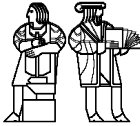


6.170 Testing



Martin Rinard
(with lots of slides from
John Chapin)
MIT EECS



Part 1. Testing Theory

validate: to increase confidence in program's correctness
correctness: program satisfies its specification

2 ways to validate:

verify: use proof techniques
test: execute program for purpose of detecting errors
(detect presence, not absence, of errors)

Test \neq debug

debug: identify the mistake that causes an error
test: compare input-output pairs to specification
debug: study the steps that lead to a given error

© John Chapin/John Guttag

2



Validation Problems

Incorrect specification

Program itself
Parts of system that it interfaces with
Really bad for verification

Incomplete specification

Specification is hard to apply

Don't know what specification should be

Don't know what you want program to do
Building on top of underlying system; don't know for sure
what underlying system does
Don't know what a specification is

© John Chapin/John Guttag

3



What is 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
(But, can think about doing this many tests today)

Key problem: choosing the test suite
small enough to finish quickly (time == \$\$)
large enough to validate the program

© John Chapin/John Guttag

4



Approach: partition the input space

Input space very large, program small
==> behavior is the "same" for sets of inputs

Ideal test suite:

identify sets with same behavior
try one input from each set

Two problems

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

© John Chapin/John Guttag

5



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 $\{-3, 3\}$

© John Chapin/John Guttag

6

Why execution equivalence doesn't work

Consider the following bug:

```
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)

© John Chapin/John Guttag 7

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

You'd like your partition to have at least one revealing subdomain

© John Chapin/John Guttag 8

Better approach: revealing subdomains

A subdomain S is revealing for an error E when
 E present and E affects any input in S →
 All inputs in S produce incorrect output

© John Chapin/John Guttag 9

Revealing subdomain example

For buggy abs, what are revealing subdomains?

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

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

© John Chapin/John Guttag 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

© John Chapin/John Guttag 11

Part 2: Black box testing

Heuristic: Partition using the specification
 Procedure or ADT is a black box, internals hidden
 Liskov: "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 partitions, so 3 test cases:
 $(4, 3) \Rightarrow 4$ (i.e. any input in the subdomain $a > b$)
 $(3, 4) \Rightarrow 4$
 $(3, 3) \Rightarrow 3$

© John Chapin/John Guttag 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

© John Chapin/John Guttag 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

© John Chapin/John Guttag 14

Heuristic: boundary testing

Create partitions at the edges of other partitions

Why do this?
 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

© John Chapin/John Guttag 15

Technical details of 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

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)
 Point is on a boundary if either
 there exists an adjacent point in different partition
 point is isolated

© John Chapin/John Guttag 16

Using the boundary heuristic

```
int find(int[] a, int value) throws Missing
// effects: returns the smallest i such
//          that a[i] == value, unless
//          value not in a => Missing
```

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

Extend the previous tests with boundary tests
 ([], 5) => Missing
 ([4], 5) => Missing
 ([4], 4) => 0 ([4, 6, 4], 4) => 0
 ([4, 5, 6], 6) => 2 ([4, 6, 6], 6) => 1

© John Chapin/John Guttag 17

Other boundary cases

Arithmetic
 Smallest/largest values
 Zero

Objects
 Same object passed to multiple arguments (aliasing)

© John Chapin/John Guttag 18

Black-box testing for ADTs

Generate values with creators, mutators and producers

Use observers to check outputs

```
// remove a missing element
IntSet i1 = new IntSet();
i1.insert(3);
i1.isIn(3) => true
i1.remove(5) // test operation
i1.isIn(3) => true
```

Test case states which operation is the actual test

© John Chapin/John Guttag 19

Tradeoffs of black-box testing

Advantages

- Robust vs. implementation changes
- Allows testing without reading implementation

Limitations

- Ignores implementation information

© John Chapin/John Guttag 20

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

© John Chapin/John Guttag 21

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

© John Chapin/John Guttag 22

Coverage metric depends on the application

RTCA DO-178B FAA standard for commercial aircraft

Level C: failure reduces safety margins

- radio data link
- statement coverage

Level B: failure reduces capability of the aircraft or crew

- GPS, collision alert system
- decision coverage

Level A: failure can cause loss of aircraft

- engine controls, flight computer
- modified condition/decision coverage

© John Chapin/John Guttag 23

Statement coverage

FAA level C

Measures:

- whether each line of code has been executed

Advantage:

- can be measured on object code

Limitation:

```
y = 0;
if (x > 0) {
  y = 5;
}
z = z+3/y;
```

SC reports 100% coverage even if x>0 in all tests
If statements without else are common

© John Chapin/John Guttag 24

Basic block coverage (variant of SC)

Measures:
whether each basic block has been executed

Advantage:
avoids a reporting problem in statement coverage

```

if (x != 0) {
  // 100 lines of straightline
  // code
} else {
  x = -1;
}

```

One test with $x = 0$ reports 1% statement coverage
One test with $x \neq 0$ reports 99% SC

Basic block coverage reports 50% for each one

© John Chapin/John Guttag 25

Decision Coverage

FAA Level B

Measures:
whether all control-flow edges have been traversed

```

y = 0;
do {
  if (x > 0) {
    y = 5;
  }
  z = z + 3/y;
} while (z < 10);

```

Watch out for caught exceptions!

© John Chapin/John Guttag 26

Decision coverage

Advantage:
relatively simple (compared to condition/decision)
reveals more control-flow errors than SC

Limitations:
Reports 100% even if while loops executed only once

Ignores branches within boolean expressions

```

if (b1 && (b2 || myfunction())) {
  // do something
}

```

reports 100% even if `myfunction()` never called here
(unless code uses short-circuit conditionals)

© John Chapin/John Guttag 27

Loop coverage

Measures:
Whether all loop bodies executed 0, 1, and >1 times

Advantages:
Reveals the most common looping errors
e.g. failure to reinitialize variable
Can combine with SC, DC, or C/DC to strengthen
Can combine with PC to weaken

Limitations:
Assumes two or more iterations are equivalent
Loops expressed as recursive calls hard to measure automatically

© John Chapin/John Guttag 28

Condition/Decision coverage

FAA Level A

Measures:
Decision coverage + whether each boolean subexpression evaluates to both true and false

Advantage:
More complete control flow coverage

```

if (b1 && (b2 || myfunction())) {
  // do something
}

```

Limitations:
Very expensive
Unclear how much benefit beyond DC

© John Chapin/John Guttag 29

Path-complete coverage

Measures:
whether each path through the program has executed

Advantage:
thorough testing

Limitations:
of paths is exponential function of # of branches
Difficult to measure coverage automatically
e.g. Are there 2 or 4 paths in:

```

if (b1) statement1;
statement2;
if (b1) statement3;

```

Must prove effect of s_1 and s_2 on b_1 to answer

© John Chapin/John Guttag 30

Summary of coverage metrics

Stronger metric subsumes the weaker metric
 All test suites that reach 100% on the stronger metric are at 100% for the weaker one

© John Chapin/John Guttag 31

Testing as part of software development

Unit and integration testing
Coverage goals
Testing strategy

© John Chapin/John Guttag 32

Unit and integration testing

Unit testing
 validate each procedure or ADT in isolation

Integration testing
 validate overall program
 challenges:
 achieving coverage
 specification errors
 debugging

© John Chapin/John Guttag 33

Coverage goals

100% is ideal, but:
 dead code
 arcane test cases required for last bits of coverage

Example tradeoff:
 write test cases to boost coverage from 90% to 95%
 OR
 do a formal technical review

Practitioner recommendation:
 90-95% coverage in unit test
 80-90% coverage in integration test
 (even FAA level A allows <100%, if explained)

© John Chapin/John Guttag 34

Testing strategy

Goals:
 detect errors with the least effort
 detect errors early enough to fix before release date

Strategy:
 breadth-first search (not depth-first)

Example integration test strategy:
 1. Invoke at least one function in 90% of the classes
 2. Invoke 90% of the functions
 3. Achieve 90% decision coverage in 100% of functions

© John Chapin/John Guttag 35

Practical Testing

Test Driver
 Runs sequence of tests on unit to be tested
 Automatically checks results

Steps
 Set up environment
 Initialize variables, open files, prepare inputs
 Run component or system
 Save results
 Check

© John Chapin/John Guttag 36



Stubs

Driver simulates part of program that calls unit

Stubs simulate part of program that unit calls

- Check that environment from unit is OK
- Check that arguments to stub are OK
- Provide return values (easier said than done...)

Can often use person as a stub!



Complications

Setting up environment can be very difficult

- Real-time inputs
- Complex environment

Invisible parts of environment

- Hashcodes in Java
- Virtual machine state in Smalltalk, LISP systems
- Nondeterministic execution

Bugs in driver or stubs



Test Suite

Produce a good set of tests

Built up over time using

- Black box approaches
- Clear box approaches
- Debugging experience

Typically inserted in automated driver

Makes testing much more effective



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 reversions that reintroduce bug
- Arguably is an easy error to make (after all, it was made once, why not again?)



Inversion of Testing and Specification

Standard situation:

- Develop specification
- Test to verify that implementation conforms to spec

Inversion

- Hack code
- Test it to determine what it does
- Use experience to develop specification (or not)

Extremely Common Variation

- Are given code, but not given (usable) spec



When is this useful?

Whenever you don't know exactly what you want program to do

- User interfaces
- Neural networks
- Machine learning

Whenever you don't know what program really does

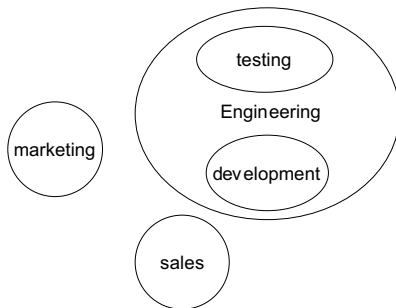
- Code reuse
- Libraries or APIs

Elements in many programming situations

Exploratory programming



Testing and your career



© John Chapin/John Guttag

43



Summary

Testing as validation activity

Input partitioning

Black-Box testing

Clear-Box testing and concept of coverage

Inversion of testing and specification

Testing and your career

© John Chapin/John Guttag

44