Testing

slides compiled from

Alex Aiken's,

Neelam Gupta's,

Tao Xie's.

Why Testing

- Researchers investigate many approaches to improving software quality

- But the world tests

- > 50% of the cost of software development is testing

- Testing is consistently a hot research topic

- **Test-input generation**
  - Specification/model-based: Korat (Marinov@UIUC), TestEra (Khurshid@UT Austin ECE), NASA Java Pathfinder (Visser@NASA), Spec#, AsmIT (MSR FSE)
  - Code-based: Rostra/Symstra (Xie@NCSU), JCrasher/CnC(Smaragdakis@GeorgiaTech), TGEN(Gupta@Arizona), Blast (Berkeley), SLAM (MSR)
- **In-field testing**
  - Residual testing (Young@Oregon), Gamma (Orso, Harrold@GeorgiaTech)
  - Skoll (Porter, Memon@Maryland, Schmidt@Vanderbilt)
- **Regression testing**
  - Regression test selection/prioritization (Rothermel, Elbaum@UNL, Porter@Maryland, Harrold@GIT)
  - Continuous testing (Ernst@MIT), Capture/Replay (Ernst@MIT, Orso@GeorgiaTech)
- **Testing various types of programs/based on different artifacts**
  - Testing GUI app- GUITAR (Memon@Maryland),
  - Testing database app- AGENDA (Frankl@Polytech), DIATOMS(Soffa@Virginia),
  - Testing spreadsheet app (Rothermel@UNL, Burnett@OregonStateU),
  - Testing aspect-oriented programs (Xie@NCSU, Zhao@SHJT)
  - Testing web app (Offut@GMU, Pollock@Delaware)
  - Testing embedded systems (Soffa@Virginia)
  - Testing access-control policies (Xie@NCSU)
  - Testing firewall systems (Hoffman@Victoria)
  - Testing multithreading systems (Lei@UTArlington)
  - Architecture-based testing (Richardson@UC Irvine)
  - Security testing (McGraw@Cigital)
- **Dynamic property inference**
  - Daikon (Ernst), Spec mining (Bodik@Berkeley), Hastings-sequencing constraints (Lam@Stanford), Terracotta-temporal properties (Evans@Virigina), Algebraic spec inference (Diwan@Colorado), Statistical algebraic spec inference, Object state machines (Xie@NCSU)
- **Debugging**
  - Delta debugging (Zeller@Saarland)
  - Berkeley Bug isolation (Liblit@Wisconsin)
  - Fault localization (Harrold@GIT, Gupta@Arizona)

Note

- Fundamentally, seems to be not as deep
  - Over so many years, simple ideas still deliver the best performance
- Recent trends
  - Testing + XXX
  - DAIKON, CUTE
- Messages conveyed
  - Two messages and one conclusion.
- Difficuties
  - Beat simple ideas (sometimes hard)
  - Acquire a large test suite

Outline

- Testing practice
  - Goals: Understand current state of practice
    - Boring
    - But necessary for good science
  - Need to understand where we are, before we try to go somewhere else
- Testing Research
- Some textbook concepts

Testing Practice

Outline

- Manual testing
- Automated testing
- Regression testing
- Nightly build
- Code coverage
- Bug trends

Manual Testing

- Test cases are lists of instructions
  - "test scripts"

- Someone manually executes the script
  - Do each action, step-by-step
    - Click on "login"
    - Enter username and password
    - Click "OK"
    - ...
  - And manually records results

- Low-tech, simple to implement

Manual Testing

- Manual testing is very widespread
  - Probably not dominant, but very, very common

- Why? Because
  - Some tests can't be automated
    - Usability testing
  - Some tests shouldn't be automated
    - Not worth the cost

Manual Testing

- Those are the best reasons

- There are also not-so-good reasons
  - Not-so-good because innovation could remove them
  - Testers aren't skilled enough to handle automation
  - Automation tools are too hard to use
  - The cost of automating a test is 10X doing a manual test

Topics

- Manual testing
- Automated testing
- Regression testing
- Nightly build
- Code coverage
- Bug trends

Automated Testing

- Idea:
  - Record manual test
  - Play back on demand
- This doesn't work as well as expected
  - E.g., Some tests can't/shouldn't be automated

Fragility

- Test recording is usually very fragile
  - Breaks if environment changes anything
  - E.g., location, background color of textbox
- More generally, automation tools cannot generalize a test
  - They literally record exactly what happened
  - If anything changes, the test breaks
- A hidden strength of manual testing
  - Because people are doing the tests, ability to adapt tests to slightly modified situations is built-in

Breaking Tests

- When code evolves, tests break
  - E.g., change the name of a dialog box
  - Any test that depends on the name of that box breaks

- Maintaining tests is a lot of work
  - Broken tests must be fixed; this is expensive
  - Cost is proportional to the number of tests
  - Implies that more tests is not necessarily better

Improved Automated Testing

- Recorded tests are too low level
  - E.g., every test contains the name of the dialog box

- Need to abstract tests
  - Replace dialog box string by variable name X
  - Variable name X is maintained in one place
    - So that when the dialog box name changes, only X needs to be updated and all the tests work again

Data Driven Testing (for Web Applications)

- Build a database of event tuples

  < Document, Component, Action, Input, Result >

- E.g.,

  < LoginPage, Password, InputText, $password, “OK”>

- A test is a series of such events chained together

- Complete system will have many relations
  - As complicated as any large database

Discussion

- Testers have two jobs
  - Clarify the specification
  - Find (important) bugs
- Only the latter is subject to automation
- Helps explain why there is so much manual testing

Topics

- Manual testing
- Automated testing
- Regression testing
- Nightly build
- Code coverage
- Bug trends

Regression Testing

- Idea
  - When you find a bug,
  - Write a test that exhibits the bug,
  - And always run that test when the code changes,
  - So that the bug doesn't reappear

- Without regression testing, it is surprising how often old bugs reoccur

Regression Testing (Cont.)

- Regression testing ensures forward progress
  - We never go back to old bugs
- Regression testing can be manual or automatic
  - Ideally, run regressions after every change
  - To detect problems as quickly as possible
- But, regression testing is expensive
  - Limits how often it can be run in practice
  - Reducing cost is a long-standing research problem

Nightly Build

- Build and test the system regularly
  - Every night
- Why? Because it is easier to fix problems earlier than later
  - Easier to find the cause after one change than after 1,000 changes
  - Avoids new code from building on the buggy code
- Test is usually subset of full regression test
  - “smoke test”
  - Just make sure there is nothing horribly wrong

A Problem

- So far we have:

  Measure changes regularly (nightly build)

  Make monotonic progress (regression)

- How do we know when we are done?
  - Could keep going forever

- But, testing can only find bugs, not prove their absence
  - We need a proxy for the absence of bugs

Topics

- Manual testing
- Automated testing
- Regression testing
- Nightly build
- Code coverage
- Bug trends

Code Coverage

- Idea
  - Code that has never been executed likely has bugs

- This leads to the notion of *code coverage*
  - Divide a program into units (e.g., statements)
  - Define the coverage of a test suite to be

$$\frac{\text{\# of statements executed by suite}}{\text{\# of statements}}$$

Code Coverage (Cont.)

- Code coverage has proven value
  - It's a real metric, though far from perfect
- But 100% coverage does not mean no bugs
  - E.g., a bug visible after loop executes 1,025 times
- And 100% coverage is almost never achieved
  - Infeasible paths
  - Ships happen with < 60% coverage
  - High coverage may not even be desirable
    - May be better to devote more time to tricky parts with good coverage

Using Code Coverage

- Code coverage helps identify weak test suites

- Code coverage can't complain about missing code
  - But coverage can hint at missing cases
    - Areas of poor coverage $\Rightarrow$ areas where not enough thought has been given to specification

More on Coverage

- Statement coverage
- Edge coverage
- Path coverage
- Def-use coverage

Topics

- Manual testing
- Automated testing
- Regression testing
- Nightly build
- Code coverage
- Bug trends

Bug Trends

- Idea: Measure rate at which new bugs are found
- Rational: When this flattens out it means
  1. The cost/bug found is increasing dramatically
  2. There aren't many bugs left to find

The Big Picture

- Standard practice
  - *Measure progress often* (nightly builds)
  - *Make forward progress* (regression testing)
  - *Stopping condition* (coverage, bug trends)

Testing Research

Outline

- Overview of testing research
  - Definitions, goals
- Three topics
  - Random testing
  - Efficient regression testing
  - Mutation analysis

Overview

- Testing research has a long history
  - At least to the 1960′s

- Much work is focused on metrics
  - Assigning numbers to programs
  - Assigning numbers to test suites
  - Heavily influenced by industry practice

- More recent work focuses on deeper analysis
  - Semantic analysis, in the sense we understand it

What is a Good Test?

- Attempt 1:

If program P implements function F on domain D, then a test set $T \subseteq D$ is *reliable* if

$$(\forall t \in T.\ P(t) = F(t)) \Rightarrow \forall t \in D.\ P(t) = F(t)$$

- Says that a good test set is one that implies the program meets its specification

Good News/Bad News

- Good News
  - There are interesting examples of reliable test sets
  - Example: *A function that sorts N numbers using comparisons sorts correctly iff it sorts all inputs consisting of 0,1 correctly*
  - This is a finite reliable test set

- Bad News
  - There is no effective method for generating finite reliable test sets

An Aside

- It's clear that reliable test sets must be impossible to compute in general
- But most programs are not diagonalizing Turing machines . . .
- It ought to be possible to characterize finite reliable test sets for certain classes of programs

Adequacy

- Reliability is not useful if we don't have a full reliable test set
  - Then it is possible that the program passes every test, but is not the program we want

- A different definition

If program P implements function F on domain D, then a test set $T \subseteq D$ is *adequate* if

$$(\forall \text{progs } Q.\ Q(D) \neq F(D)) \Rightarrow \exists\, t \in T.\ Q(t) \neq F(t)$$

Adequacy

- Adequacy just says that the test suite must make every incorrect program fail
- This seems to be what we really want

Outline

- Overview of testing research
  - Definitions, goals
- Three topics
  - Random testing
  - Efficient regression testing
  - Mutation analysis

Random Testing

- About ¼ of Unix utilities crash when fed random input strings
  - Up to 100,000 characters
- What does this say about testing?
- What does this say about Unix?

What it Says About Testing

- Randomization is a highly effective technique
  - And we use very little of it in software
- "A random walk through the state space"
- To say anything rigorous, must be able to characterize the distribution of inputs
  - Easy for string utilities
  - Harder for systems with more arcane input
    - E.g., parsers for context-free grammars

What it Says About Unix

- What sort of bugs did they find?
  - Buffer overruns
  - Format string errors
  - Wild pointers/array out of bounds
  - Signed/unsigned characters
  - Failure to handle return codes
  - Race conditions

- Nearly all of these are problems with C!
  - Would disappear in Java
  - Exceptions are races & return codes

A Nice Bug

csh !0%8f

- ! is the history lookup operator
  - No command beginning with 0%8f
- csh passes an error "0%8f: Not found" to an error printing routine
- Which prints it with printf()

Outline

- Overview of testing research
  - Definitions, goals
- Three topics
  - Random testing
  - Efficient regression testing
  - Mutation analysis

Efficient Regression Testing

- Problem: Regression testing is expensive
- Observation: Changes don't affect every test
  - And tests that couldn't change need not be run
- Idea: Use a conservative static analysis to prune test suite

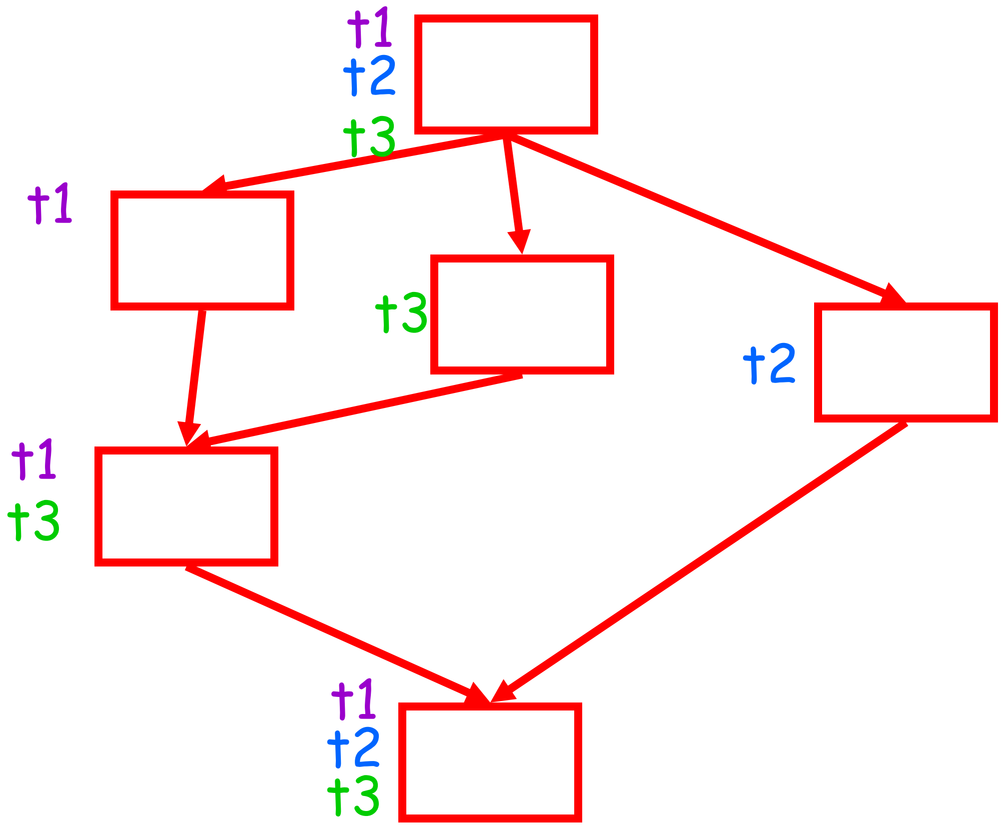
The Algorithm

Two pieces:

1. Run the tests and record for each basic block which tests reach that block

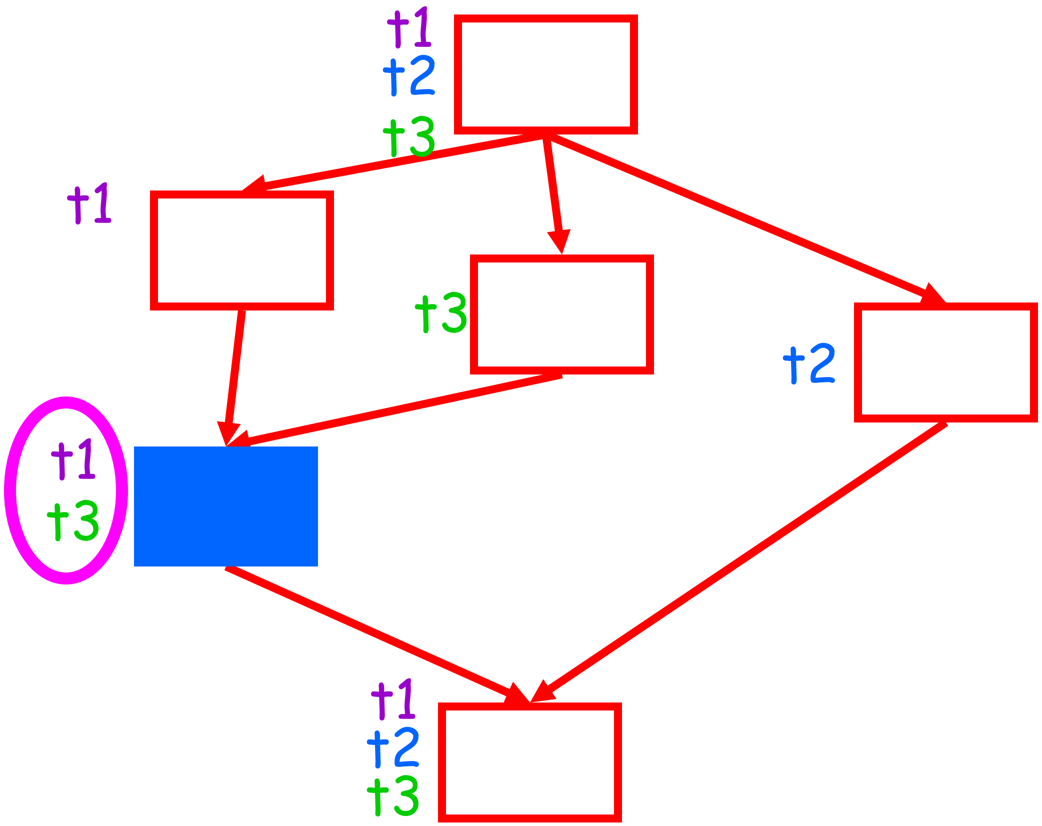
2. After modifications, do a DFS of the new control flow graph. Wherever it differs from the original control flow graph, run all tests that reach that point

Example



Label each node of the control flow graph with the set of tests that reach it.

Example (Cont.)



When a statement is modified, rerun just the tests reaching that statement

Experience

- This works
  - And it works better on larger programs
  - # of test cases to rerun reduced by > 90%
- Total cost less than cost of running all tests
  - Total cost = cost of tests run + cost of tool
- Impact analysis

Outline

- Overview of testing research
  - Definitions, goals
- Three topics
  - Random testing
  - Efficient regression testing
  - Mutation analysis

Adequacy (Review)

If program P implements function F on domain D, then a test set $T \subseteq D$ is *adequate* if

$$(\forall \text{progs } Q.\ Q(D) \neq F(D)) \Rightarrow \exists\, t \in T.\ Q(t) \neq F(t)$$

But we can't afford to quantify over all programs . . .

From Infinite to Finite

- We need to cut down the size of the problem
  - Check adequacy wrt a smaller set of programs
- Idea: Just check a finite number of (systematic) variations on the program
  - E.g., replace x > 0 by x < 0
  - Replace I by I+1, I-1
- This is *mutation analysis*

Mutation Analysis

- Modify (mutate) each statement in the program in finitely many different ways
- Each modification is one *mutant*
- Check for adequacy wrt the set of mutants
  - Find a set of test cases that distinguishes the program from the mutants

If program P implements function F on domain D, then a test set $T \subseteq D$ is *adequate* if

$$(\forall \text{mutants } Q.\ Q(D) \neq F(D)) \Rightarrow \exists\, t \in T.\ Q(t) \neq F(t)$$

What Justifies This?

- The "competent programmer assumption"
  The program is close to right to begin with

- It makes the infinite finite
  We will inevitably do this anyway; at least here it is clear what we are doing

- This already generalizes existing metrics
  If it is not the end of the road, at least it is a step forward

The Plan

- Generate mutants of program P

- Generate tests
  - By some process

- For each test t
  - For each mutant M
    - If $M(t) \neq P(t)$ mark M as *killed*

- If the tests kill all mutants, the tests are adequate

The Problem

- This is dreadfully slow
- Lots of mutants
- Lots of tests
- Running each mutant on each test is expensive
- But early efforts more or less did exactly this

Simplifications

- To make progress, we can either
  - Strengthen our algorithms
  - Weaken our problem

- To weaken the problem
  - Selective mutation
    - Don't try all of the mutants
  - Weak mutation
    - Check only that mutant produces different state after mutation, not different final output
    - 50% cheaper

Better Algorithms

- Observation: Mutants are nearly the same as the original program

- Idea: Compile one program that incorporates and checks all of the mutations simultaneously
  - A so-called *meta-mutant*

Metamutant with Weak Mutation

- Constructing a metamutant for weak mutation is straightforward

- A statement has a set of mutated statements
  - With any updates done to fresh variables

    $X := Y << 1$ $X_1 := Y << 2$ $X_2 := Y >> 1$
  - After statement, check to see if values differ

    X == X_1 X == X_2

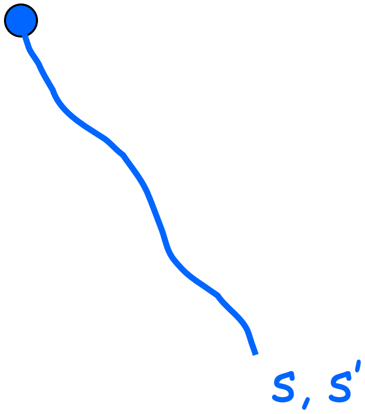
Comments

- A metamutant for weak mutation should be quite practical
  - Constant factor slowdown over original program
- Not clear how to build a metamutant for stronger mutation models

Generating Tests

- Mutation analysis seeks to generate adequate test sets automatically

- Must determine inputs such that
  - Mutated statement is reached
  - Mutated statement produces a result different from original



Automatic Test Generation

- This is not easy to do

- Approaches
  - Backward approach
    - Work backwards from statement to inputs
      - Take short paths through loops
    - Generate symbolic constraints on inputs that must be satisfied
    - Solve for inputs

Automatic Test Generation (Cont.)

- Work forwards from inputs
  - Symbolic execution (Tao)
  - Concrete execution (CUTE)
  - Arithmetic rep. based (gupta)

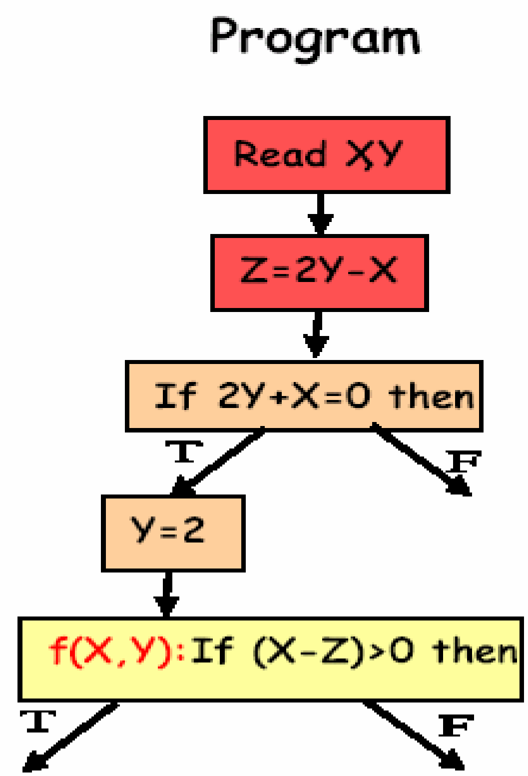
Linear arith. rep. of predicate function

- Let $f(X,Y) = a\,X + b\,Y + c$
- Execute program rep. of $f(X,Y)$ to compute $f(X_0,Y_0)$, $f(X_0+\Delta X,Y_0)$ & $f(X_0,Y_0+\Delta Y)$
- Compute a, b & c as follows:

  $a = (f(X_0+\Delta X,Y_0) - f(X_0,Y_0)) / \Delta X$

  $b = (f(X_0,Y_0+\Delta Y) - f(X_0,Y_0)) / \Delta Y$

  $c = f(X_0,Y_0) - a\,X_0 - b\,Y_0$



Comments on Test Generation

- Apparently works well for
  - Small programs
  - Without pointers
  - For certain classes of mutants

- So not very clear how well it works in general
  - Note: Solutions for pointers are proposed

A Problem

- What if a mutant is equivalent to the original?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
    - Try to prove program equivalence automatically
    - Often requires manual intervention
  - Undermines the metric
- How about more complicated mutants?

Opinions

- Mutation analysis is a good idea
  - For all the reasons cited before
  - Also technically interesting
  - And there is probably more to do . . .

- How important is automatic test generation?
  - Still must manually look at output of tests
    - This is a big chunk of the work, anyway
  - Weaken the problem
    - Directed ATG is a quite interesting direction to go.
  - Automatic tests likely to be weird
    - Both good and bad

Opinions

- Testing research community trying to learn
  - From programming languages community
    - Slicing, dataflow analysis, etc.
  - From theorem proving community
    - Verification conditions, model checking

Some Textbook Concepts

- About different levels of testing
  - System test, Model Test, Unit test, Integration Test
- Black box vs. White box
  - Functional vs. structural

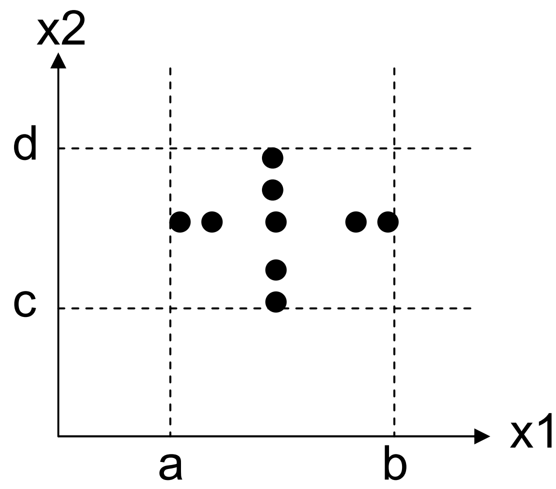
Boundary Value Test

Given F(x1,x2) with constraints

$$a \leq x1 \leq b$$
$$c \leq x2 \leq d$$

Boundary Value analysis focuses on the boundary of the input space to identify test cases.

Use input variable value at min, just above min, a nominal value, just above max, and at max.



Next Lecture

- Program Slicing