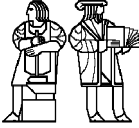



6.170 Lecture 5 Choosing Test Data



John Guttag
MIT EECS




Validation

Process designed to uncover problems, increase confidence
Combination of reasoning and test

Focus on testing for now

Today is only the first pass on this topic
Some things that may make your life easier
More on these topics later

©Srinivas Devadas/John Guttag Spring 2002 Slide 2




Testing

Three step process
Choose input data
Run Program
Examine results

First rule of testing
Do it early and and do it often

Second rule of testing
Be systematic

©Srinivas Devadas/John Guttag Spring 2002 Slide 3



Some Testing Buzzwords


Unit testing (today's topic)
Does a single module behave as it should

System (integration) testing (later in term)
Does things work when you put them together
Interface errors a common problem

Black box testing
Choose test data without looking at program
Specification play a major role

Glass box testing
Choose test data with knowledge of program

©Srinivas Devadas/John Guttag Spring 2002 Slide 4



What's So Hard About Testing ?


"just try it and see if it works..."

```
int procl(int x, int y, int z)
// requires: 1 <= x,y,z <= 1000
// effects: computes some f(x,y,z)
```

Exhaustive testing would require 1 billion runs!
Sounds totally impractical
Could see how input set size would get MUCH bigger

Key problem: choosing the test suite
Small enough to finish quickly
Large enough to validate the program

©Srinivas Devadas/John Guttag Spring 2002 Slide 5



Approach: Partition the Input Space

Key insight
Program small relative to input space
==> behavior is the "same" for sets of inputs

Ideal test suite:
Identify sets with same behavior
Try one input from each set

Two problems
Notion of the same behavior is subtle
Naive approach: execution equivalence
Better approach: revealing subdomains
Discovering the sets requires perfect knowledge
Use heuristics to approximate cheaply

©Srinivas Devadas/John Guttag Spring 2002 Slide 6

Naive Approach: Execution Equivalence

```
int abs(int x) {
    if (x < 0) return -x;
    else return x;
}
```

All $x < 0$ are execution equivalent:
Program takes same sequence of steps for any $x < 0$

All $x \geq 0$ are execution equivalent

Suggests a test suite $\{-3, 3\}$

What's wrong with this ?

©Srinivas Devadas/John Guttag Spring 2002 Slide 7

Why Execution Equivalence Doesn't Work

```
int abs(int x) {
    if (x < -2) return -x;
    else return x;
}
```

$\{-3, 3\}$ does not reveal the error!

Two executions:
 $x < -2$ $x \geq -2$

Three behaviors:
 $x < -2$ (OK) $x = -2$ or -1 (bad) $x \geq 0$ (OK)

How about
`int abs(int x) { return 3 }`

Fundamental issue: the specification matters!

©Srinivas Devadas/John Guttag Spring 2002 Slide 8

Revealing Subdomain Approach

"Same" behavior depends on specification

Say that program has same behavior on two inputs if

- 1) gives correct result on both, or
- 2) gives incorrect result on both

Subdomain is a subset of input

Subdomain is revealing if

- 1) program has same behavior on all inputs
- 2) gives incorrect result on all of those inputs

Why not

- 2) Give correct result on all of those inputs

Partitions should have at least one revealing subdomain

©Srinivas Devadas/John Guttag Spring 2002 Slide 9

Revealing subdomain example

Which are revealing subdomains?

```
int abs(int x) {
    if (x < -2) return -x;
    else return x;
}
```

$\{-1\}$? $\{-2, -1\}$? $\{-3, -2, -1\}$?

©Srinivas Devadas/John Guttag Spring 2002 Slide 10


Heuristics for Designing Test Suites

A good heuristic gives:


- few partitions
- \forall errors e in some class of errors E , high probability that some partition is revealing for e

Different heuristics target different classes of errors


In practice, combine multiple heuristics



heuristic 1
3 partitions



combined
6 partitions



heuristic 2
2 partitions

©Srinivas Devadas/John Guttag Spring 2002 Slide 11

Black Box Testing

Heuristic: Partition using the specification

Procedure or ADT is a black box, internals hidden
Explore alternate paths through the specification


Example

```
int max(int a, int b)
// effects: a > b => returns a
//          a < b => returns b
//          a = b => returns a
```

3 cases in specification suggests 3 subdomains

- $(4, 3) \Rightarrow 4$ (i.e. any input in the subdomain $a > b$)
- $(3, 4) \Rightarrow 4$ (i.e. any input in the subdomain $a < b$)
- $(3, 3) \Rightarrow 3$ (i.e. any input in the subdomain $a = b$)

©Srinivas Devadas/John Guttag Spring 2002 Slide 12

 **More Complex Example**


Write test cases based on paths through the specification
int find(int[] a, int value) throws Missing
// effects: returns the smallest i such
// that a[i] == value, unless
// value not in a => Missing

2 obvious tests:
([4, 5, 6], 5) => 1
([4, 5, 6], 7) => throw Missing

1 more subtle test
([4, 5, 5], 5) => 1

Must hunt for multiple cases in effects or requires

©Srinivas Devadas/John Guttag Spring 2002 Slide 13

 **Partitions In Example**


Consider possibilities for a, value

No i such that a[i]==value
([4, 5, 6], 7) => throw Missing

Exists unique i. a[i]==value
([4, 5, 6], 5) => 1

Exists i,j. a[i]==value and a[j]==value and i≠j
([4, 5, 5], 5) => 1

©Srinivas Devadas/John Guttag Spring 2002 Slide 14

 **Heuristic: Boundary Testing**

Create partitions at the edges of other partitions


Why do this?

Common source of errors

- Off-by-one bugs
- Forget to handle empty container
- Overflow errors in arithmetic
- Program does not handle aliasing of objects

Small partitions at the edges of the "main" partitions have a high probability of revealing these common errors

©Srinivas Devadas/John Guttag Spring 2002 Slide 15

 **Boundary Testing**

To define boundary, must define adjacent points

One approach:

- Identify basic operations on input points
- Two points are adjacent if one basic operation away
- A point is isolated if can't apply a basic operation


Point is on a boundary if either

- There exists an adjacent point in different partition
- Point is isolated

Example: array of integers

- Basic operations: append integer, remove integer
- Adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>
- Isolated point: [] (can't apply remove integer)

©Srinivas Devadas/John Guttag Spring 2002 Slide 16

 **Some Other Boundary Cases**


Arithmetic

- Smallest/largest values
- Zero

Objects

- Same object passed to multiple arguments (aliasing)

©Srinivas Devadas/John Guttag Spring 2002 Slide 17


 **Black Box Testing: Advantages**

Process not influenced by component being tested
Assumptions embodied in code not propagated to test data.

Robust with respect to changes in implementation
Test data need not be changed when code is changed

Allows for independent testers
Testers need not be familiar with code

©Srinivas Devadas/John Guttag Spring 2002 Slide 18


 **Glass Box Testing: Advantages**

Insight into test cases
Which are likely to yield new information

Finds an important class of boundaries
Consider a isPrime that uses table lookup
If $i < 100$ check table, otherwise run algorithm

Need to check numbers on each side of 100
100 is a boundary

©Srinivas Devadas/John Guttag Spring 2002 Slide 19


 **Glass-box Testing**

Goal:
Ensure test suite covers (executes) all of the program
Measure quality of test suite with % coverage

Assumption:
high coverage =>
(no errors in test suite output
=> few mistakes in the program)

Focus: features not described by specification
Control-flow details
Performance optimizations
Alternate algorithms for different cases

©Srinivas Devadas/John Guttag Spring 2002 Slide 20

 **Glass-box Challenges**

Definition of all of the program
What needs to be covered?


Options:

- Statement coverage
- Decision coverage
- Loop coverage
- Condition/Decision coverage
- Path-complete coverage

↓ increasing number of partitions

Target % coverage
100% may be unattainable (dead code)
High cost to approach the limit


©Srinivas Devadas/John Guttag Spring 2002 Slide 21

 **Pragmatics: Regression Testing**

Whenever find and fix a bug
Store input that elicited bug
Store correct output
Put into test suite

Why this is a good idea
Helps to populate test suite with good tests
Protects against reversion that reintroduce bug
Arguably is an easy error to make (after all, it was made once, why not again?)

©Srinivas Devadas/John Guttag Spring 2002 Slide 22

 **Summary**

Testing matters
You need to convince others that program works

Key to unit testing is choice of test data
Other issues arise in integration testing

Don't confuse volume with quality of test data
Can lose relevant cases in mass of irrelevant ones
Look for revealing subdomains

Choose test data to cover
Specification (black box testing)
Code (glass box testing)

©Srinivas Devadas/John Guttag Spring 2002 Slide 23