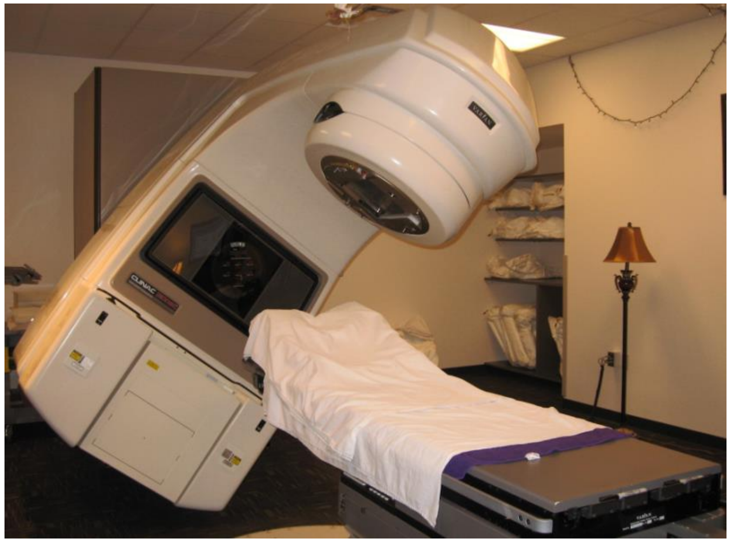
Testing

CSE 331
University of Washington

Michael Ernst

Therac-25 radiation therapy machine

- Excessive radiation killed patients (1985-87)
  - New design removed hardware interlocks. All safety checks are done in software.
  - The equipment control task did not properly synchronize if the operator changed the setup too quickly.
  - This was missed during testing, because it took practice before operators could work quickly enough to trigger the problem.
- Panama, 2000: ≥8 dead
- Many more! (NYT 12/28/2010)



Therac-2 machine

Excessiv 35-87)

New d s. All

safet

The ec erly

sync e setup

too c

This w it took

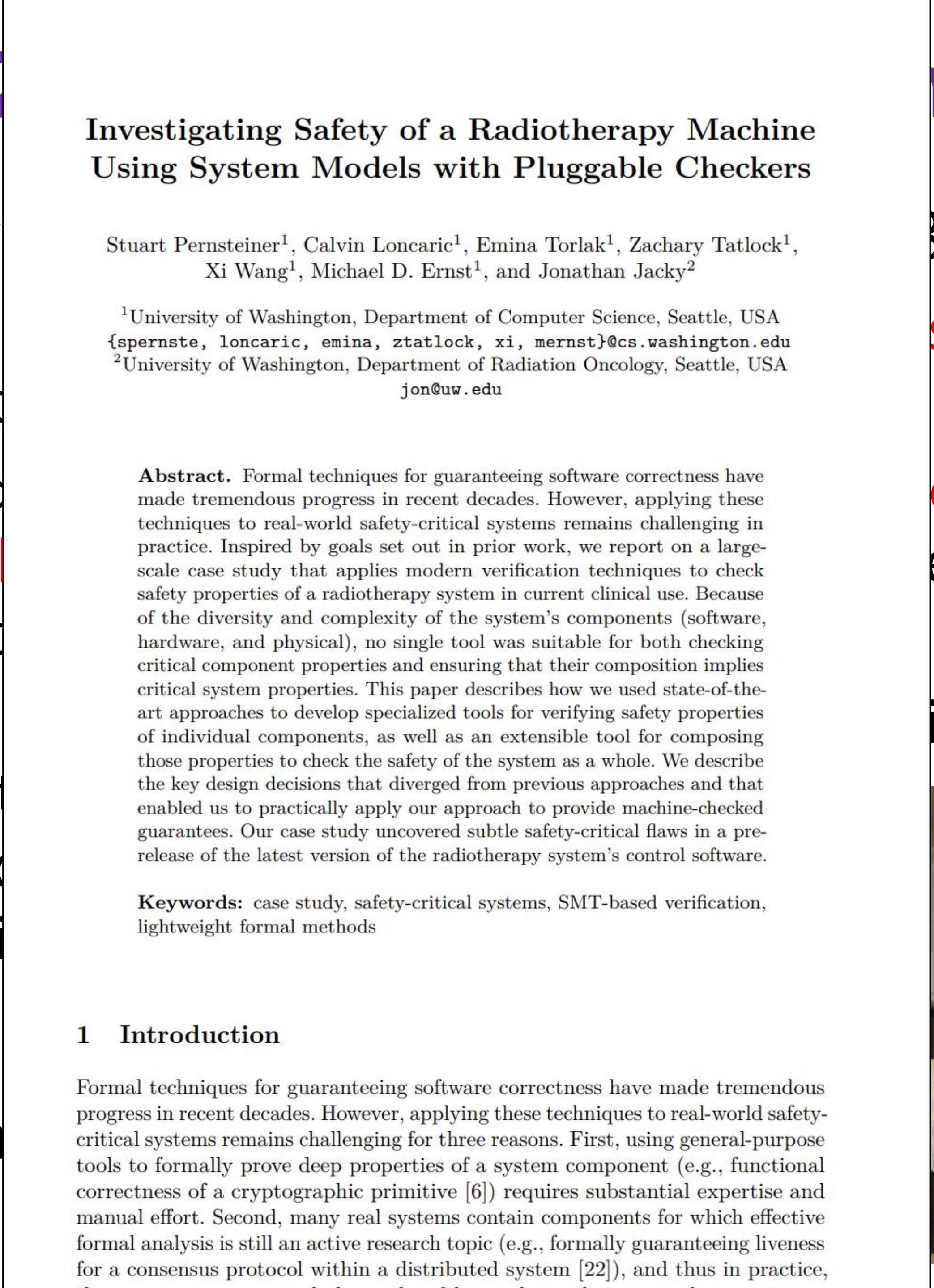
pract

coul

to tr

Panama

Many m



Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers

Stuart Pernsteiner[1], Calvin Loncaric[1], Emina Torlak[1], Zachary Tatlock[1], Xi Wang[1], Michael D. Ernst[1], and Jonathan Jacky[2]

[1]University of Washington, Department of Computer Science, Seattle, USA
{spernste, loncaric, emina, ztatlock, xi, mernst}@cs.washington.edu
[2]University of Washington, Department of Radiation Oncology, Seattle, USA
jon@uw.edu

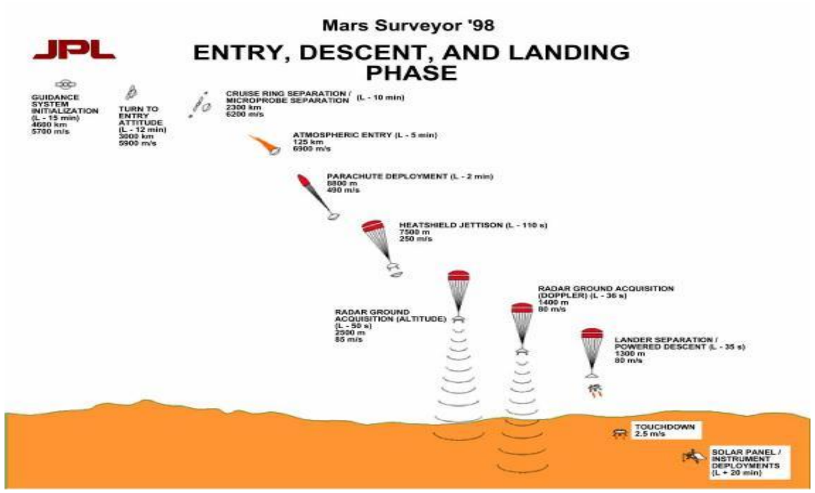
Abstract. Formal techniques for guaranteeing software correctness have made tremendous progress in recent decades. However, applying these techniques to real-world safety-critical systems remains challenging in practice. Inspired by goals set out in prior work, we report on a large-scale case study that applies modern verification techniques to check safety properties of a radiotherapy system in current clinical use. Because of the diversity and complexity of the system's components (software, hardware, and physical), no single tool was suitable for both checking critical component properties and ensuring that their composition implies critical system properties. This paper describes how we used state-of-the-art approaches to develop specialized tools for verifying safety properties of individual components, as well as an extensible tool for composing those properties to check the safety of the system as a whole. We describe the key design decisions that diverged from previous approaches and that enabled us to practically apply our approach to provide machine-checked guarantees. Our case study uncovered subtle safety-critical flaws in a pre-release of the latest version of the radiotherapy system's control software.

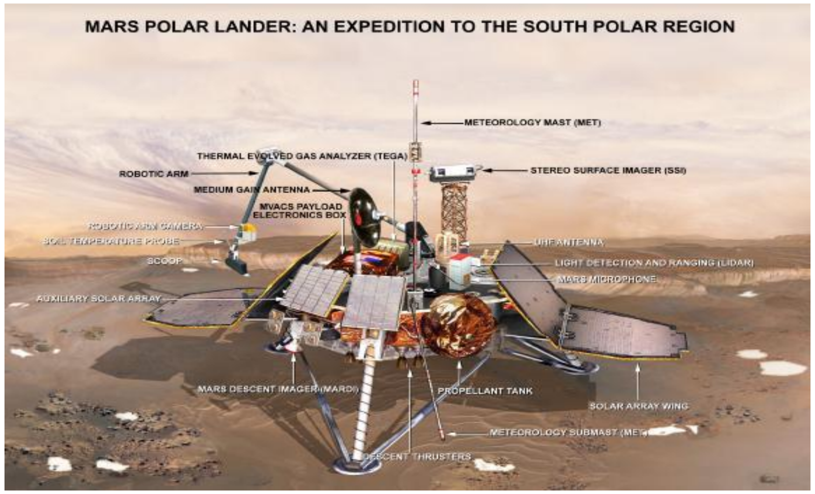
Keywords: case study, safety-critical systems, SMT-based verification, lightweight formal methods

1 Introduction

Formal techniques for guaranteeing software correctness have made tremendous progress in recent decades. However, applying these techniques to real-world safety-critical systems remains challenging for three reasons. First, using general-purpose tools to formally prove deep properties of a system component (e.g., functional correctness of a cryptographic primitive [6]) requires substantial expertise and manual effort. Second, many real systems contain components for which effective formal analysis is still an active research topic (e.g., formally guaranteeing liveness for a consensus protocol within a distributed system [22]), and thus in practice,

Mars Polar Lander


Mars Surveyor '98
JPL
ENTRY, DESCENT, AND LANDING PHASE
GUIDANCE SYSTEM INITIALIZATION (L - 15 min) 4600 km 5700 m/s
TURN TO ENTRY ATTITUDE (L - 12 min) 3000 km 5900 m/s
CRUISE RING SEPARATION / MICROPROBE SEPARATION (L - 10 min) 2300 km 6200 m/s
ATMOSPHERIC ENTRY (L - 5 min) 125 km 6900 m/s
PARACHUTE DEPLOYMENT (L - 2 min) 8800 m 490 m/s
HEATSHIELD JETTISON (L - 110 s) 7500 m 250 m/s
RADAR GROUND ACQUISITION (ALTITUDE) (L - 50 s) 2500 m 85 m/s
RADAR GROUND ACQUISITION (DOPPLER) (L - 36 s) 1400 m 80 m/s
LANDER SEPARATION / POWERED DESCENT (L - 35 s) 1300 m 80 m/s
TOUCHDOWN 2.5 m/s
SOLAR PANEL / INSTRUMENT DEPLOYMENTS (L + 20 min)


MARS POLAR LANDER: AN EXPEDITION TO THE SOUTH POLAR REGION
METEOROLOGY MAST (MET)
THERMAL EVOLVED GAS ANALYZER (TEGA)
STEREO SURFACE IMAGER (SSI)
ROBOTIC ARM
MEDIUM GAIN ANTENNA
MVACS PAYLOAD ELECTRONICS BOX
ROBOTIC ARM CAMERA
SOIL TEMPERATURE PROBE
SCOOP
UHF ANTENNA
LIGHT DETECTION AND RANGING (LIDAR)
MARS MICROPHONE
AUXILIARY SOLAR ARRAY
MARS DESCENT IMAGER (MARDI)
PROPELLANT TANK
SOLAR ARRAY WING
METEOROLOGY SUBMAST (MET)
DESCENT THRUSTERS

Legs deployed → Sensor signal falsely indicated that the craft had touched down (130 feet above the surface)

Then the descent engines shut down prematurely

The error was traced to a single line of code

Why didn't they blame the sensor?

Software bugs cost money

- Inadequate infrastructure for software testing costs the U.S. $22-$60 billion per year (NIST, 2002)
- Software bugs cost global economy $312 billion per year (Cambridge University, 2013)
- $6 billion loss from 2003 blackout in NE USA & Canada
  - Software bug in alarm system in Ohio power control room
- $440 million loss by Knight Capital Group in 30 minutes (2012)
  - High-frequency trading system

- Economies and lives destroyed by austerity measures based on study linking national debt to slow growth (2010)

Outline:
Testing principles and strategies

- Purpose of testing
- Kinds of testing
- Heuristics for good test suites
- Black-box testing
- Clear-box testing and coverage metrics
- Regression testing

- NOT: tool details (JUnit, continuous integration)

Building Quality Software

What affects software quality?

External

Correctness	*Does it do what it supposed to do?*
Reliability	*Does it do it accurately all the time?*
Efficiency	*Does it do with minimum use of resources?*
Integrity	*Is it secure?*

Internal

Portability	*Can I use it under different conditions?*
Maintainability	*Can I fix it?*
Flexibility	*Can I change it or extend it or reuse it?*

Quality Assurance (QA)

The process of uncovering problems and improving the quality of software.

Testing is a major part of QA.

Software Quality Assurance (QA)

Testing plus other activities including:

- Static analysis (assessing code without executing it)
- Proofs of correctness (theorems about program properties)
- Code reviews (people reading each others' code)
- Software process (methodology for code development)
- …and many other ways to find problems and increase confidence

No single activity or approach can guarantee software quality

> "Beware of bugs in the above code;
> I have only proved it correct, not tried it."
> -Donald Knuth, 1977

What can you learn from testing?

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsgar Dijkstra

Notes on Structured Programming
1970

Nevertheless testing is essential. Why?

What is testing for?

Validation = reasoning + testing

- Make sure module does what it is specified to do
- Uncover problems, increase confidence

Two rules:

1. Do it early and do it often
   - Catch bugs quickly, before they have a chance to hide
   - Automate the process if you can
2. Be systematic
   - Have a strategy, and test everything eventually
   - If you thrash about randomly, the bugs will hide in the corner until you're gone

Kinds of testing

Unit testing versus system/integration testing

Black-box testing versus clear-box testing

Specification testing versus implementation testing

Orthogonal choices (8 varieties total)

- Other kinds of testing exist

Unit testing and system testing

Unit testing: one module's functionality
- Method, class, interface, package, component

Test the unit in isolation from all others
- If test fails, the defect is localized
- Difficult if the unit uses other libraries
- Difficult if the unit does mutations

Done early in the software lifecycle
- – As soon as the implementation exists
- – Whenever it changes
- Don't do integration until unit tests pass

System testing = integration testing = end-to-end testing
- Run the whole system, ensure the pieces work together

Black-box tests and clear-box tests

Black-box testing

Tests depend only on the specification

Clear-box (= white-box = glass-box) testing

The implementation influences test creation

Both types of tests pass for *any* implementation

Black-box vs. clear-box affects choice of inputs

More details later in this lecture

Implementing a specification

A spec may be partial or may be weak.
An implementation is complete and has specific behavior.

- We saw this when comparing transition relations

Clients must only depend on what is in the spec.
The implementer knows how the implementation behaves.

Your boss asks you to implement specification S_1

You actually implement specification S_2 which is stronger

- S_1 says "returns *some* value that …", S_2 says "returns *smallest* value that …"
- S_1 says "returns a list that …", S_2 says, "returns a *sorted* list that …"
- S_1 says "requires $x > 0$", S_2 says, "requires $x \geq 0$"
- S_1 says "*requires* y != null", S_2 says, "*throws* Exception if y == null"
- S_2 has a particular **`toString`** implementation

Your boss doesn't even realize you did this

Your coworkers only depend on S_1

Specification and implementation tests

A specification test verifies behavior guaranteed by the specification

An implementation test verifies additional behavior of the implementation

- Ensures that your implementation behaves as designed
- One person’s implementation detail is another person’s specification

Specification vs. implementation tests affects assertions

- (can also broaden inputs)

Orthogonal to black-box vs. clear-box tests

- affects choice of inputs, within the spec’s domain

How is testing done?

Write the test

1) Choose input data/configuration
2) Define the expected outcome

Run the test

3) Run program/method on the input and record the results
4) Compare *observed* results to the *expected* outcome

sqrt example

```
// throws: IllegalArgumentException if x<0
// returns: approximation to square root of x
public double sqrt(double x)
```

What are some values or ranges of *x* that might be worth probing?

- $x < 0$ (exception thrown)
- $x \geq 0$ (returns normally)
- around $x = 0$ (boundary condition)
- perfect squares (sqrt(*x*) an integer), non-perfect squares
- *x*<sqrt(*x*) and *x*>sqrt(*x*) – that's *x*<1 and *x*>1 (and *x*=1)
- *Specific tests: say x = -1, 0, 0.5, 1, 4*

What’s so hard about testing?

“Just try it and see if it works...”

```
// requires: 1 ≤ x,y,z ≤ 10000
// effects:  computes some f(x,y,z)
int proc(int x, int y, int z)
```

Exhaustive testing would require 1 trillion runs!

- Sounds totally impractical – and this is a trivially small problem

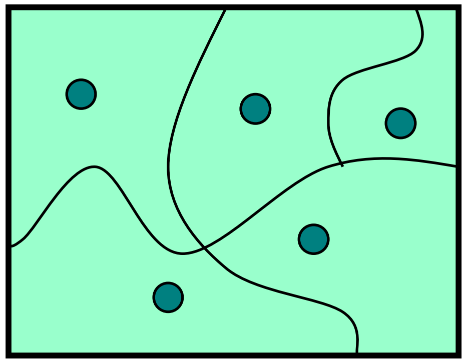
Key problem: choosing test suite (partitioning of inputs)

- Small enough to finish quickly
- Large enough to validate the program

Approach: Partition the Input Space

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

Naive approach: Execution equivalence

```
// returns:  if x < 0   ⇒ returns -x
//           otherwise ⇒ returns x
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 ≥ 0 are execution equivalent

Suggests that {-3, 3}, for example, is a good test suite

Execution equivalence is not enough

Consider the following buggy code:

```
// returns:   x < 0       ⇒ returns -x
//            otherwise ⇒ returns x
int abs(int x) {
   if (x < -2) return -x;
   else        return x;
}
```

Two execution behaviors:

x < -2 x ≥ -2

Three behaviors:

x < -2 (OK) x = -2 or -1 (bad) x ≥ 0 (OK)

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

Heuristic: Revealing Subdomains

A subdomain is a subset of possible inputs

A subdomain is *revealing* for error E if either:

- *Every* input in that subdomain triggers error E, *or*
- *No* input in that subdomain triggers error E

Need to test only one input from each subdomain

If subdomains cover the entire input space, then we are *guaranteed* to detect the error if it is present

The trick is to guess these revealing subdomains

Example

For buggy **abs**, what are revealing subdomains?

```
// returns:   x < 0          ⇒ returns -x
//            otherwise ⇒ returns x
int abs(int x) {
    if (x < -2) return -x;
    else        return x;
}
```

Example sets of subdomains:

… {-2} {-1} {0} {1} …

{…, -4, -3} {-2, -1} {0, 1, …}

Which is best? … ~~{-6, -5, -4} {-3, -2, -1} {0, 1, 2}~~ …

Heuristics for designing test suites
= heuristics for choosing inputs
= heuristics for dividing the domain

A good heuristic gives:

- few subdomains
- For all errors E in some class of errors,
  high probability that some subdomain is revealing for E

Different heuristics target different classes of errors

In practice, combine multiple heuristics

Black Box Testing

Heuristic: Explore each case/path in the specification

Procedure is a black box: (interface visible, internals hidden) but its spec is like an implementation you can test

Example

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

3 cases, so 3 tests:

(4, 3) => 4 *(i.e. any input in the subdomain a > b)*
(3, 4) => 4 *(i.e. any input in the subdomain a < b)*
(3, 3) => 3 *(i.e. any input in the subdomain a = b)*

Black Box Testing: Advantages

- Process is not influenced by component being tested
  - Assumptions embodied in code not propagated to test data.
  - Avoids “group-think” of making the same mistake.
- 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
  - Tests can be developed before the code

More Complex Example

Write test cases based on cases in the specification

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

Two obvious tests:

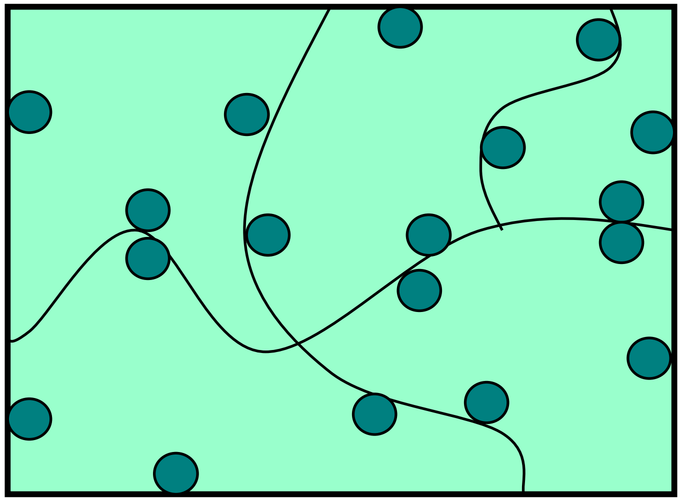
```
( [4, 5, 6], 5 )      => 1
( [4, 5, 6], 7 )      => throw Missing
```

Have we captured all the paths? **([4, 5, 5], 5) => 1**

Must hunt for multiple cases (see effects, requires)

Heuristic: Boundary Testing

- Create tests at the edges of subdomains
- Why do this?
  - off-by-one bugs
  - forgot to handle empty container, null, etc
  - arithmetic overflow
  - aliasing



- Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors
- Also, you might have misdrawn the boundaries

Boundary Testing

To define the boundary, need a *metric space*

A distance metric that defines adjacent inputs

One approach: operations define the metric space

Two values are adjacent if one operation apart

Point is on a boundary if either:

- There exists an adjacent point in a different subdomain
- Some basic operation cannot be applied to the point

Example: list of integers

Basic operations: **`create`**, **`insert`**, **`remove`**, …

Adjacent values: <[2,3],[2,3,4]>, <[2,3],[2]>

Boundary value: [] (can't apply **`remove`**)

Other Boundary Cases

Arithmetic

- Smallest/largest values
- Zero

Objects

- Null
- Circular list
- Same object passed to multiple arguments (aliasing)

Boundary Cases: Arithmetic Overflow

```
// returns: |x|
public int abs(int x)
```

What are some values or ranges of x that might be worth probing?

- *x* < 0 (flips sign) or *x* ≥ 0 (returns unchanged)
- around *x* = 0 (boundary condition)
- *Specific tests: say x = -1, 0, 1*

How about…

```
int x = Integer.MIN_VALUE;              // x = -2147483648
System.out.println(x<0);                  // true
System.out.println(Math.abs(x)<0);        // also true!
```

From Javadoc for **`Math.abs`**:

If the argument is Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative

Boundary Cases: Duplicates & Aliases

```
// modifies: src, dest
// effects:  removes all elements of src and apends
//           them in reverse order to the end of dest
<E> void appendList(List<E> src, List<E> dest) {
  while (src.size()>0) {
    E elt = src.remove(src.size()-1);
    dest.add(elt)
  }
}
```

What happens if src and dest refer to the same object?

- This is *aliasing*
- It’s easy to forget!
- Watch out for shared references in inputs

Heuristic: Clear-box testing

Focus: features not described by specification

- Control-flow details
- Performance optimizations
- Alternate algorithms for different cases

Common metric for test suite quality: Coverage

Goal: test suite covers (executes) all of the program

Assumption:

high coverage → good test suite → few mistakes remain in the program

Clear-box motivation

Some subdomains are not evident from the specification (which black-box testing uses

```
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
  if (x > CACHE_SIZE) {
    for (int i = 2; i < x/2; i++) {
      if (x%i == 0)
        return false;
      }
      return true;
    }
  } else {
    return primeTable[x];
  }
}
```

Subdomain boundary (execution difference) at *x* = CACHE_SIZE

Clear-box testing: advantages

- Provides an important class of boundaries
  - Yields useful test cases
- Gives an objective test suite quality metric (coverage)

- **Disadvantages?**

  Tests may have same bugs as implementation

  Buggy code tricks you into complacency once you look at it

Statement coverage is not enough

```
static int min (int a, int b) {
    int r = a;
    if (a <= b) {
        r = a;
    }
    return r;
}
```

Consider any test with a ≤ b, e.g., **`min(1,2)`**

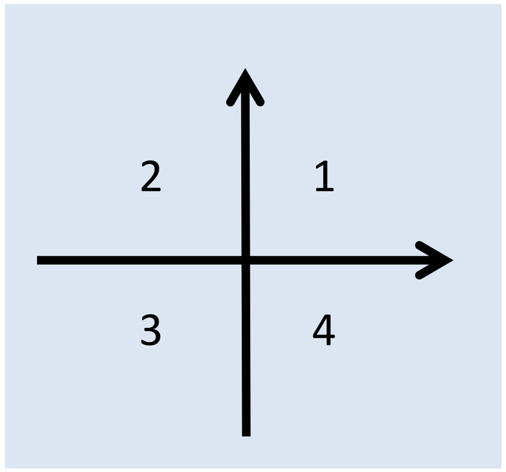
- It executes every instruction
- It misses the bug

Statement coverage is not enough

- Branch coverage = every conditional evaluates to true and false = every branch “goes both ways”

Branch coverage is not enough

```
int quadrant(int x, int y) {
  int answer;
  if (x >= 0)
    answer = 1;
  else
    answer = 2;
  if (y < 0)
    answer = 4;
  return answer;
}
```

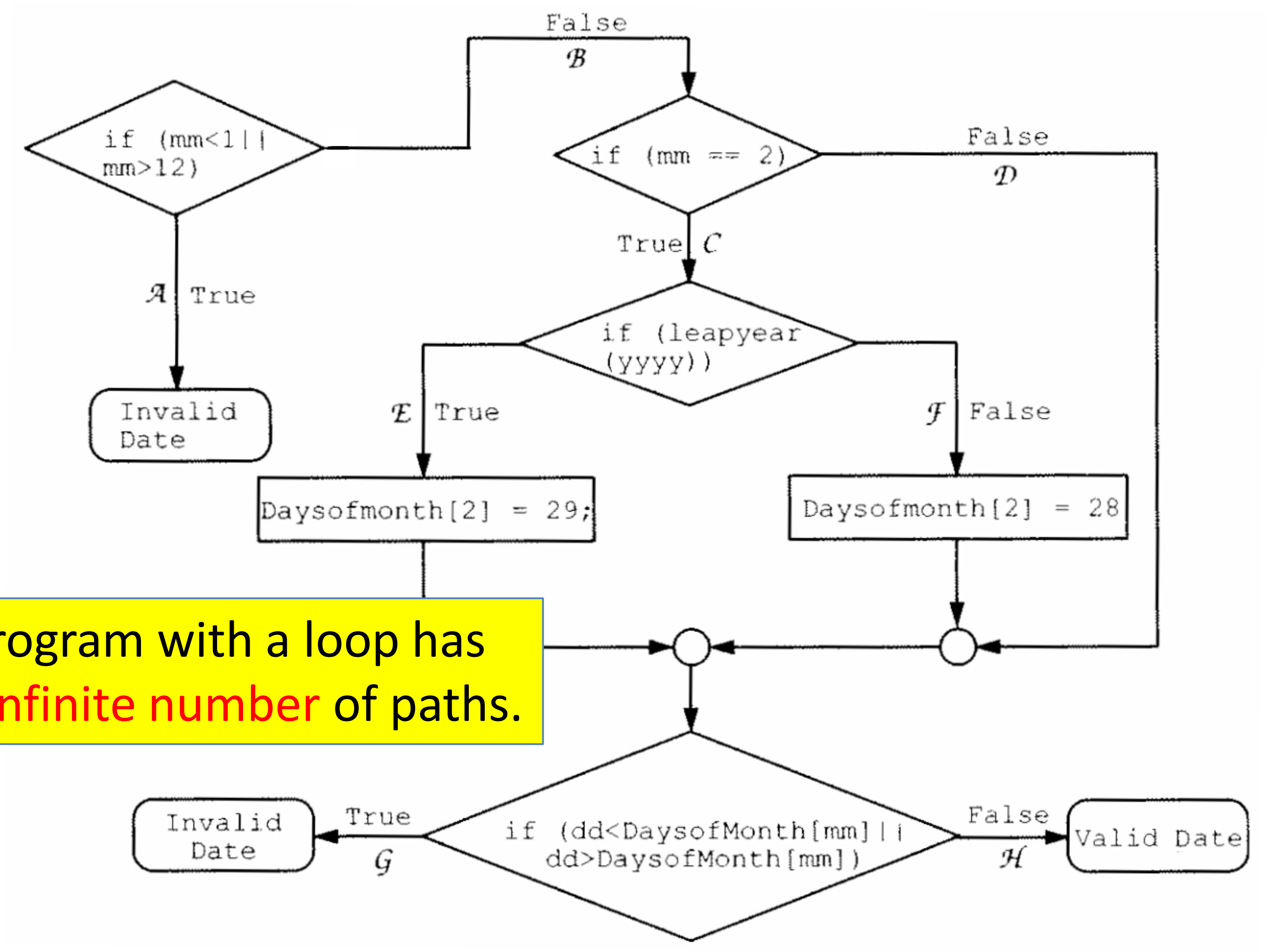


Consider a suite with two test inputs: (2,-2) and (-2,2)

- Achieves 100% branch coverage
- Misses the bug.

Path coverage = execute every path through the code

Path coverage example

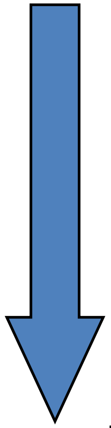


A program with a loop has an infinite number of paths.

Varieties of coverage

Covering all of the program:

Statement coverage
Branch coverage
Decision coverage
Loop coverage
Condition/decision coverage
Path coverage



Limitations of coverage:

1. 100% coverage is not always a reasonable target
   100% may be unattainable (dead code)
   High cost to approach the limit
2. Tested code is not necessarily correct
   Ex: defective `abs` method from earlier in lecture
3. Coverage is just a heuristic
   We really want the revealing subdomains

Try to write unit tests

- Ideal: each test checks one component (method,…)
  - And checks only one aspect/behavior of that component
  - Test failure indicates the exact problem
  - Debugging is a breeze
- Reality: can’t always test in complete isolation
  - Example: need to use observer(s) to see if creator, mutator, or producer yields correct results
    - If test of constructor fails, defect could be in creator *or* observer
  - Example: tested code calls other libraries (JDK)
- Advice: make each test depend on as little as possible

- Reality: your time is limited
  - Testing is of value, but any activity reaches diminishing returns
  - Goal: increase confidence to levels dictated by the business case

Pragmatics: Regression Testing

Whenever you find a bug

1. Record the input that revealed the bug, plus the correct output
2. Add these to the test suite
3. Verify that the test suite fails
4. Fix the bug
5. Verify the fix

Why is this a good idea?

Ensures that your fix solves the problem

- Don't add a test that succeeded to begin with!

Helps to populate test suite with good tests

Protects against regressions that reintroduce the bug

- It happened at least once, and it might happen again

Rules of Testing

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

- Best to catch bugs soon, before they have a chance to hide.
- Automate the process if you can
- Regression testing will save time.

Second rule of testing: ***Be systematic***

- If you randomly thrash, bugs will hide in the corner until later
- Writing tests is a good way to understand the spec
    - Think about revealing domains and boundary cases
    - If the spec is confusing → change it and/or write more tests
- The spec can be buggy too
    - Incorrect, incomplete, ambiguous, missing corner cases
- When you find a bug → write a test for it first and then fix it

Testing summary

- Testing matters
  - You need to convince others that module works
- Catch problems earlier
  - Bugs become obscure beyond the unit they occur in
- 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 (clear box testing)
- Testing can't generally prove absence of bugs
  - But it can increase quality and confidence