Testing, Debugging, and Verification

Testing, Part III

Wolfgang Ahrendt, Vladimir Klebanov, Moa Johansson

5 November 2012

Admin

- Make sure you are registered for the course. Otherwise your marks cannot be recorded.
  - Even if you are repeating the course, only taking exercises or exam.
  - If in doubt, contact the student center to double check.
  - PhD students excluded (but drop me an email so I know who you are)
- Those who have a clashing exam: contact student center/exam administration.
- Please sign up to the News group.

Exercise sessions

- First exercise session this Wednesday
- please bring laptops
- install relevant tools before
  - topic: testing
  - install JUnit beforehand
    (version JUnit4 upwards)

Overview of this Lecture

This lecture is all about unit testing

Specific topics:

- Recap JUnit: a framework for rapid unit testing
- Integrating test units
- Principles of test set construction
- Quality criteria for test sets

JUnit (Recap.)

- JAVA testing framework to write and run automated tests
- JUnit features include:
  - Assertions for testing expected results
  - Annotations to designate test cases
  - Sharing of common test data
  - Graphical and textual test runners
- JUnit is widely used in industry
- JUnit used from command line or within an IDE (e.g., Eclipse)

Recap: JUnit and Extreme Testing

- Test-cases first: Clear idea of what program should do before coding.
- Understanding of specification and requirements.
- Regression testing: re-run after changes to code.

Extreme Testing Example: Class Money

```
class Money {
    private int amount;
    private Currency currency;

    public Money(int amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }
    public Money add(Money m) {
        // NO IMPLEMENTATION YET, WRITE TEST FIRST
    }
}

class Currency {
    private String name;
    public Currency(String name) {
        this.name = name;
    }
}
```

Write a Test Case for add()

```
import org.junit.*;
import static org.junit.Assert.*;

public class MoneyTest {

    @Test public void simpleAdd() {
        Currency sek = new Currency("SEK");
        Money m1 = new Money(120, sek);
        Money m2 = new Money(160, sek);
        Money result = m1.add(m2);
        Money expected = new Money(280, sek);
        assertTrue(expected.equals(result));
    }
}
```

@Test is an annotation, turning simpleAdd into a test case

Example: Class Money

Now, implement the method under test, and make sure it fails

```
class Money {
    private int amount;
    private Currency currency;

    ....

    public Money add(Money m) {
        return null;
    }
}
```

Compile and Run JUnit test class

- JUnit reports failure
- Produce first 'real' implementation

Example: Class Money

First real attempt to implement the method under test

```
class Money {
    private int amount;
    private Currency currency;

    public Money(int amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public Money add(Money m) {
        return new Money(amount+m.amount, currency);
    }
}
```

Compile and Run JUnit test class

- JUnit will still report failure
- Fix possible defects, until test passes.
  - Can you spot it?
- What if we have different currencies?

Extend Functionality

Extend Money with Euro-exchange-rate first in test cases

```
public class MoneyTest {
    @Test public void simpleAdd() {
        Currency sek = new Currency("SEK",9.01);
        Money m1 = new Money(120, sek);
        ....
    }
    @Test public void addDifferentCurr() {
        Currency sek = new Currency("SEK",9.01);
        Money m1 = new Money(120, sek);
        Currency nok = new Currency("NOK",7.70);
        Money m2 = new Money(160, nok);
        Money result = m1.add(m2);
        Money expected = new Money(307, sek);
        assertTrue(expected.equals(result));
    }
}
```

Change, and test implementation

Common Parts into Test Fixture

```
public class MoneyTest {
    private Currency sek;
    private Money m1;

    @Before public void setUp() {
        sek = new Currency("SEK",9.01);
        m1 = new Money(120, sek);
    }

    @Test public void simpleAdd() {
        Money m2= new Money(140, sek);
        ....
    }
    @Test public void addDifferentCurr() {
        Currency nok = new Currency("NOK",7.70);
        Money m2 = new Money(160, nok);
        ...
    }
}
```

Integrating Test Units

Testing a unit may require:

Stubs to replace called procedures

Drivers to replace calling procedures

Incremental Testing: Top-Down and Bottom-Up

Explore *incremental* test strategies, following call hierarchy:

Top-Down Testing

Test main procedure, then go down the call hierarchy

- requires stubs, but no drivers

Bottom-Up Testing

Test leaves in call hierarchy, and move up to the root.
Procedure is not tested until all 'children' have been tested.

- requires drivers, but no stubs

Top-Down Testing: Pros and Cons

Advantages of Top-Down Testing

- Advantageous if major flaws occur toward top level.
- Early skeletal program allows demonstrations and boosts morale.

Disadvantages of Top-Down Testing

- Stubs must be produced (often more complicated than anticipated).
- Judgement of test results more difficult.
- Tempting to defer completion of testing of certain modules.

Bottom-Up Testing: Pros and Cons

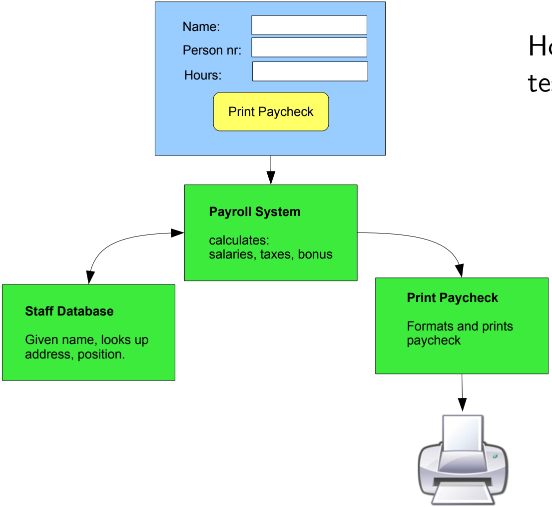
Advantages of Bottom-Up Testing

- Advantageous if major flaws occur toward bottom level.
- Judgement of test results is easier.

Disadvantages of Bottom-Up Testing

- Driver units must be produced.
- The program as an entity does not exist until the last unit is added.

Discussion: Top-down vs Bottom-up Testing



How would you go about testing the Paycheck system

- Bottom-up?
  - Which drivers do you need?
- Top-down?
  - Which stubs do you need?
- What are the advantages/disadvantages of each approach?

Test Suites

Test Suite

A test suite is a set of test cases.

- Very simple definition, but important concept
- Most central activity of testing is the creation of test suites
- Quality of test suites defines quality of overall testing effort

(When presenting test suites, we show only relevant parts of test cases.)

Principles of Test Suite Construction

Black-box testing

Deriving test suites from external descriptions of the software, including specifications, requirements, design, and input space knowledge

White-box testing

Deriving test suites from the source code internals of the software, specifically including branches, individual conditions, and statements

- Many modern techniques are a hybrid of both
- Black- and white-box are only two extremes in the space of the considered levels of abstraction from the implementation under test

Coverage Criteria

Most metrics used as quality criteria for test suites describe the degree of some kind of coverage.

These metrics are called coverage criteria.

Categories of Coverage Criteria

Following the categorisation of [AmmannOffutt] (simplified), we group coverage criteria as follows:

Coverage Criteria Grouping

- Control flow graph coverage
- Logic coverage
- Input space partitioning

Control Flow Graph

Control Flow Graph

Represent implementation under test as graph:

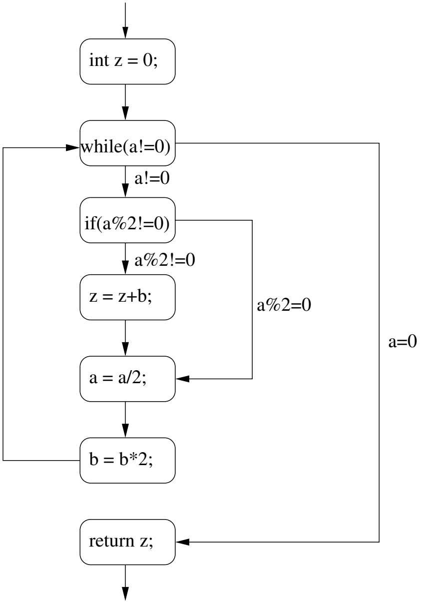
- Every statement represented by a node
- Edges describe control flow between statements
- Edges can be constrained by conditions

Example

```
int russianMultiplication(int a, int b){
    int z = 0;
    while(a != 0){
        if(a%2 != 0){
            z = z+b;
        }
        a = a/2;
        b = b*2;
    }
    return z;
}
```

[example and graph by Christian Engel]

Control Flow Graph of `russianMultiplication()`



Control Flow Graph Notions

Execution Path:

a path through a control flow graph, that starts at the entry point and is either infinite or ends at one of the exit points.

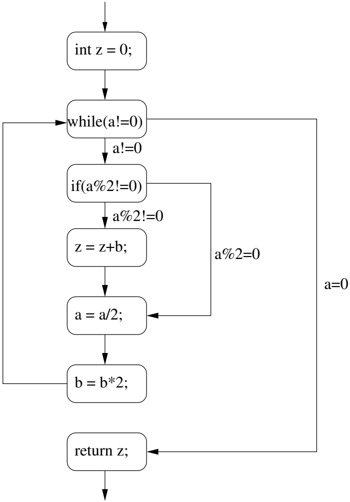
Path Condition:

a path condition PC_p for an execution path p within a piece of code c is a condition on the prestate of c causing c to execute p.

Feasible Execution Path:

an execution path for which a satisfiable path condition exists. A branch or statement is called feasible if it is contained in at least one feasible execution path.

Statement Coverage



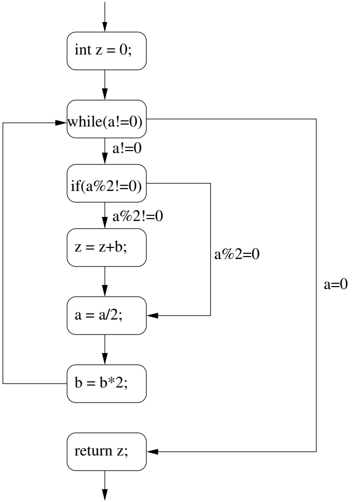
Statement Coverage (SC)

SC is satisfied by a test suite *TS*, iff for every node *n* in the control flow graph there is at least one test in *TS* causing an execution path via *n*.

For `russianMultiplication()`:

- $TS = \{(\mathtt{a} = 1, \mathtt{b} = 0)\}$ satisfies statement coverage

Branch Coverage



Branch Coverage (BC)

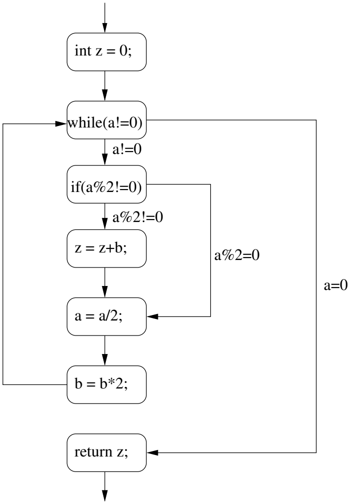
BC is satisfied by a test suite TS, iff for every edge e in the control flow graph there is at least one test in TS causing an execution path via e.

BC subsumes SC.

For `russianMultiplication()`:

- $TS = \{(\mathtt{a} = 2, \mathtt{b} = 0)\}$ satisfies branch coverage

Path Coverage



Path Coverage (PC)

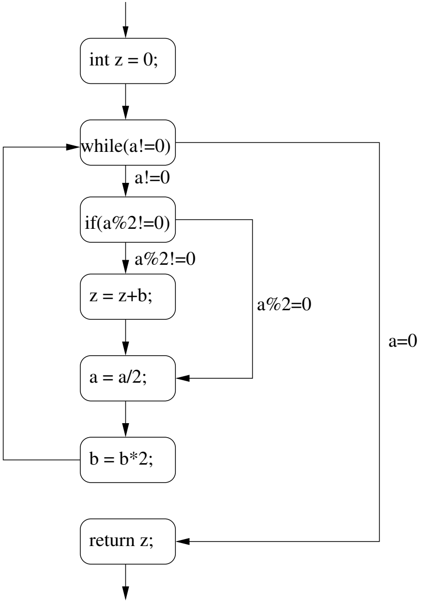
PC is satisfied by a test suite *TS*, iff for every execution path *ep* of the control flow graph there is at least one test in *TS* causing *ep*.

PC subsumes BC.

For `russianMultiplication()`:

- Number of execution paths is 2^{31}
- Size of a test suite satisfying PC is 2^{31}
- PC cannot be achieved in practice

Mini Quiz: Graph Coverage



Does the following test cases satisfy Statement Coverage, Branch Coverage and/or Path Coverage?

- [a=3, b=3] SC
- [a=0, b=2] neither
- [a=4, b=1] SC and BC

Logic Coverage

Logical (`boolean`) expressions can come from many sources:

1. Decisions in source code (e.g., `if`, `while`)
2. Decisions in software models (FSMs and statecharts)
3. Case distinctions in requirements

We focus on 1.

Decision Coverage

Let the decisions of a program p, $D(p)$, be the set of all boolean expressions which p branches on.

Decision Coverage (DC)

For a given decision d, DC is satisfied by a test suite TS if it contains at least two tests, one where d evaluates to *false*, and one where d evaluates to *true*.

For a given program p, DC is satisfied by TS if it satisfies DC for all $d \in D(p)$.

Decision Coverage

Example

For decision $((a < b) \mid\mid D)\ \&\&\ (m \geq n * o)$,
DC is satisfied for instance if *TS* triggers executions with:

$a = 5, b = 10, D = \mathit{true}, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = \mathit{false}, m = 1, n = 1, o = 1$

Inner Value Problem

- the above values are not test case inputs, but values at the time of executing the decision
- separate problem to find corresponding input values

Implicit Decisions Problem

- JAVA has implicit decisions (e.g., potential `null`-pointer access)

Condition Coverage

Let the conditions of a program p, $C(p)$, be the set of all boolean sub-expressions c of decisions in $D(p)$, such that c does not contain other boolean sub-expressions.

Given the decision $((a < b) \mid\mid D)$ && $(m \geq n * o)$,
the conditions are: $(a < b)$, D, and $(m \geq n * o)$.

Condition Coverage (CC)

For a given condition c, CC is satisfied by a test suite TS if it contains at least two tests, one where c evaluates to *false*, and one where c evaluates to *true*.

For a given program p, CC is satisfied by TS if it satisfies CC for all $c \in C(p)$.

Condition Coverage

Example

For *each condition* in $((a < b) \mid\mid D)$ && $(m \geq n * o)$,
CC is satisfied for instance if *TS* triggers executions with:

$a = 5, b = 10, D = true, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = false, m = 1, n = 2, o = 2$

No subsumption

- CC does not subsume DC
- DC does not subsume CC
- Consider $p \mid\mid q$

Modified Condition Decision Coverage, MCDC

Modified Condition Decision Coverage, MCDC

For a given condition c in decision d, MCDC is satisfied by a test suite TS if it contains at least two tests, one where c evaluates to *false*, one where c evaluates to *true*, d evaluates differently in both, and the other conditions in d evaluate identically in both.

For a given program p, MCDC is satisfied by TS if it satisfies MCDC for all $c \in C(p)$.

Modified Condition Decision Coverage, MCDC

Example

For condition $a < b$ in decision $((a < b) \mid\mid D)$ && $(m \geq n * o)$, MCDC is satisfied for instance if TS triggers executions with:

$a = 5, b = 10,\ D = false, m = 1, n = 1, o = 1$
and
$a = 10, b = 5,\ D = false, m = 8, n = 2, o = 3$

Note: To have MCDC for whole decision also need test-cases for conditions D and $(m \geq n * o)$

(Note that the examples on slides 34 and 36 do *not* guarantee MCDC.)

Modified Condition Decision Coverage, MCDC

MCDC in industrial certification standard

MCDC is required in the avionics certification standard DO-178 as the criterion to test adequately Level A software (failure of which is classified as ‘Catastrophic’).

Mini Quiz: Logical Coverage

Suppose a program contains the decision `if(x < 1 || y > z)`
Does the following test sets satisfy Decision Coverage, Condition Coverage and/or MCDC?

- `[x=0, y=0, z=1]` and `[x=2, y=2, z=1]`
  CC
- `[x=2, y=2, z=1]` and `[x=2, y=0, z=1]`
  DC
- `[x=2, y=2, z=2]`, `[x=0, y=0, z=1]`, `[x=2, y=0, z=0]`, `[x=2, y=2, z=1]`
  DC, CC, MCDC

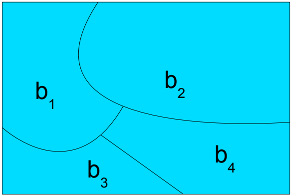
Input Space Partitioning

- Ultimately all testing is about choosing elements from input space
- Input space partitioning takes that view in a more direct way
- Input space partitioned into regions that are assumed to contain 'equally useful values'
- Test cases contain values from each region

Partitioning Domains

A partitioning q of a domain D defines a set of blocks, $B_q = \{b_1, \ldots, b_n\}$, such that:

- the blocks b_i are pairwise disjoint (no overlap)
- together the blocks cover the domain D (complete)



Normally, different partitionings are combined (see below)

Examples

Consider the domain of integer arrays.

Are the following blocks a partitioning?

- b_1 = sorted in ascending order
- b_2 = sorted in descending order
- b_3 = arbitrary order

Answer: no!

- The array [1] belongs to all blocks
- Unclear whether the array *null* belongs to any block

Combining Partitionings

When creating test cases for `findElement (int[] arr, int elem)`

partitioning q**:** `arr` is `null` (b_{q1}) or not (b_{q2})

partitioning r**:** `arr` is empty (b_{r1}) or not (b_{r2})

partitioning s**:** number of `elem` in `arr` is 0 (b_{s1}), 1 (b_{s2}), or >1 (b_{s3})

Note:

- r is a sub-partitioning of b_{q2}
- b_{s2} and b_{s3} are sub-blocks of b_{r2}
- b_{s1} overlaps with b_{r1} and b_{q2}
  (fine, as r and s are different partitionings)

Strategies for Identifying Values

After partitioning, one still has to choose values from the blocks.

Strategies

- Include valid, invalid and special values
- Sub-partition some blocks
- Explore boundaries of domains

Discussion: Input Space Partitionings

Recall the method `russianMultiplication(int a, int b)`.

Suggest some input space partitionings.

E.g.

- $a \geq 0$ or $a < 0$
- $b \geq 0$ or $b < 0$
- $a \geq b$ or $a < b$

Summary: Coverage Criteria

- Control Flow Graph
  - Statement coverage: every node visited.
  - Branch coverage: every edge traversed.
  - Path coverage: every excecution path (usually too many!)
- Logic Based
  - Decision coverage: test for each decision true/false.
  - Condition coverage: each sub-expression true/false.
  - MCDC: sub-expression true/false AND affecting decision.
- Input Space Partitioning
  - Input space split into disjoint regions.

Literature related to this Lecture

AmmannOffutt see course literature.