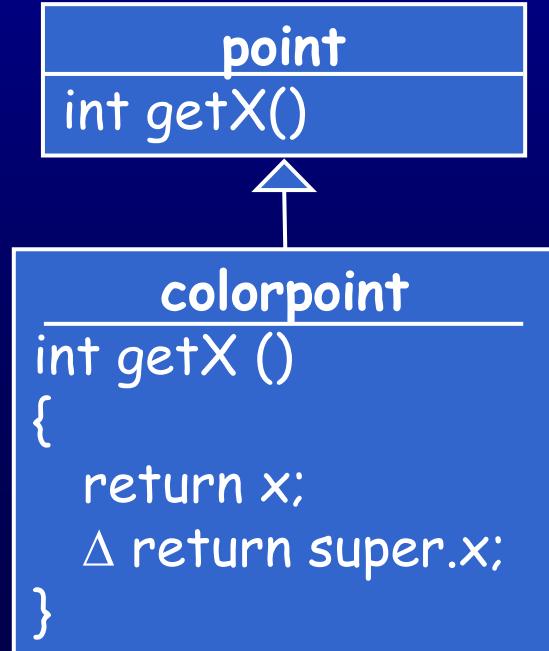
```
...
void set (int x, int y)
```

OO Mutation Operators— *Inheritance*

7. ISI — Super Keyword Insertion

Inserts the `super` keyword before overriding variables or methods (if the name is also defined in an ancestor class)

Example

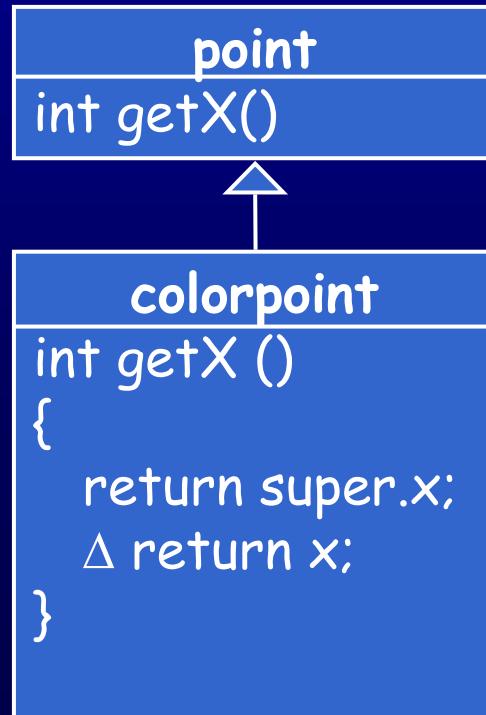


OO Mutation Operators— *Inheritance*

8. ISD — Super Keyword Deletion

Delete each occurrence of the `super` keyword

Example



OO Mutation Operators— *Inheritance*

9. IPC — *Explicit Parent Constructor Deletion*

Each call to a **super** constructor is deleted

Example

```
point
point (int x, int y)
...
```



```
colorpoint
colorpoint (int x, int y, int color)
{
    super (x, y);
    Δ // super (x, y);
    ...
}
```

Class Mutation Operators for Java

(1) Encapsulation

AMC

(2) Inheritance

(3) Polymorphism

ISI, ISD, IPC

PNC, PMD, PPD, PCI, PCD, PCC, PRV, OMR, OMD, OAC

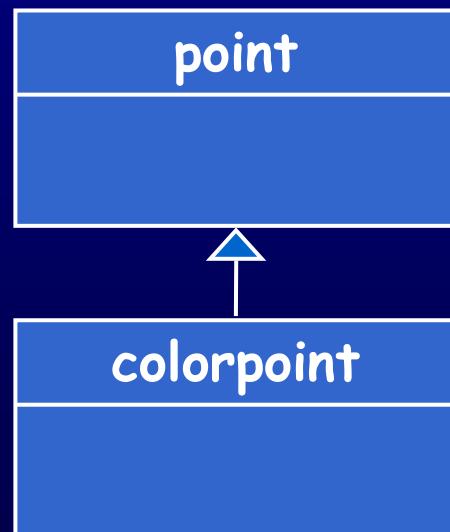
JTI, JTD, JSI, JSD, JID, JDC

OO Mutation Operators— *Polymorphism*

10. PNC — *new Method Call With Child Class Type*

The actual type of a new object is changed in the `new()` statement

Example



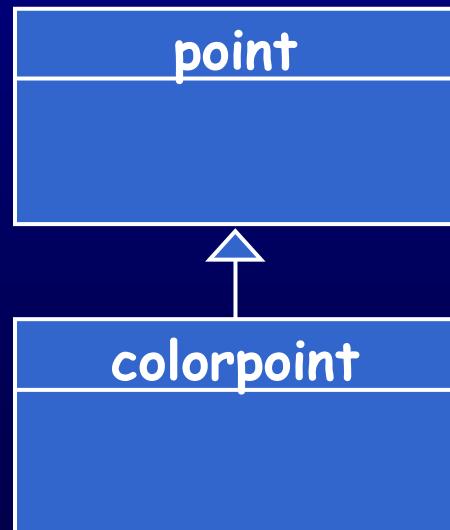
```
point p;  
p = new point ();  
Δ p = new colorpoint ();
```

OO Mutation Operators— *Polymorphism*

II. PMD — Member Variable Declaration with Parent Class Type

The declared type of each new object is changed in the declaration

Example



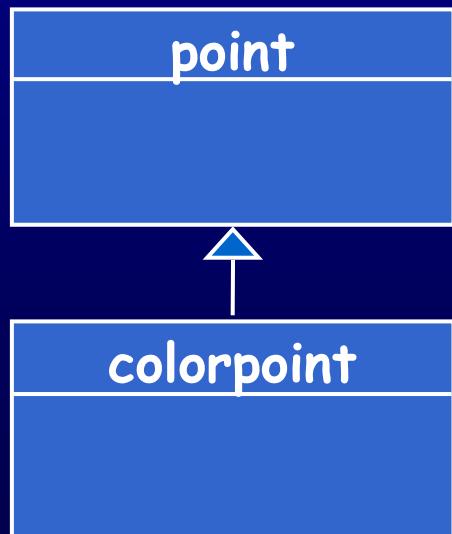
```
point p;  
Δ colorpoint p;  
p = new colorpoint();
```

OO Mutation Operators— *Polymorphism*

12. PPD — Parameter Variable Declaration with Child Class Type

The declared type of each parameter object is changed in the declaration

Example



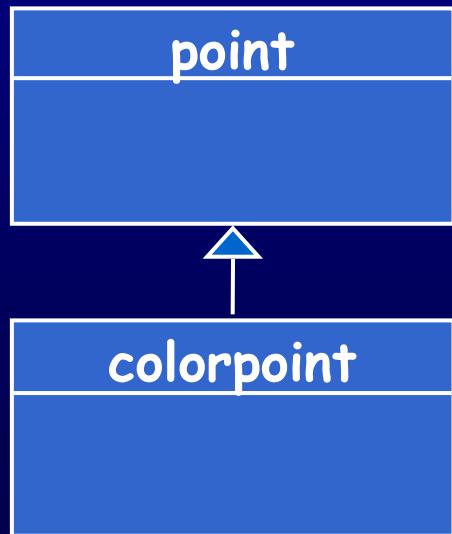
```
boolean equals (point p)  
{ ... }  
△ boolean equals (colorpoint p)  
{ ... }
```

OO Mutation Operators— *Polymorphism*

I3. PCI — *Type Cast Operator Insertion*

The actual type of an object reference is changed to the parent or to the child of the original declared type

Example



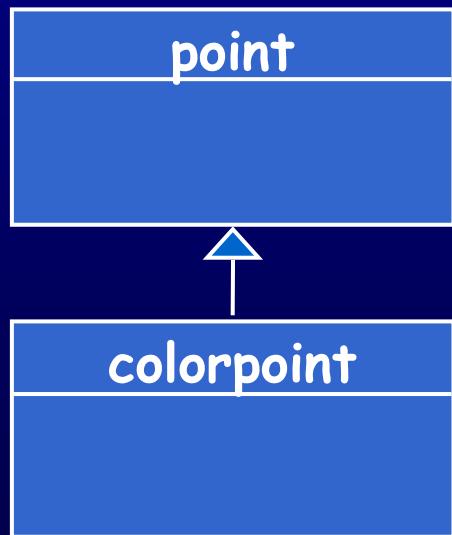
```
point p;  
p = new colorpoint ();  
int x = p.getX ();  
Δ int x = (point) p.getX ();
```

OO Mutation Operators— *Polymorphism*

I4. PCD — *Type Cast Operator Deletion*

Type casting operators are deleted

Example



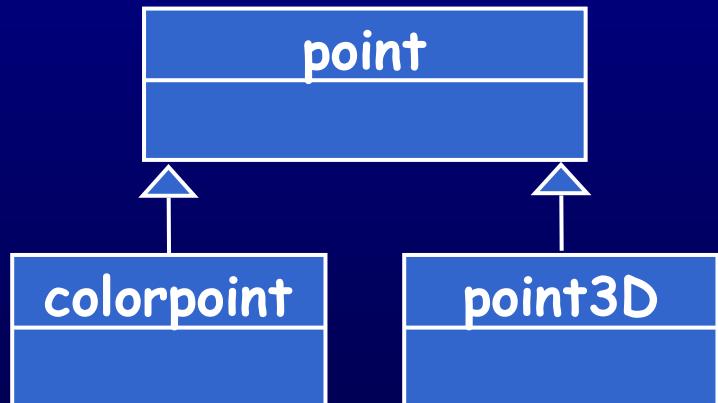
```
point p;  
p = new colorpoint ();  
int x = (point) p.getX ();  
Δ int x = p.getX ();
```

OO Mutation Operators— *Polymorphism*

15. PPC — *Cast Type Change*

Changes the type to which an object reference is being cast

Example



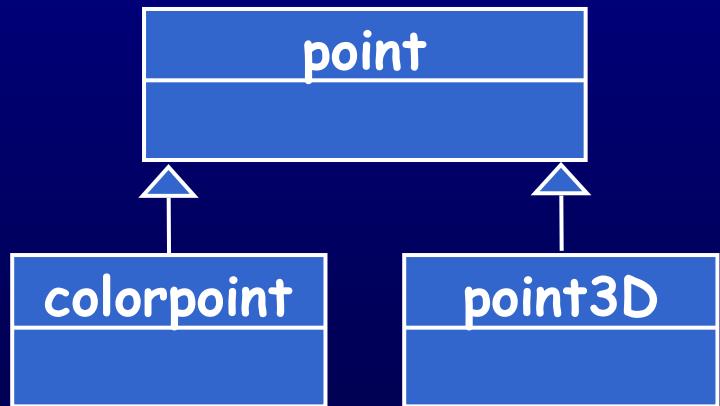
```
point p = new point (0, 0);
int x = (colorpoint) p.getX ();
Δ int x = (point3D) p.getX ();
```

OO Mutation Operators— *Polymorphism*

16. PRV — Reference Assignment with Other Compatible Type

The right side objects of assignment statements are changed to refer to objects of a compatible type

Example



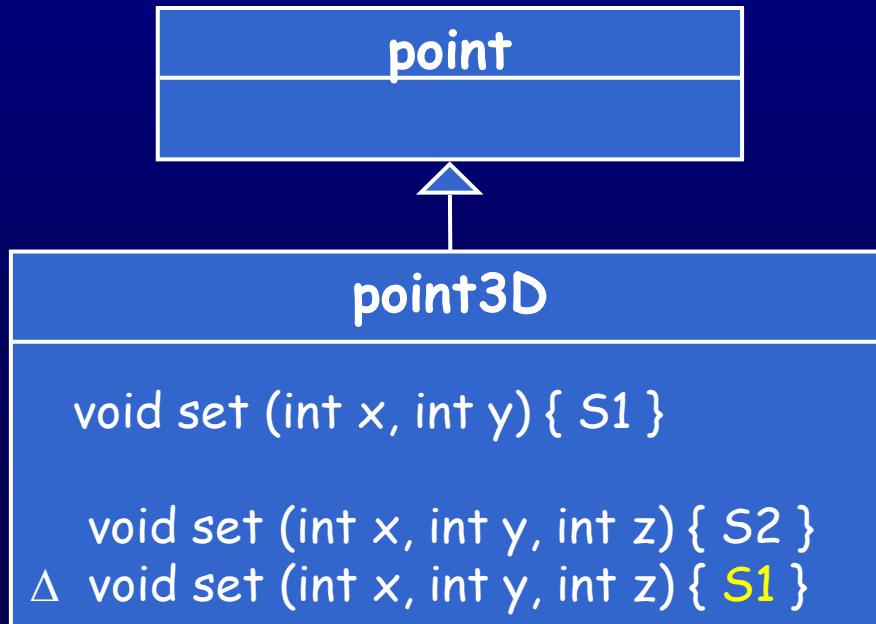
```
point p;  
colorpoint cp = new colorpoint (0, 0);  
point3D p3d = new point3D (0, 0, 0);  
p = cp;  
Δ p = p3d;
```

OO Mutation Operators— *Polymorphism*

17. OMR — Overloading Method Contents Replace

For each pair of methods that have the same name, the bodies are interchanged

Example

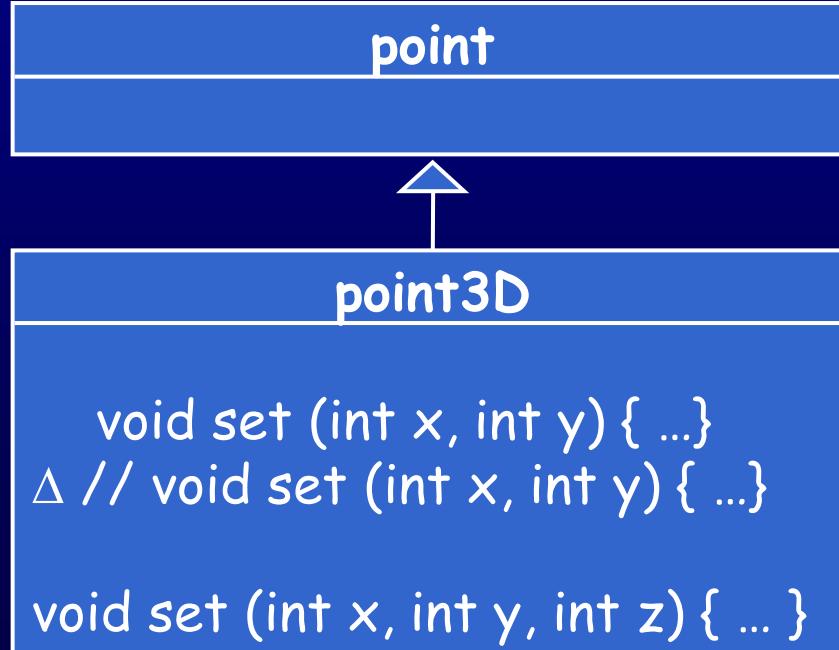


OO Mutation Operators— *Polymorphism*

18. OMD — Overloading Method Deletion

Each overloaded method declaration is deleted, one at a time

Example

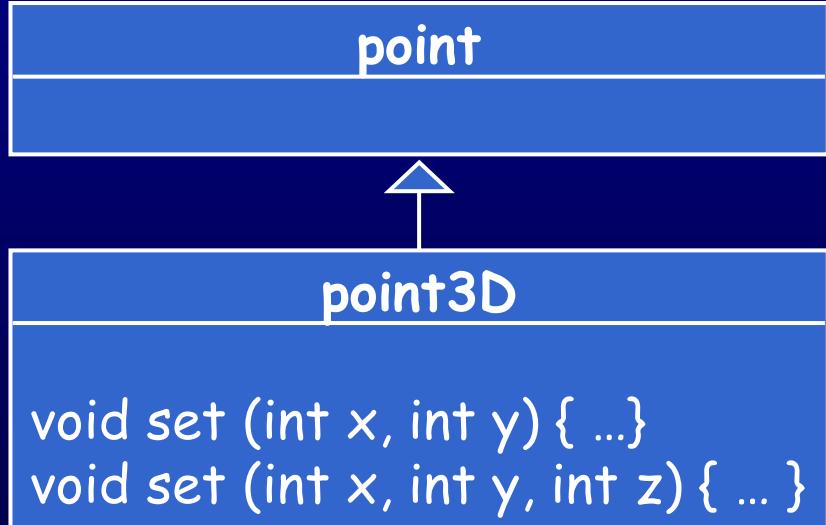


OO Mutation Operators— *Polymorphism*

19. OAC — Arguments of Overloading Method Call Change

The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists

Example



```
point p = new point3D ();
p.set (5, 7, 9);
△ p.set (5, 7);
```

Class Mutation Operators for Java

(1) Encapsulation

AMC

(2) Inheritance

IHI, IHD, IOD, IOP, IOR, ISI, ISD, IPC

(3) Polymorphism

(4) Java-Specific

PCD PCC PRV OMR OMD OAC

JTI, JTD, JSI, JSD, JID, JDC

OO Mutation Operators—*Language Specific*

20. JTI — *this Keyword Insertion*

The keyword **this** is inserted whenever possible

Example

```
point  
...  
void set (int x, int y)  
{  
    x = x;  
    Δ1 this.x = x;  
    y = y;  
    Δ2 this.y = y;  
}  
...
```

OO Mutation Operators—*Language Specific*

21. JTD — *this* Keyword Deletion

The keyword **this** is deleted whenever possible

Example

```
point  
...  
void set (int x, int y)  
{  
    this.x = x;  
     $\Delta 1$  x = x;  
    this.y = y;  
     $\Delta 2$  y = y;  
}  
...
```

OO Mutation Operators—*Language Specific*

22. JSI — *Static Modifier Insertion*

The **static** modifier is added to instance variables

Example

point

```
public int x = 0;  
Δ1 public static int x = 0;  
public int y = 0;  
Δ2 public static int y = 0;
```

...

OO Mutation Operators—*Language Specific*

23. JSD — *Static Modifier Deletion*

Each instance of the **static** modifier is removed

Example

```
point  
public static int x = 0;  
Δ1 public int x = 0;  
    public static int Y = 0;  
Δ2 public int y = 0;  
  
...
```

OO Mutation Operators—*Language Specific*

24. JID — Member Variable Initialization Deletion

Remove initialization of each member variable

Example

point

```
int x = 5;  
Δ int x;  
...
```

OO Mutation Operators—*Language Specific*

25. JDC — Java-supported Default Constructor Deletion

Delete each declaration of default constructor (with no parameters)

Example

```
point
point() { ... }
Δ // point() { ... }
...
```

Class Mutation Operators for Java

(1) Encapsulation

AMC

(2) Inheritance

HVD, HVI, IOD, OMM, OMR, SKD, PCD

(3) Polymorphism

PNC, PMD, PPD, PCI, PCD, PCC, PRV, OMR, OMD, OAC

(4) Java-Specific

TKD, SMC, VID, DCD

Integration Mutation Summary

- Integration testing often looks at **couplings**
- We have not used **grammar testing** at the integration level
- Mutation testing modifies **callers** and **callees**
- **OO mutation** focuses on inheritance, polymorphism, dynamic binding, information hiding and overloading
 - The access levels make it easy to make mistakes in OO software
- **mujava** is an educational & research tool for mutation testing of Java programs
 - <http://cs.gmu.edu/~offutt/mujava/>