CS 230 Software Design and Construction - Part 3: Introduction to software engineering

Topic 6: Verification

James Skene
Department of Computer Science
University of Auckland
May 2009

Overview

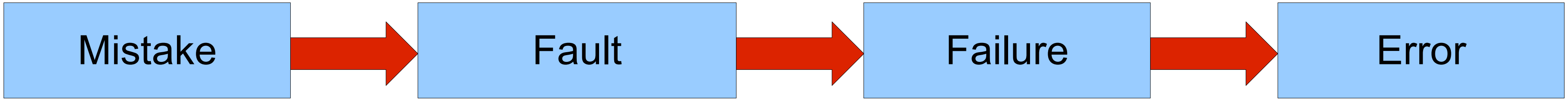
- What is verification?
- Testing
  - What is testing?
  - Approaches to testing:
    - Black-box
    - White-box
  - Systematic testing
  - Testing for different purposes
    - Unit, integration

What is verification?

- Verification is any activity intended to
  - detect faults in software
  - and/or generate confidence that software conforms to its requirements
- There are two main approaches to verification:
  - *Testing* - where we attempt to demonstrate some behaviour by running the code
  - *Formal verification* - where we attempt to prove the that some behaviour is inevitable by looking at the code (we don't have to run it, because we know what it does)
- Of these two approaches
  - testing is more commonly used
  - formal verification is more difficult, hence more costly, but...
    - we can sometimes prove that a program is completely free of certain classes of error - which is hard to do with testing
    - compilers often implement some aspects of formal verification

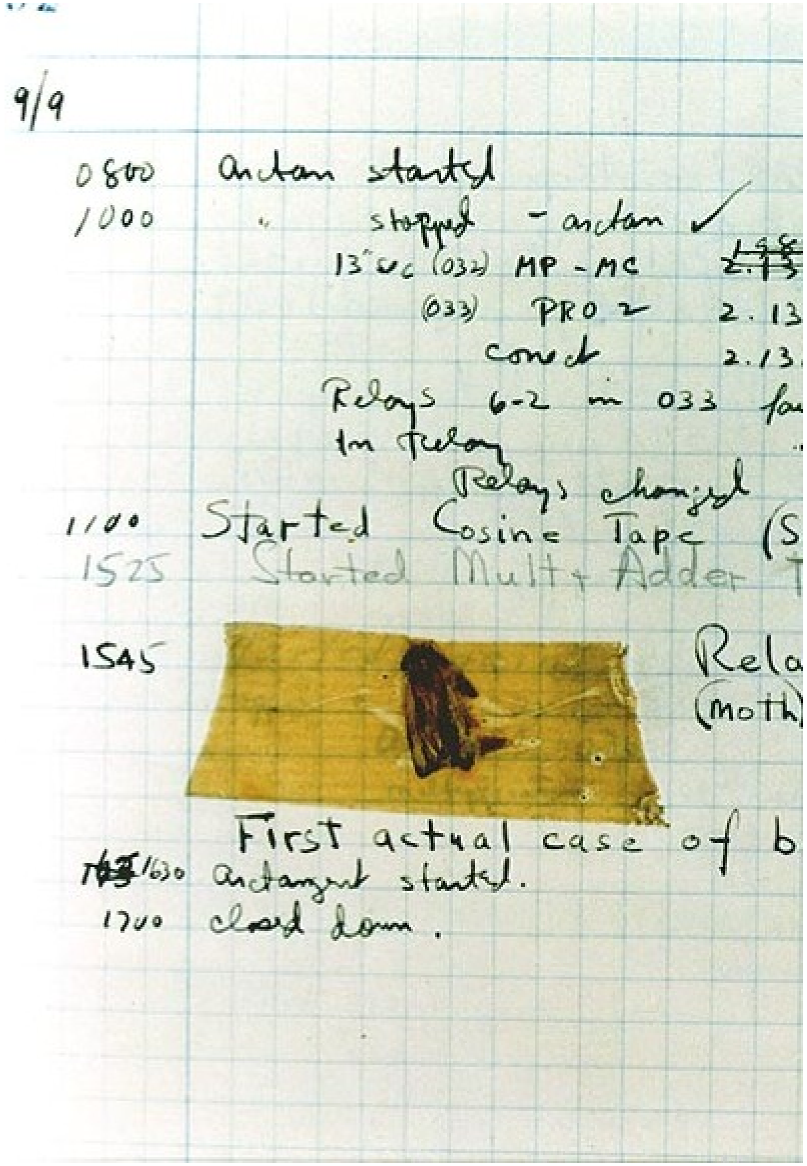
Faults

- In verification we are often concerned with the presence or absence of *faults*
- IEEE/ANSI
    - *Mistake*: A human action that produces an incorrect result
    - *Fault*: An incorrect step, process or data definition in a computer program
    - *Failure*: An incorrect result. The result of the fault
    - *Error*: The amount by which the result is incorrect


Mistake → Fault → Failure → Error

Famous faults

- First bug as reported by Grace Murray Hopper:
- Space:
    - ESA Ariane 5 explodes during take off (June 4 1996) - Arithmetic overflow 64bit -> 16bit
    - NASA Mariner 1 - FORTRAN DO statement missing decimal comma
    - NASA Mars climate orbiter (1999) - failed to convert yards to meters
    - NASA Mars Rover (Jan 21 2004) - too many open files for the flash memory; the robotic rover freezes in mid-motion; the problem was fixed remotely from Earth.



9/9

0800 Arctan started
1000 " stopped - arctan ✓
13" uc (032) MP - MC ~~2.13~~
(033) PRO 2 2.13
const 2.13
Relays 6-2 in 033 fa
in relay
Relays changed
1100 Started Cosine Tape (S
1525 Started Mult + Adder T
1545 Rela
(moth)
First actual case of b
1630 Antangent started.
1700 closed down.

More famous faults

- Medical:
  - The Therac-25 accidents (1985 - 1987), 3 people killed and 6 injured - race condition
- Computing
  - Pentium FDIV bug
  - The year 2000 problem, popularly known as the “Y2K bug”, spawned fears of worldwide economic collapse and an industry of consultants providing last-minute fixes
- Telecommunications
  - AT&T long distance network crash (1990) - chain reaction

Common types of program fault

- Logic errors - the program does not match the specification (e.g. the requirements, or design)
- Divide by zero
- Infinite loops
- Arithmetic overflow or underflow
- Exceeding array bounds
- Using an uninitialised variable
- Attempting to access memory not owned by the process (access violation)
- Memory leak
- Stack underflow or overflow
- Buffer overflow
- Off by one error
- Deadlock

What is testing?

- Testing is running the program to see what it does
- Some historical definitions
  - Establishing confidence that a program does what it is supposed to do (Hetzel, 1973)
  - The process of executing a program or system with the intent of finding errors (Myers, 1979)
  - The measurement of software quality (Hetzel, 1983)
- Detecting deviations from the specifications
- Detecting behaviour in violation of common sense
- Learning about the behaviour of a system

Automated vs. manual testing

- Testing may be automated, or not
- Automated testing
    - Expensive to set up - have to write the test
    - Cheap to repeat - so called 'regression testing'
- Manual testing
    - Cheap to do
    - May not require any programming expertise
    - Expensive to repeat
- We usually prefer automated testing during development, because it is important to repeat tests frequently as program changes.

An automated test

```
class Calculator {
  int add(int x, int y) { return x + y; }

  void testAdd() {
    if(add(5, 6) != 11)
      throw new RuntimeException(“Test failed!”);
  }

  public static void main(String [] args) {
    new Calculator().testAdd();
  }
}
```

A manual test

```
class Calculator {
  int add(int x, int y) { return x + y; }

  public static void main(String [] args) {
    try {
    System.out.println(Integer.intVal(args[0]),
      Integer.intVal(args[1]));
    }
    catch(NumberFormatException nFE){
      nFE.printStackTrace();
    }
  }
}
```

```
> java Calculator 5 6
11
```

Testing is a risk

- Writing and executing tests both have associated costs
- These have to be balanced against:
  - The opportunity of detecting a fault
  - The increased confidence in the program gained by having the test
- The opportunity of detecting a fault can be broken down into:
  - The likelihood that the test will find a fault
  - The benefit of knowing that the fault is there (which is usually mainly related to the benefit of eliminating the fault)
- It is hard to quantify the advantages in terms of increased confidence
- Corollaries:
  - We should write tests that are likely to find faults
  - We should write tests that are likely to find important faults
  - We should write tests that are likely to increase our confidence in the program (i.e. not tests that we have written before)

How should we design/choose particular tests?

- Two major strategies
  - Black-box
    - Specification oriented: does the software implement the specification?
      - Here the specification may be requirements or part of the design
    - Pros: we are testing what we need to test
    - Cons: are we sure we know what we need to test? We don't test all possible behaviours
  - White-box
    - Code oriented: if we exercise the code in all possible ways (or the most probably ways) do we get good behaviour?
    - Pros: finds all (or many) interesting behaviours
    - Cons: quickly gets difficult as code gets larger - to the point of impracticality
- Objectives: find interesting behaviours with as few tests as possible

Black-box testing

- Requires:
    - System or subsystem to be tested
    - Specifications for subsystem's output in terms of its inputs
    - The test cases - example output given inputs
    - Method for comparing actual results with expected results
    - Optional: a way to store test case results for further analysis
    - Optional: software tools for automation of testing process

Exhaustive testing

- Testing all possible input values of a function
- E.g. the identity function for integers: f(x) = x
- If integers are 16 bit - 2^16 = 65535 test cases
- If 32 bit - 2 ^32 = 4294967296 test cases
- If 64 bit - 2^64 = 18446744073709551616 test cases
- If each test case took 1 nanosecond (pretty fast - even for a function that does nothing) it would take a mere 585 years to exhaustively test this function for a 64 bit architecture
- Not practical - and probably not necessary

Testing combinations of inputs

- Consider a function that returns true if the sum of its two inputs is greater than 10:
  - f(a, b) = true, iff a + b > 10
  - f(a, b) = false, otherwise
- If we were to exhaustively test this function on a 64 bit architecture, the input space would be 2^64*2^64 = 2^128 or approximately 3 * 10^38
- According to Wikipedia there are about 1 * 10^80 atoms in the observable universe
- This is fewer than the test cases for a function of five 64 bit integers (how would we store the results?)
- The space of possible inputs tends to grow geometrically (according to a product law) in the size of input dimensions

Equivalence partitioning

- Black-box testing approach to reducing the number of test cases
- Identify a set of Equivalence Classes (ECs) of input conditions to be tested
    - each class covers a large set of other possible tests.
    - **The program is expected to behave the same for all members of a particular class according to the specification**
- Produce 1 test case from each class
- (I wonder if it is sensible to produce 2 tests from each class - to verify the assumption that the inputs form a class?)

Equivalence partitioning

- Examples
  - Input:: an integer [1-12], define
    - One valid EC within [1-12]
    - Two invalid ECs: input < 1, input > 12
  - Input: a valid value [a-z], define
    - One valid EC within [a-z]
    - One invalid EC outside [a-z] (or two?)
  - Input: must be a string representing a number, define
    - One valid EC which is a number
    - One valid EC that is not a number
  - Input: several different options may be selected
    - One valid EC for each valid input

Is equivalence partitioning good enough?

- Consider a function that takes a number and returns true if the number is over 10:
  - f(x) = true, iff x > 10
  - f(x) = false, otherwise
- There are two equivalence classes
  - x <= 10 (test case: 20)
  - x > 10 (test case: 5)
- But might we have cause to suspect that the programmer might have mishandled the situation if the number were negative (test case: -5)?
- Another example
  - f(x, y) = true, iff x + y > 10
  - Sufficient tests: (15, 0) and (0, 5)
  - Might we also want to test (0, 15) and (5, 0)?
- Any strategy to reduce the number of test cases carries an element of risk

Boundary value testing

- Errors are more likely to occur at extreme values of an equivalence class due to the difficulty involved in producing code that handles subtle distinctions correctly
- Therefore, if input specifies a range of valid values, create
  - Test cases for ends of the range
  - Invalid-input test cases for conditions just beyond the ends
- E.g. A function takes non-negative integers and adds them together: try 0 + 0 and MAX_INT + MAX_INT

Example boundary values

Argument type	Test cases
Boolean	True, False
Integer	MIN_INT, -1, 0, 1, MAX_INT
Positive integer	1, MAX_INT
Non-negative integer	0, 1, MAX_INT
Float	MIN_FLOAT, -1.0, 0.0, 1.0, MAX_FLOAT
Double	MIN_DOUBLE, -1.0, 0.0, 1.0, MAX_DOUBLE
Char	Printable, escape sequences, null-byte
Array index	min index, max index
Pointer	Null, normal
String	NULL, Length 1, Length 2. MAX_STRING_LENGTH
Structured type	One or more test cases for each field of the structure

Testing bad inputs

- For most functions, it is easy to identify equivalence classes that the specification says the function should not be able to handle
  - E.g. can't take the log of a negative number
- Should we test the function for these values?
  - It's a gamble
    - The function may never be called with a bad input value
    - But if it is, do we want it to fail gracefully (e.g. throw an exception), or do something unpredictable (e.g crash the whole program)?
    - Can we predict all contexts in which the function will be used/reused?
  - Often we do test bad values
- But, we don't always need to test all possible classes of bad input:
  - If a language has type-checking, the compiler or interpreter will ensure that the input is of the correct general type (providing we specify the types correctly in the interface to the function)
  - Not all languages have type checking!

Choosing tests can be hard

- Consider the triangles example from your test-first programming assignment. Given three numbers, classify the triangle as isosceles, or scalene.

1) a test case for an equilateral triangle
2) a test case for an isosceles triangle. (must be a triangle, not, e.g. (2,2,4))
3) a test case for an admissible scalene triangle (must be a real triangle, not, e.g. (1,2,3))
4) at least three test cases for isosceles triangles, where all permutations of sides are considered? (e.g. (3,3,4), (3,4,3), (4,3,3))
5) a test case with one side zero?
6) a test case with negative values?
7) a test case where the sum of two sides equals the third one. (e.g. (1,2,3))
8) at least three test cases for such non-triangles, where all permutations of sides are considered. (e.g. (1,2,3), (1,3,2), (3,1,2))
9) at least three test case where the sum of the two smaller inputs is greater than the third one.
10) the test case (0,0,0)
11) test cases with very large integers (maxint)?
12) a test case with non-integer values? (e.g., real numbers, hex values, strings,…)
13) a test case where 2 or 4 inputs are provided

Due to Myers who originally proposed this example. Other people have other ideas...

White-box module testing

- Selects test cases according to *program structure*
- Examines how code works - treat program as a transparent box
- Seeks faults in program code
- Attempts to exercise all of the implementation instructions

White-box module testing

- Requires:
  - Same as for black-box:
    - System or subsystem to be tested
    - Specifications for subsystem's output in terms of its inputs
    - The test cases - example output given inputs
    - Method for comparing actual results with expected results
    - Optional: a way to store test case results for further analysis
    - Optional: software tools for automation of testing process
  - In addition:
    - Code for system/subsystem to be tested

Statement coverage

- Objective:
  - Each statement in the code is executed at least once
  - Note if-else is considered one statement
  - E.g. the following code segment contains two statements

```
int remainder = inputNum % 2;
```

```
if((inputNum < 0) && (remainder == -1))
   System.out.println(
     “The input number is a negative odd integer”);
else System.out.println(
   “The input number is not a negative odd integer.”);
```

- No special input values are needed to guarantee statement coverage for this example

Decision coverage

- Each statement is executed at least once
- Each decision takes on all possible outcomes at least once

```
if((a<0) && (b>20))
   doAction1();
else if(a > 100)
   doAction2();
else doAction3();
```

- We could choose test cases (a = -1, b = 25), a = 101 and a = 50

Condition coverage

- Each statement is executed at least once
- Each condition in a decision takes on all possible outcomes at least once

```
if((a<0) && (b>20))
  doAction1();
else if(a > 100)
  doAction2();
else doAction3();
```

- We could choose test cases (a = 50, b = 5), (a = -1, b = 5), (a = 10, b = 25), (a = -1, b = 25), a = 101

Testing combinations of statements

- A program with an if-else statements often requires 2 test cases

```
if( a > 10)
  doSomething();
if(b > 10)
  doSomethingElse();
```

- Branches multiply the number of possible program execution paths - there are four possible routes through this program and we would like to test them all
- Loops also multiply the possibilities
- Number of test cases becomes very large for more complex code
  - Similar problem to the input space problem for black box testing but less obvious how we exclude tests

Systematic testing

- We may employ a test plan:
  - A collection of test cases, for invalid, unexpected, as well as valid and expected input conditions
  - Strategy for systematically examining the software to detect faults
  - May contain
    - List of requirements for testing
    - Test cases for situations most likely to occur
      - Author, date, purpose... etc...
      - Pre-condition (program state)
      - Input
      - Expected output
      - Observed output
      - Pass/fail conditions
    - Specification of operational environment (OS/hardware)
    - Etc.

Testing the whole system

- A large computer program can be regarded as consisting of a number of modules
  - Different people may be responsible for writing different modules
  - Coders other than the module authors may be responsible for combining modules
  - Testers other than the module authors may be responsible for testing the system
- We need to understand that the modules:
  - Operate correctly
  - Work together as expected
  - Result in a system that meets the requirements

Levels of testing

- Low-level testing (performed by developers)
  - Unit (module) testing
  - Integration testing
- High-level testing (preferably performed by independent test group)
  - System function testing
  - Usability testing
  - Performance testing
  - Resource use testing
  - Stress testing
  - Acceptance testing
  - Compatibility testing
  - Volume testing
  - Recovery testing
  - Security testing
  - Configuration testing

Unit (module testing)

- Units are typically units of compilation. However, in modern OO languages we consider a class to be a unit
- Test individual components of system
- Discover discrepancies between the module's interface specification and its actual behaviour
- Black-box and white-box testing are both used
- Mock objects or 'test harnesses' satisfy module dependencies

Integration testing

- Combining and testing multiple components
- Discover errors in the interfaces between components
- Non-incremental (“big-bang”)
- Incremental
    - Bottom-up
    - Top-down

Bottom-up integration testing

- Starts from lowest level
- Create drivers for modules
- Integrate an upper level module when all of its lower level modules have been tested
- Good because
  - Errors in critical modules a low level are detected early
  - Interface errors are detected late (low level modules may not provide functionality required by high-level modules)

Top-down integration testing

- Starts from highest level
- Create stubs for modules
- Integrate a low level module when all its upper level modules have been tested
- Good because
  - Interface errors are detected early
  - Errors in critical modules at low level are detected late
  - Creating stubs is more time consuming (than creating drivers?)

System function testing

- Test program as a whole
- Detect discrepancies between program's functional specifications and its actual behaviour
- Black-box testing (of whole system)

Performance and reliability testing

- Performance testing
  - Evaluate performance of system in terms of processing time
  - Look for possibility of improving the functions occupying most of the program's execution time
- Stress testing
  - Test to see what happens when the system is pushed to the high end and beyond its expected processing requirements
    - Crash, incorrect processing, delay?
    - Should respond gracefully - e.g. generate good error messages

Usability testing

- End users work with product and their responses are observed
- Adapt software to user behaviour
- Collect information from intended users
- Evaluate a product's presentation
- Characteristics including
  - Accessibility
  - Responsiveness
  - Efficiency
  - Comprehensibility

Acceptance testing

- Compare end product to requirements
- Performed by customer or end user (or their agents - e.g. another group of consultants)
- Acceptance criteria defined in contract

Alpha/beta testing

- Alpha testing
  - Internal acceptance/functional testing (e.g. for a shrink-wrap product)
- Beta testing
  - Use by real users
  - Functionality may be limited
  - Distribution may be limited (to limit system load and support requirement)

What do we do with the results of testing?

- Testing is the process of showing that a fault is present
- Debugging is the process of revealing the cause of the fault and removing it
  - Gather data
  - Develop hypothesis
  - Predict behaviour
  - Perform experiments
  - Prove/disprove hypothesis
  - Fix bug
- In the worst cases (which are not that uncommon) fixing a bug will require substantial redesign
  - Reimplementing a misunderstood feature
  - Correcting performance errors
- Some tests may even reveal problems with the requirements
  - 'It doesn't do what I wanted!'
  - This is one reason why an iterative life-cycle is often a good idea!

Formal methods

- Formal methods attempt to prove properties about a program based on the structure of the code and the semantics (meaning of the programming language)
- Formal methods are usually require a bit more expertise than testing
  - But maybe not more than good testing!
  - This makes them more expensive
  - But they generally prove strong results, such as the absence of faults of a particular kind, which are impractical to prove using testing
  - Hence they are often used in safety-critical systems where we are prepared to pay more for certainty

Model checking

- Model checking attempts to explore all possible states of a program
- A program state is the condition of the executing program at a particular instant of time. It includes
  - Values for all variables
  - What threads are active and what the next instruction they will execute is
- We can infer the value for the next state based on the previous state and the next operation - because we know what operations do
- By looking at all possible states we can determine whether bad situations ever arise
- Similar to exhaustive white box testing
- The limitations of the method are due to 'state space explosion'. Even relatively simple programs can have impractically large state spaces.

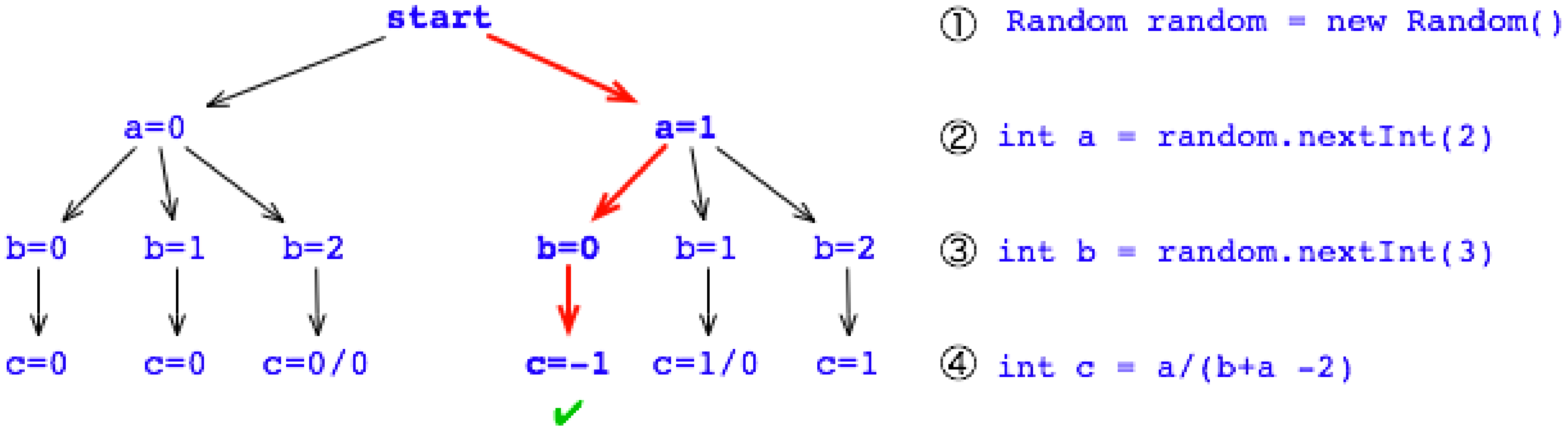
Model checking example

```
import java.util.Random;

public class Rand {
     public static void main (String[] args) {
          Random random = new Random(42);      // (1)

          int a = random.nextInt(2);           // (2)
          System.out.println("a=" + a);

          //... lots of code here

          int b = random.nextInt(3);           // (3)
          System.out.println("  b=" + b);

          int c = a/(b+a -2);                  // (4)
          System.out.println("    c=" + c);
     }
}
```

From: http://javapathfinder.sourceforge.net/sw_model_checking.html
Java pathfinder is an interesting model checking tool for Java programs

States

- Here is the state space for the program on the previous slides, considering only the variables a, b and c, and the operations that affect these variables

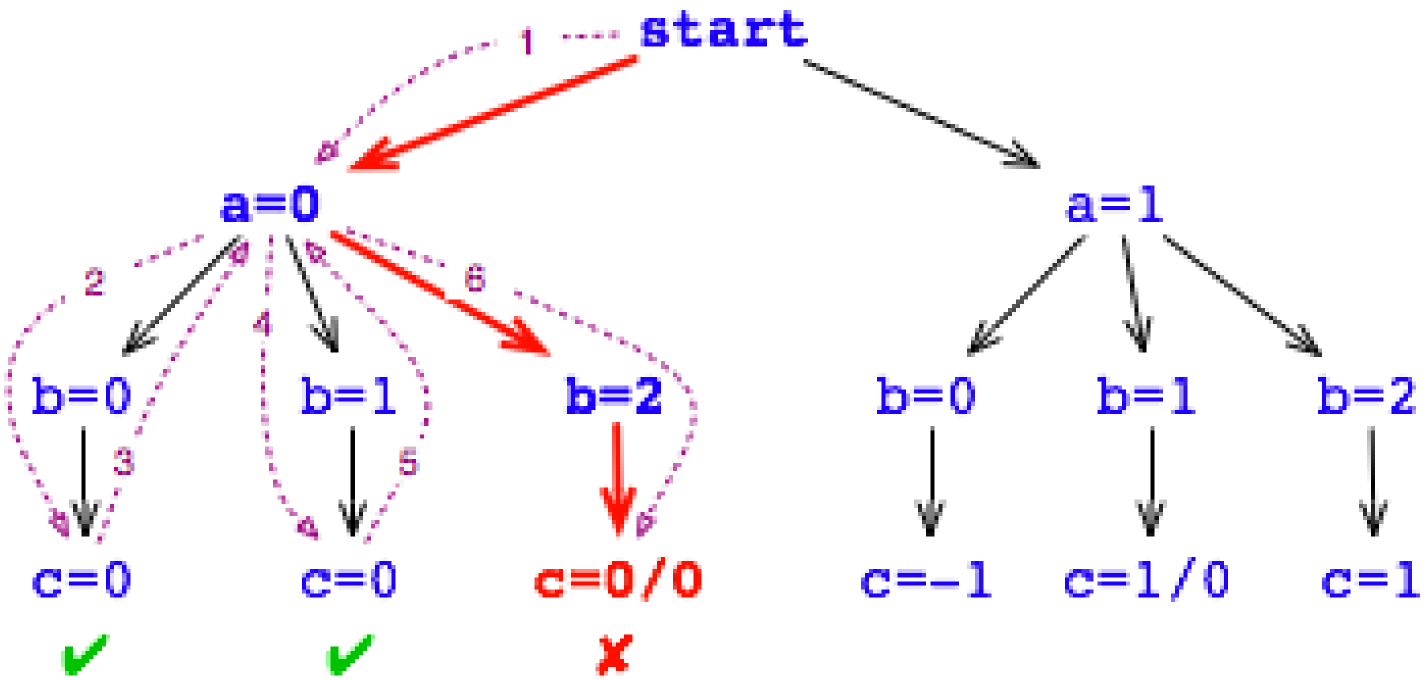


```
Random random = new Random()
int a = random.nextInt(2)
int b = random.nextInt(3)
int c = a/(b+a -2)
```

- We can verify that a given set of values can possibly be produced by the program

Attempting to prove the absence of runtime exceptions

- A useful property for Java programs, where runtime exceptions indicate problems that the programmer should have anticipated and avoided!



- Java Pathfinder does depth-first search on states - but some model checkers construct the state graph first
- Here the model-checker reveals the possibility of a null pointer exception

Addressing state space explosion

- State matching - if the program gets to the same state via different routes, only have to remember one state
- Ignore states that are not relevant to the property (as in the example)
- State abstraction - treat states that are similar from the point of view of the property being proven as the same
  - Also don't have to record all state information
- The above possibilities are the main advantage of model checking over exhaustive testing
- Heuristic state choice
  - Try to guess what states are good and ignore others - may lose the benefits of strong proof

Next topic: Wrap up