Testing State Machines

Saman Amarasinghe
Spring 2008
Real Programmers need no Testing!

The TOP Five List

5) I want to get this done fast, testing is going to slow me down.

4) I started programming when I was 2. Don’t insult me by testing my perfect code!

3) Testing is for incompetent programmers who cannot hack.

2) We are not Harvard students, our code actually works!

1) “Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them.” – an excerpt from a 6.170 student’s e-mail
Who said Software is Buggy?

• **Remember Ariane 5?**

  ![Ariane 5 rocket](image1.png)

  ![Ariane 5 launch](image2.png)

  ![Ariane 5 crash](image3.png)

• **The rocket self-destructed 37 seconds after launch**

• **Reason:** A control software bug that went undetected
  
  – Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
    
    • The floating point number was larger than 32767 (max 16-bit signed integer)
  
  – Efficiency considerations had led to the disabling of the exception handler.
  
  – Program crashed → rocket crashed

• **Total Cost: over $1 billion**
Another Prominent Software Bug

• **Mars Polar Lander**

  • Sensor signal falsely indicated that the craft had touched down when it was 130-feet above the surface.
    - Then the descent engines shut down prematurely
  • The error was traced to a single bad line of software code.
  • NASA investigation panels blame for the landers failure, "are well known as difficult parts of the software-engineering process,"

![Mars Polar Lander Diagram](image1)

![Mars Surveyor '98 Entry, Descent, and Landing Phase Diagram](image2)
What Impacts the Software Quality?
External

Internal
Building Quality Software

What Impacts the Software Quality?

External
- Correctness: Does it do what it suppose 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
- The process of uncovering problems and improving the quality of software.
- Testing is a major part of QA.
The Phases of Testing

**Unit Testing**
- Is each module does what it suppose to do?

**Integration Testing**
- Do you get the expected results when the parts are put together?

**Validation Testing**
- Does the program satisfy the requirements

**System Testing**
- Does it work within the overall system
Unit Testing

A test is at the level of a method/class/interface
Check if the implementation matches the specification.

Black box testing
- Choose test data without looking at implementation

Glass box (white box) testing
- Choose test data with knowledge of implementation
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
A Classic Thermostat

User set the desired temperature $T_d$

Thermostat measures the ambient temperature $T_a$

If the desired temperature is higher than the ambient temperature turn on the heat. If desired temperature is lower than the ambient temperature turn on the AC.
A Classic Thermostat

User set the desired temperature $T_d$

Thermostat measures the ambient temperature $T_a$

If the desired temperature is higher than the ambient temperature turn on the heat. If desired temperature is lower than the ambient temperature turn on the AC.

If $T_d > T_a$ turn on heating

If $T_d < T_a$ turn on air-conditioning

If $T_d = T_a$ turn heat/AC off
SHOW THE THERMOSTAT
How do we test this?

“Just try it and see if it works...”

If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off
How do we test this?

“Just try it and see if it works...”
Exhaustive testing would require millions of runs!
  Sounds totally impractical

Could see how input set size would get MUCH bigger

**Key problem:** choosing test suite (set of partitions of inputs)
  Small enough to finish quickly
  Large enough to validate the program

If \( T_d > T_a \) turn on heating
If \( T_d < T_a \) turn on air-conditioning
If \( T_d = T_a \) turn heat/AC off
**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
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

\[
\begin{align*}
& \text{If } T_d > T_a \text{ turn on heating} \\
& \text{If } T_d < T_a \text{ turn on air-conditioning} \\
& \text{If } T_d = T_a \text{ turn heat/AC off}
\end{align*}
\]
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

If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off
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 possible inputs

Subdomain is revealing for an error, E, if

1) Each element has same behavior
2) If program has error E, it is revealed by test

Trick is to divide possible inputs into sets of revealing subdomains for various errors
Heuristics for Designing Test Suites

A good heuristic gives:
- few subdomains
- ∀ errors e in some class of errors E,
  high probability that some subdomain is revealing for e

Different heuristics target different classes of errors
- In practice, combine multiple heuristics
Heuristic: Boundary Testing

Create tests at the edges of subdomains

Why do this?
- off-by-one bugs
- forget to handle empty container
- overflow errors in arithmetic
- program does not handle aliasing of objects

Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors
Heuristic: Boundary Testing

Create tests at the edges of subdomains

Why do this?
- off-by-one bugs
- forget to handle empty container
- overflow errors in arithmetic
- program does not handle aliasing of objects

Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors

If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off
Heuristic: Boundary Testing

Create tests at the edges of subdomains

Why do this?
- off-by-one bugs
- forget to handle empty container
- overflow errors in arithmetic
- program does not handle aliasing of objects

Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors

If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off
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: list of integers
- Basic operations: create, append, remove
- 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 subdomain
- Point is isolated
Other Boundary Cases

**Arithmetic**
- Smallest/largest values
- Zero

**Objects**
- Null
- Circular
- Same object passed to multiple arguments (aliasing)
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
Testing State Machine

Kind of coverage

All Actions

All States

All Transactions

All Paths
Testing State Machine

Kind of coverage

All Actions

All States

All Transactions

All Paths
All paths
A Classic Thermostat: State Diagram

User set the desired temperature $T_d$
Thermostat measures the ambient temperature $T_a$

If the desired temperature is higher than the ambient temperature turn on the heat. If desired temperature is lower than the ambient temperature turn on the AC.

If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off
A Classic Thermostat: State Diagram

User set the desired temperature $T_d$

Thermostat measures the ambient temperature $T_a$

If the desired temperature is higher than the ambient temperature turn on the heat. If desired temperature is lower than the ambient temperature turn on the AC.

If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off
Testing State Machine

Kind of coverage

- All Actions
- All States
- All Transactions
- All Paths
A Classic Thermostat: State Diagram

Implementing the Thermostat
If $T_d > T_a$ turn on heating
If $T_d < T_a$ turn on air-conditioning
If $T_d = T_a$ turn heat/AC off

What pattern should be used?
The GUI Code

```java
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

public class Main extends JFrame {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Main main = new Main();
                main.setVisible(true);
            }
        });
    }

    private JLabel statusView;
    private JSlider sliderAmbient;
    private JSlider sliderDesired;
    private State state;

    public Main() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        sliderAmbient = new JSlider();
        sliderAmbient.setBorder(BorderFactory.createTitledBorder("Ambient Temperature (" + state.get_ambient() + ")");
        sliderDesired = new JSlider();
        sliderDesired.setBorder(BorderFactory.createTitledBorder("Desired Temperature (" + state.get_desired() + ")");
        statusView = new JLabel();
        statusView.setBorder(BorderFactory.createTitledBorder(state.status()));
        controlPanel = new JPanel(new BorderLayout());
        controlPanel.setBorder(BorderFactory.createTitledBorder("Thermostat");
        controlPanel.setBackground(new Color(220, 255, 200));
        sliderDesired.setBackground(new Color(220, 255, 200));
        controlPanel.add(sliderDesired, BorderLayout.CENTER);
        controlPanel.add(statusView, BorderLayout.SOUTH);
        JPanel outerPanel = new JPanel(new BorderLayout());
        outerPanel.add(sliderAmbient, BorderLayout.NORTH);
        add(controlPanel, BorderLayout.NORTH);
        add(outerPanel, BorderLayout.SOUTH);
    }
}
```
public abstract class State {
    int desired_temp;
    int ambient_temp;

    abstract State update();
    abstract String status();

    void set_ambient(int t) { ambient_temp = t; }
    void set_desired(int t) { desired_temp = t; }

    int get_ambient() { return ambient_temp; }
    int get_desired() { return desired_temp; }

    State change_ambient(int t) {
        set_ambient(t);
        return update();
    }
    State change_desired(int t) {
        set_desired(t);
        return update();
    }
}

public class AllOff extends State {
    AllOff(State s) {
        set_ambient(s.get_ambient());
        set_desired(s.get_desired());
    }
    State update() {
        if (get_ambient() > get_desired())
            return new AcOn(this);
        if (get_ambient() < get_desired())
            return this;
        return new AllOff(this);
    }
    String status() { return "All off"; }
}

public class AcOn extends State {
    AcOn(State s) {
        set_ambient(s.get_ambient());
        set_desired(s.get_desired());
    }
    State update() {
        if (get_ambient() > get_desired())
            return new AcOn(this);
        else
            return new AllOff(this);
    }
    String status() { return "AC is on"; }
}

public class HeatOn extends State {
    HeatOn(State s) {
        set_ambient(s.get_ambient());
        set_desired(s.get_desired());
    }
    State update() {
        if (get_ambient() > get_desired())
            return new HeatOn(this);
        else
            return this;
    }
    String status() { return "Heat On"; }
}

What pattern was used? Why?
SHOW THE CODE

© Saman Amarasinghe 2008
Code Coverage

Definition of all of the program

- What needs to be covered?
- Options:
  - Statement coverage
  - Decision coverage
  - Loop coverage
  - Condition/Decision coverage
  - Path-complete coverage

100% coverage not always reasonable target

100% may be unattainable (dead code)
High cost to approach the limit
There are some subdomains that black-box testing won't give:

```java
boolean isPrime(int x) {
    for (int i=2; i<x/2; i++) {
        if (x%i==0) return false;
    }
    return true;
}
```

What is the input set?
What are the reveling subdomains?
What is a good set of boundary tests?
There are some subdomains that black-box testing won't give:

```java
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];
    }
}
```

Important transition around \(x = \text{CACHE\_SIZE}\)
Glass Box Testing: Advantages

Insight into test cases
- Which are likely to yield new information

Finds an important class of boundaries
- Consider CACHE_SIZE in isPrime example

Need to check numbers on each side of CACHE_SIZE
- CACHE_SIZE-1, CACHE_SIZE, CACHE_SIZE+1
Whenever you find and fix a bug

- Store input that elicited that bug
- Store correct output
- Put into test suite

Why is this a good idea

- Helps to populate test suite with good tests
- Protects against reversions that reintroduce bug
- Arguably is an easy error to make (happened at least once, why not again?)
A Energy Efficient Thermostat

User set the desired temperature $T_d$
Thermostat measures the ambient temperature $T_a$
Thermostat measures outdoor temperature $T_o$

If the desired temperature is higher than the ambient temperature, then turn on the heat only when outside temperature is lower than the desired temperature. Otherwise turn on the intake from outside. If desired temperature is lower than the ambient temperature turn on the AC only when outside temperature is higher than the desired temperature. Otherwise turn on the intake from outside.
A Energy Efficient Thermostat

User set the desired temperature $T_d$
Thermostat measures the ambient temperature $T_a$
Thermostat measures outdoor temperature $T_o$

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d <= T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d >= T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

What is the space of all values?

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d \leq T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d \geq T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

What is the space of all values?

- If $T_d > T_a$ and $T_d > T_o$ turn on heating
- If $T_d > T_a$ and $T_d \leq T_o$ turn on intake from outside
- If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
- If $T_d < T_a$ and $T_d \geq T_o$ turn on intake from outside
- If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

What is the space of all values?

Partition the space

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d \leq T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d \geq T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

What is the space of all values?
Partition the space

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d \leq T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d \geq T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

What is the space of all values?

Partition the space

Boundary Testing

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d \leq T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d \geq T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

What is the space of all values?

Partition the space

Boundary Testing

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d <= T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d >= T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

User set the desired temperature $T_d$
Thermostat measures the ambient temperature $T_a$
Thermostat measures outdoor temperature $T_o$

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d <= T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d >= T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
A Energy Efficient Thermostat

User set the desired temperature $T_d$
Thermostat measures the ambient temperature $T_a$
Thermostat measures outdoor temperature $T_o$

If $T_d > T_a$ and $T_d > T_o$ turn on heating
If $T_d > T_a$ and $T_d <= T_o$ turn on intake from outside
If $T_d < T_a$ and $T_d < T_o$ turn on air-conditioning
If $T_d < T_a$ and $T_d >= T_o$ turn on intake from outside
If $T_d = T_a$ turn heat/AC off
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 you're gone
- Writing tests is a good way to understand the spec
  - Think about revealing domains and boundary cases
  - If the spec is confusing → write more tests
- Spec can be buggy too
  - Incorrect, incomplete, ambiguous, and missing corner cases
- When you find a bug → fix it first and then write a test for it
**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 (glass box testing)

**Testing can't generally prove absence of bugs**
- But can increase quality by reducing the bugs
Lecture Exercise

I) Can you implement the Energy Efficient Thermostat using 4 states?

II) One inefficiency of the Energy Efficient Thermostat is when $T_d < T_o < T_a$ and $T_d > T_o > T_a$. In this case the AC or heat will be turned on even though part of the cooling/heating can still be done by pumping outside air. Can you redesign the state machine to handle this case?

III) For II, what is the partitioned space? Give a set of boundary tests.

IV) Briefly describe how to change the implementation of the Simple Thermostat to the Energy Efficient Thermostat