An introduction to testing & quality assurance
Part 1: Software testing & quality assurance: an overview
Definition: software quality assurance

A set of policies and activities to:

- **Define** quality objectives
- Help **ensure** that software products and processes meet these objectives
- **Assess** to what extent they do
- **Improve** them over time
The product side

Quality is the absence of “deficiencies” (or “bugs”).

More precise terminology (IEEE):

- Errors caused by Faults result in Failures
What is a failure?

For this discussion, a failure is any event of system execution that violates a stated quality objective.
Definition: testing

To test a software system is to try to make it fail

Testing is none of:
- Ensuring software quality
- Assessing software quality
- Debugging

Fiodor Chaliapine as Mephistopheles
LOS ANGELES. Failure of the Southwest's main air traffic radar system was traced to new computer software unable to recognize handoff data typed manually by Mexico controllers, the Federal Aviation Administration said Friday.

The software installed Wednesday night at the FAA's Los Angeles Center in the Mojave Desert, which controls aircraft over a 100,000-square-mile area, is the same upgrade completed successfully at 19 other FAA radar centers. But designers didn't allow for information typed in by Mexico controllers handing off flights to the facility from south of the border, which doesn't have a computerized system, FAA spokesman Jerry Snyder said. "The computer didn't recognize the information when it was passed from Mexico and it aborted." "A digit out of place could do it."

When controllers at the LA Center switched to the new system Thursday morning, it quickly failed when data from a Mexico controller was received. The FAA radar system instantly switched to backup. The computer with the new software was restarted later, but failed again. The old system was reinstalled and the system returned to operation more than two hours later. Air travel schedules were left in disarray as the FAA ordered a nationwide ground stop for all flights bound for the Southwest, causing cancellations, rerouting, long delays and airport gridlock.

Technicians must now rewrite the software to recognize Mexico controller information. It wasn't clear when a revised program would be installed.
1998 Mars Orbiter Vehicle*

The orbiter was lost due to a miscalculation in trajectory. The miscalculation was caused by an unintended and undetected mismatch between metric and English units of measurement. The use of metric units as well as the data formats to employ were specified in a navigation software interface specification (SIS) published by JPL in 1996. Despite this, the flight operations team at Lockheed Martin provided impulse data in English units of pound-force seconds rather than newton seconds. These values were incorrect by a factor of 4.45 (1 lbf = 4.45 N). The mix-up caused erroneous course corrections that resulting in the orbiter descending too low in Mars atmosphere. The vehicle either burned up or bounced off into space.

## Quality assurance techniques

<table>
<thead>
<tr>
<th>Process</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>Tool-supported</td>
</tr>
<tr>
<td>Technology-generic</td>
<td>Technology-specific</td>
</tr>
<tr>
<td>Phase-generic</td>
<td>Phase-specific (analysis, design, implementation...)</td>
</tr>
<tr>
<td>Product-generic</td>
<td>Product-specific (code, documentation...)</td>
</tr>
<tr>
<td>Build (a priori)</td>
<td>Assess (a posteriori)</td>
</tr>
<tr>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Informal</td>
<td>Mathematical</td>
</tr>
<tr>
<td>Complete</td>
<td>Partial</td>
</tr>
<tr>
<td>Free</td>
<td>Commercial</td>
</tr>
</tbody>
</table>
Quality assurance throughout the process

“Software” is not just code!

Quality affects code, documentation, design, analysis, management, the software process, and the software quality policy itself.

Most of the techniques presented will, however, be for code.
Process-based approaches to quality assurance

- Lifecycle models
- Inspections
- Open-source process
- eXtreme Programming (XP)
Inspections

(Also known as code walk-throughs)
Analysis of code (or other products) performed in a cooperative, organized manner. Rules:

- 2 to 8 participants
- Focus is discovery of errors (not correction or alternatives)
- All documents available in advance, preparation required
- Predefined amount of time
- Not chaired by developer’s boss
- Well organized (author, chair, secretary, ...)
- Report must be produced, lessons learned and applied
- Can be punctual or systematic

Pros: more eyes see more problems
Cons: time consuming, manual; tools may be more effective
Open-source processes

“Given enough eyes, all bugs are shallow"
Eric Raymond

The opposite of “security through obscurity”
Someone has been making a major effort to clean up the code in the FreeBSD tree. In two days he has reported three instances of the following common C error:

```
if (x = y)
```

instead of

```
if (x == y)
```

This is in running code, in an OS whose developers consider stability to be one of its major advantages over other offerings.

He also reported some missing breaks in a switch statement---many of us remember what THAT error did not too long ago.

(Fred Gilham, comp.risks, 28 January 1999)
More buts...

"Two buffer overflows recently found in Kerberos version 5 could only be exploited when used in conjunction with each other.

Security reviews of source code tend to be complex and boring. Even many experts don't like to do them.

Consider the open source FTP server wu-ftp. In the past two years, several very subtle buffer overflow problems have been found in the code. Almost all had been in the code for years, even though it had been examined many times by both hackers and security auditors... One tool was able to identify one of the problems as potentially exploitable, but researchers examined the code thoroughly and came to the conclusion that there was no way the problem could be exploited."

Process practices: configuration management

Configuration management: the tool-supported control of accessing and re-creating software elements in their successive versions

Origin: 1970s, Make & RCS

Numerous tools available today

Applicable to all software elements

Key “best practice” of the industry
<table>
<thead>
<tr>
<th>Quality assurance techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process</strong></td>
</tr>
<tr>
<td>Manual</td>
</tr>
<tr>
<td>Technology-generic</td>
</tr>
<tr>
<td>Phase-generic</td>
</tr>
<tr>
<td>Product-generic</td>
</tr>
<tr>
<td>Build (a priori)</td>
</tr>
<tr>
<td>Static</td>
</tr>
<tr>
<td>Informal</td>
</tr>
<tr>
<td>Complete</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Relative cost to correct a defect

Technology support

Faster computers → faster turnaround
Better programming languages
  ➢ Static typing
  ➢ Garbage collection
  ➢ More abstraction
Interactive Development Environments
Component-based development
Configuration management tools
Static analysis tools
Testing tools
Model checking tools
Proof tools
Part 2:
Model checking
The Blue Screen of Death

Windows

A fatal exception 0E has occurred at 0028:C0011E36 in UXD UMM(01) + 00010E36. The current application will be terminated.

- Press any key to terminate the current application.
- Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue _
Did you know?

Microsoft does not like blue screens either.
On blue screens

The majority of blue screens are caused by 3rd party software

Most of this software is device drivers

- Complex software (concurrency, race conditions, lock keeping)
- Running “unprotected” by the OS
- Written for top performance
- Written by non-software-engineers
- Difficult to debug
Does a program $P$ satisfy a certain property $Q$?
- Proving is difficult
- Testing is not complete
Model checking

**Models Checking:** Let’s test every possible input

(this works for hardware!)
But:

We just have too many states (state space explosion)

```vhdl
positive_max (a, b : INTEGER) : INTEGER is
  require
    a_positive: a >= 0
    b_positive: b >= 0
  do
    if a > b then Result := a else Result := b end
  ensure
    result_positive: Result >= 0
  end

has got $2^{64} = 18,446,744,073,709,551,616$ different inputs
```
Boolean abstraction to the rescue

Let's replace every $x \geq 0$ by $\text{POS}_x$

positive_max $(\text{POS}_a, \text{POS}_b : \text{BOOLEAN}) : \text{BOOLEAN}$ is

require

$a_{\text{positive}} : \text{POS}_a$
$b_{\text{positive}} : \text{POS}_b$

do

if $?$ then $\text{POS}_{\text{Result}} := \text{POS}_a$
else $\text{POS}_{\text{Result}} := \text{POS}_b$ end

ensure

result_positive: $\text{POS}_{\text{Result}}$

end

How many possible inputs do we have now?
The SLAM process

C-Code \rightarrow Boolean Program Generator \rightarrow Boolean Program

Bug found \rightarrow Predicate Discoverer \rightarrow Error Path

No \rightarrow Check successful?

Yes \rightarrow Model Checker for Boolean Programs \rightarrow Code correct

Error Path \rightarrow Bug found

Predicate Discoverer \rightarrow NEWTON

Model Checker for Boolean Programs \rightarrow BEBOP

C2BP
Part 3:
Static analysis
Static analysis

Definition: analyze software texts for possible deficiencies

Static: no execution

- The rich man’s compiler
- The poor man’s prover!
Tools

Lint (for C, originally under Unix)

Numerous tools for Fortran, starting with RSVP-80.

PREfix, PREfast (Microsoft)

ESC/Java (DEC then Compaq SRC)

Boogie (Microsoft)
PREfix and PREfast: static analysis tools*

Analyze code and detect potential defects

- Advantages:
  - Not limited by test cases
  - Identify location of defect precisely (easy to fix)
  - Usable early in the cycle

Tools - not magic bullets!
Complement other quality techniques

- Testing, AppVerifier and other dynamic checking, code review, debuggers, ...

*Source for PREfix slides: John Pincus, Microsoft
Some Defects PREfix Finds

Memory Management
- Double free
- Freeing pointer to non-allocated memory (stack, global, etc.)
- Freeing pointer in middle of memory block

Initialization
- Using uninitialized memory
- Freeing or dereferencing uninitialized pointer

Bounds violations
- Overrun (reference beyond end)
- Underflow (reference before start of buffer)
- Failure to validate buffer size

Resource Leakage
- Leaking Memory/Resource

Pointer Management
- Dereferencing NULL pointer
- Dereferencing invalid pointer
- Returns pointer to local
- Dereferencing or returning pointer to freed memory

Illegal State
- Resource in illegal state
- Illegal value
- Divide by zero
- Writing to constant string
Example defect

extern TCHAR g_szName[MAX_PATH + 1];
static TCHAR c_szServerName[] =
    "SERVER_NAME";
DWORD dwSize = sizeof(g_szName);
TCHAR szAnsiName[MAX_NAME_LENGTH + 1];
pECB->GetVariable (pECB->ConnID,
    c_szServerName, szAnsiName, &dwSize);
Example defect (continued)

extern TCHAR g_szName[MAX_PATH + 1];
static TCHAR c_szServerName[] = "SERVER_NAME";
DWORD dwSize = sizeof(g_szName);
TCHAR szAnsiName[MAX_NAME_LENGTH + 1];
pECB->GetVariable (pECB->ConnID,
    c_szServerName, szAnsiName, &dwSize);

Buffer overflow!
PREfix: HTML User Interface

**Annotated Source File - Microsoft Internet Explorer provided by ITG**

**View Annotated Source**

**PREfix simulation path begins**

```c
128  if (Size = Size / sizeof(RTL_CRITICAL_SECTION_DEBUG))
    p = (RTL_CRITICAL_SECTION_DEBUG)BaseAddress + Size
130  *(RTL_CRITICAL_SECTION_DEBUG *)p = NULL;
131  while (--Size) {
132    p1 = p - 1;
133    *(RTL_CRITICAL_SECTION_DEBUG *)p1 = p;
134    p = p1;
135  }
136
138  return p;
```

resource.c, line 138 : warning (1): using uninitialized memory 'p'
problem occurs in function 'RtlpChainDebugInfo'
variable declared on line 126
Problem occur when the following conditions are true:
Limitations

Not complete:
Functions may have huge numbers of paths
PREfix only explores $N$ paths per function
  - User-configurable, default is 50
  - I.e., gives up on completeness
Experiments indicate
  - Number of defects grows slowly with more paths
    - E.g., defects for 200 paths = 1.2 * defects for 50 paths
    - defects for 1000 paths = 1.25 * defects for 50 paths
  - Analysis time grows linearly with more paths
    - E.g., time for 1000 paths = 20 * time for 50 paths

Not sound:
  - Loops: traverse 0 or 1 time and then approximate
  - Predicates: solver optimized for speed, can return unnecessary “don’t know”
  - Recursion: explore cycles “until we’re bored”
Noise

Noise = “messages people don’t care about”
   ➢ (not just “bogus” messages)

Usually, noise is worse than missing a defect

Too much noise
   => people won’t use the tool
   == missing all the defects
Noise sources

- Incorrect requirements
- Integration issues
- Usability issues (e.g., unclear messages)
- Parser incompatibilities
- Analysis inaccuracies

...
“Works well in practice”

Finds enough real defects to be useful

Noise is low enough that people use it

Scales well, so works on large code bases
Sample usage: Windows organization

PREfix: centralized runs
- Defects filed automatically
- Roughly monthly from 1/2000-present
- Some “virtual build labs” and individual teams also run PREfix on their own

PREfast: run by individual developers/testers
- Fix before checkin
- Or run against checked-in code
A multi-year project: Win2K/XP/.NET Server

<table>
<thead>
<tr>
<th></th>
<th>Ramp up</th>
<th>Regular use</th>
<th>Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time Frame</strong></td>
<td>12/98-10/99</td>
<td>10/99-6/01</td>
<td>7/01-</td>
</tr>
<tr>
<td><strong>Code Size</strong></td>
<td>4M</td>
<td>27M-30M</td>
<td>30+M</td>
</tr>
<tr>
<td><em>(SLOC, C/C++)</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Goal</strong></td>
<td>Fixes for Win2K</td>
<td>Improvements in XP reliability</td>
<td>Fix security defects</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Security defects

The tools originally focused on the classic “reliability defects”
- Uninitialized memory, NULL/invalid pointers, Memory leaks, State violations

Added since:
- Buffer overflows
- Format string bugs
- NULL DACL
- CreateProcess misuse
Static analysis: from bleeding-edge to mundane

PREfix and PREfast established credibility
- Useful: benefits outweigh costs
- Real: a part of standard development methodology

Natural questions:
- How to make tools more effective?
- What problems are amenable to static analysis?

Product groups are now
- Actively soliciting other research tools
- Developing their own tools
Part 4:
Test basics
What does testing involve?

Determine which parts of the system you want to test
Find input values which should bring significant information
Run the software on the input values
Compare the produced results to the expected ones
(Measure execution characteristics: time, memory used, etc)
Components of a test

**Test case** - specifies:
- The state of the implementation under test (IUT) and its environment before test execution
- The test inputs
- The expected result

**Expected results** - what the IUT should produce:
- Returned values
- Messages
- Exceptions
- Resultant state of the IUT and its environment

**Oracle** - produces the results expected for a test case
- Can also make a pass/no pass evaluation
Test execution

**Test suite** - collection of test cases

**Test driver** - class or utility program that applies test cases to an IUT

**Stub** - partial, temporary implementation of a component

- May serve as a placeholder for an incomplete component or implement testing support code

**Test harness** - a system of test drivers and other tools to support test execution
Types of tests w.r.t. scope

**Unit test** - scope: typically a relatively small executable

**Integration test** - scope: a complete system or subsystem of software and hardware units
- Exercises interfaces between units to demonstrate that they are collectively operable

**System test** - scope: a complete, integrated application
- Focuses on characteristics that are present only at the level of the entire system
- Categories:
  - Functional
  - Performance
  - Stress or load
Types of tests w.r.t. intent

Fault-directed testing - intent: reveal faults through failures
- Unit and integration testing

Conformance-directed testing - intent: to demonstrate conformance to required capabilities
- System testing

Acceptance testing - intent: enable a user/customer to decide whether to accept a software product
Types of tests w.r.t. intent (continued)

Regression testing - Retesting a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

Mutation testing - Purposely introducing faults in the software in order to estimate the quality of the tests.
Testing and the development phases

- Unit testing - implementation
- Integration testing - subsystem integration
- System testing - system integration
- Acceptance testing - deployment
- Regression testing - maintenance
V-shaped lifecycle model

FEASIBILITY STUDY

REQUIREMENTS ANALYSIS

GLOBAL DESIGN

DETAILED DESIGN

IMPLEMENTATION

UNIT VALIDATION

SUBSYSTEM VALIDATION

SYSTEM VALIDATION

DISTRIBUTION
Black box vs white box testing (1)

<table>
<thead>
<tr>
<th>Black box testing</th>
<th>White box testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses no knowledge of the internals of the SUT</td>
<td>Uses knowledge of the internal structure and implementation of the SUT</td>
</tr>
<tr>
<td>Also known as responsibility-based testing and functional testing</td>
<td>Also known as implementation-based testing or structural testing</td>
</tr>
<tr>
<td>Goal: to test how well the SUT conforms to its requirements (Cover all the requirements)</td>
<td>Goal: to test that all paths in the code run correctly (Cover all the code)</td>
</tr>
</tbody>
</table>
## Black box vs white box testing (2)

<table>
<thead>
<tr>
<th>Black box testing</th>
<th>White box testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses no knowledge of the program except its specification</td>
<td>Relies on source code analysis to design test cases</td>
</tr>
<tr>
<td>Typically used in integration and system testing</td>
<td>Typically used in unit testing</td>
</tr>
<tr>
<td>Can also be done by user</td>
<td>Typically done by programmer</td>
</tr>
</tbody>
</table>
White box testing

Allows you to look inside the box
Some people prefer “glass box” or “clear box” testing
Partition testing

If you can’t test every value of the input domain, how do you choose the inputs for your tests?

One solution is partition testing

Partition – divides the input space into sets which hopefully have the property that any value in the set will produce a failure if a bug exists in the code related to that partition

A partition must satisfy two properties:

- **Completeness**: the partition must cover the entire domain
- **Disjointness**: the sets must not overlap
Examples of partition testing

Equivalence class - a set of input values so that if any value in the set is processed correctly (incorrectly) then any other value in the set will be processed correctly (incorrectly)

Boundary value analysis

Special values testing
Choosing values

Each Choice (EC): A value from each set for each input parameter must be used in at least one test case.

All Combinations (AC): A value from each set for each input parameter must be used with a value from every set for every other input parameter.
Partition testing: conclusions

Applicable to *all levels* of testing - unit, class, integration, system, etc.

Based only on the *input space* of the program, not the implementation
Test automation

Testing is so difficult and time consuming...

So why not do it automatically?

What is most commonly meant by “automated testing” currently is automatic test execution

But actually...
What can you automate?

**Test generation**
- Generation of test data (objects used as targets or parameters for feature calls)
- Procedure for selecting the objects used at runtime
- Generation of test code (code for calling the features under test)

**Test execution**
- Running the generated test code
- Method for recovering from failures
What can you automate? (continued)

**Test result evaluation**
- Classifying tests as pass/no pass
- Other info provided about the test results

**Estimation of test suite quality**
- Report a measure of code coverage
- Other measures of test quality
- Feed this estimation back to the test generator

**Test management**
- Let the user adapt the testing process to his/her needs and preferences
- Save tests for regression testing
Push-button testing

Never write a test case, a test suite, a test oracle, or a test driver

Automatically generate

- Objects
- Feature calls
- Evaluation and saving of results

The user must only specify the system under test and the tool does the rest (test generation, execution and result evaluation)
Testing strategy

How do we plan and structure the testing of a large program?

- **Who is testing?**
  - Developers / special testing teams / customer
  - It is hard to test your own code

- **What test levels do we need?**
  - Unit, integration, system, acceptance, regression test

- **How do we do it in practice?**
  - Manual testing
  - Testing tools
  - Automatic testing
Testing basics: literature


Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, in preparation
Part 5: Unit testing
Test-driven development (TDD)

Software development methodology
One of the core practices of extreme programming (XP)
Write test, write code, refactor
More explicitly:
1. Write a small test.
2. Write enough code to make the test succeed.
3. Clean up the code.
4. Repeat.
Always used together with xunit
The generic name for any test automation framework for unit testing

- **Test automation framework** - provides all the mechanisms needed to run tests so that only the test-specific logic needs to be provided by the test writer

Implemented in all the major programming languages:

- **JUnit** - for Java
- **cppunit** - for C++
- **SUnit** - for Smalltalk (the first one)
- **PyUnit** - for Python
- **vbUnit** - for Visual Basic
Unit testing framework for Java
Written by Erich Gamma and Kent Beck
Open source (CPL 1.0), hosted on SourceForge
Current version: 4.0
Available at: www.junit.org
JUnit: Overview

Provides a framework for running test cases

Test cases

- Written manually
- Normal classes, with annotated methods

Input values and expected results defined by the tester

Execution is the only automated step
How to use JUnit

Requires JDK 5

Annotations:
- @Test for every method that represents a test case
- @Before for every method that will be executed before every @Test method
- @After for every method that will be executed after every @Test method

Every @Test method must contain some check that the actual result matches the expected one - use asserts for this
- assertTrue, assertFalse, assertEquals, assertNull, assertNotNull, assertSame, assertNotSame
Example: basics

```java
package unitests;

import org.junit.Test; // for the Test annotation
import org.junit.Assert; // for using asserts
import junit.framework.JUnit4TestAdapter; // for running

import ch.ethz.inf.se.bank.*;

public class AccountTest {
    @Test
    public void initialBalance() {
        Account a = new Account("John Doe", 30, 1, 1000);
        Assert.assertEquals("Initial balance must be the one set through the constructor", 1000, a.getBalance());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AccountTest.class);
    }
}
```

To declare a method as a test case
To compare the actual result to the expected one
Required to run JUnit4 tests with the old JUnit runner
Example: set up and tear down

```java
package unitests;

import org.junit.Before; // for the Before annotation
import org.junit.After; // for the After annotation
// other imports as before...

public class AccountTestWithSetUpTearDown {

    private Account account;

    @Before
    public void setUp() {
        account = new Account("John Doe", 30, 1, 1000);
    }

    @After
    public void tearDown() {
        account = null;
    }

    @Test
    public void initialBalance() {
        Assert.assertEquals("Initial balance must be the one set through the constructor", 1000, account.getBalance());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AccountTestWithSetUpTearDown.class);
    }
}
```

- To run this method before any `@Test` method
- To run this method after any `@Test` method
- Must make account an attribute of the class now
A method annotated with `@BeforeClass` will be executed once, before any of the tests in that class is executed.

A method annotated with `@AfterClass` will be executed once, after all of the tests in that class have been executed.

Can have several `@Before` and `@After` methods, but only one `@BeforeClass` and `@AfterClass` method respectively.
Checking for exceptions

Pass a parameter to the @Test annotation stating the type of exception expected:

```java
@Test(expected=AmountNotAvailableException.class) public void overdraft () throws AmountNotAvailableException {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.withdraw(1001);
}
```

The test will fail if a different exception is thrown or if no exception is thrown.
Pass a parameter to the `@Test` annotation setting a timeout period in milliseconds. The test fails if it takes longer than the given timeout.

```java
@Test(timeout=1000)
public void testTimeout () {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.infiniteLoop();
}
```
Part 6: Measuring test quality
Coverage

General notion expressing a percentage of elements (defined by a test strategy) exercised by a test suite. When we say that a certain coverage measure is achieved by a test suite, we mean 100% of the required elements have been exercised.

例："This test suite achieves statement coverage for method m" → every statement in method m is executed by at least one test case in the test suite.
Code coverage - how much of your code is exercised by your tests

Code coverage analysis = the process of:

- Finding sections of code not exercised by test cases
- Creating additional test cases to increase coverage
- Computing a measure of coverage (which is a measure of test suite quality)
Code coverage analyzer

Tool that automatically computes the coverage achieved by a test suite

Steps involved:
1. Source code is instrumented by inserting trace statements.
2. When the instrumented code is run, the trace statements produce a trace file.
3. The analyzer parses the trace file and produces a coverage report (example).
Basic measures of code coverage

Statement coverage - reports whether each executable statement is encountered
  - Disadvantage: insensitive to some control structures

Decision coverage - reports whether boolean expressions tested in control structures evaluate to both true and false
  - Also known as branch coverage

Condition coverage - reports whether each boolean sub-expression (separated by logical-and or logical-or) evaluates to both true and false

Path coverage - reports whether each of the possible paths in each function has been tested
  - Path = unique sequence of branches from the function entry to the exit point
Code coverage tools

Emma
- Java
- Open-source

JCoverage
- Java
- Commercial tool
- [http://www.jcoverage.com/](http://www.jcoverage.com/)

NCover
- C#
- Open-source

Clover, Clover.NET
- Java, C#
- Commercial tools
Dataflow-oriented testing

Focuses on how variables are defined, modified, and accessed throughout the run of the program

**Goal:** to execute certain paths between a definition of a variable in the code and certain uses of that variable
Access-related bugs

Using an uninitialized variable
Assigning to a variable more than once without an intermediate access
Deallocating a variable before it is initialized
Deallocating a variable before it is used
Modifying an object more than once without accessing it
Types of access to variables

Definitions (defs) - the state of a variable is changed (constructor, assignment, modifier method)

Uses - the value of a variable is read (without changing it)
  - Computational uses (c-uses) - the value of the variable is used in a computation
  - Predicative uses (p-uses) - the value of the variable is used in a predicate expression

Kills - any statement (or side effect) that results in a variable being deallocated, undefined, released or no longer visible

Examples:
  - $a = b \times c$  
    - c-use of $b$; c-use of $c$; def of $a$
  - $\text{if } (x > 0)$  
    - p-use of $x$
All measures of dataflow coverage are defined in terms of
the data flow graph

- **Sub-path** in the data flow graph = sequence of
  consecutive nodes
- **Path** in the data flow graph = a sub-path starting at
  the entry node and ending in the exit node
Characterizing paths in the data flow graph in terms of dataflow analysis:

- A sub-path is *def-clear* with respect to a certain variable iff no definition of this variable occurs in the sub-path.

- A sub-path $p$ starting with a def of a variable $v$ is called *du-path* w.r.t. $v$ if $p$ is def-clear w.r.t. $v$ except for the first node and $v$ encounters either a c-use in the last node or a p-use along the last edge of $p$. 
public class Account {

    private int balance;

    public void withdraw(int sum) {
        if (balance >= sum) {
            balance = balance - sum;
            if (balance == 0)
                System.out.println(
                    "There were only " + sum + 
                    "CHF in the account. The account is now empty.");
        } else
            System.out.println(
                "There are less than " + sum + "CHF in the account.");
    }

    ...
}

Example: control flow graph for \texttt{withdraw}

- Definition of sum (0)
  - if balance >= sum (1)
    - True: balance = balance - sum (2)
    - False: print(sum) (5)
  - False: print(sum) (4)

- if balance == 0 (3)
  - True: print(sum) (4)
  - False: print(sum) (5)
Example: data flow graph for `sum` in `withdraw`

Definition of `sum` (0)

- If `balance >= sum` (1)
  - `balance = balance - sum` (2)
    - If `balance == 0` (3)
      - Print `sum` (4)
  - True
    - Print `sum` (4)
- False

- If `balance >= sum` (1)
  - Print `sum` (5)
- True
  - Print `sum` (5)
- False
Definition of sum (0)

if balance >= sum (1)

balance = balance - sum (2)

if balance == 0 (3)

print(sum) (4)

True

print(sum) (5)

False

p-use (1)

True

c-use; def (2)

p-use (3)

True

(4)

(5)

False
Dataflow coverage criteria

**all-defs**: execute *at least one* def-clear sub-path between *every* definition of every variable and *at least one* reachable use of that variable.

**all-p-uses**: execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable p-use of that variable.

**all-c-uses**: execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable c-use of the respective variable.
Dataflow coverage criteria (continued)

all-c-uses/some-p-uses: apply all-c-uses; then if any definition of a variable is not covered, use p-use

all-p-uses/some-c-uses: symmetrical to all-c-uses/some-p-uses

all-uses: execute at least one def-clear sub-path from every definition of every variable to every reachable use of that variable
Example: dataflow coverage criteria for `sum` in `withdraw`

- **all-defs**: at least one def-clear sub-path between every definition and at least one reachable use (0,1)
- **all-p-uses**: at least one def-clear sub-path from every definition to every reachable p-use (0,1)
- **all-c-uses**: at least one def-clear sub-path from every definition to every reachable c-use (0,1,2); (0,1,2,3,4); (0,1,5)
Example: dataflow coverage criteria for `sum` in `withdraw` (continued)

all-c-uses/some-p-uses: apply all-c-uses; then if any definition of a variable is not covered, use p-use (0,1,2); (0,1,2,3,4); (0,1,5)

all-p-uses/some-c-uses: symmetrical to all-c-uses/some-p-uses (0,1)

all-uses: at least one def-clear sub-path from every definition to every reachable use (0,1); (0,1,2);(0,1,2,3,4);(0,1,5)
Specification coverage

**Predicate** = an expression that evaluates to a boolean value
- e.g.: \( a \lor b \lor (f(x) \land x > 0) \)

**Clause** = a predicate that does not contain any logical operator
- e.g.: \( x > 0 \)

**Notation:**
- \( P \) = set of predicates
- \( C_p \) = set of clauses of predicate \( p \)

If specification expressed as predicates on the state, specification coverage translates to **predicate coverage**.
Predicate coverage (PC)

A predicate is covered iff it evaluates to both true and false in 2 different runs of the system.

Example:

\[ a \lor b \lor (f(x) \land x > 0) \]

is covered by the following 2 test cases:

- \{a=true; b=false; f(x)=false; x=1\}
- \{a=false; b=false; f(x)=true; x=-1\}
Clause coverage (CC)

Satisfied if every clause of a certain predicate evaluates to both true and false.

Example:

\[ x > 0 \lor y < 0 \]

Clause coverage is achieved by:

- \( \{x=-1; y=-1\} \)
- \( \{x=1; y=1\} \)
Combinatorial coverage (CoC)

Every combination of evaluations for the clauses in a predicate must be achieved.

Example:

\[( (A \lor B) \land C) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>((A \lor B) \land C)\</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Determination

We say that a major clause \( c_M \) of a predicate \( p \) determines \( p \) if the remaining clauses \( c_m \in p, m \neq M \) (called minor clauses) have such values that changing the truth value of \( c_M \) changes the truth value of \( p \). \( c_M \) will be the active clause.

Example:

\[
p = a \lor b
\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_M = a )</td>
<td>T</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>f</td>
</tr>
<tr>
<td>( c_M = b )</td>
<td>f</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>F</td>
</tr>
</tbody>
</table>
Correlated Active Clause Coverage (CACC)

For each $p \in P$ and each major clause $c_M \in C_p$, choose minor clauses $c_m, m \neq M$ so that $c_M$ determines $p$.

Every $c_M$ must evaluate to both true and false.

The values chosen for the minor clauses must cause $p$ to be true for one value of the major clause $c_M$ and false for the other.

Example:

$p = a \land (b \lor c)$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$a \land (b \lor c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

We satisfy CACC for $a$ if we choose one test case out of rows 1, 2, or 3, and one out of rows 5, 6, or 7.
**Restricted Active Clause Coverage (RACC)**

For each \( p \in P \) and each major clause \( c_M \in C_p \), choose minor clauses \( c_m, m \neq M \) so that \( c_M \) determines \( p \).

Every \( c_M \) must evaluate to both true and false.

The values chosen for the minor clauses must be the same when \( c_M \) is true as when it is false.

**Example:**

\[
p = a \land (b \lor c)
\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>( a \land (b \lor c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

We satisfy RACC for \( a \) if we choose (1,5), or (2,6), or (3,7).
Mutation testing

How do you count the Eggli in the Zürichsee?
Mutation testing

Idea: make small changes to the program source code (so that the modified versions still compile) and see if your test cases fail for the modified versions

Purpose: estimate the quality of your test suite
Faulty versions of the program = mutants

- We only consider mutants that are not equivalent to the original program!

A mutant is said to be killed if at least one test case detects the fault injected into the mutant.

A mutant is said to be alive if no test case detects the injected fault.

A mutation score (MS) is associated to the test set to measure its effectiveness.
Mutation operators

**Mutation operator** = a rule that specifies a syntactic variation of the program text so that the modified program still compiles

**Mutant** = the result of an application of a mutation operator

The quality of the mutation operators determines the quality of the mutation testing process.

**Mutation operator coverage (MOC):** For each mutation operator, create a mutant using that mutation operator.
Examples of mutants

Original program:

```plaintext
if (a < b)
    b := b - a;
else
    b := 0;
```

Mutants:

```plaintext
if (a < b)
    b := b - a;
if (a <= b)
    b := b + a;
else
    b := x - a;
if (c < b)
    b := b - a;
else
    b := 0;
if (a > b)
    b := 1;
a := 0;
```
OO mutation operators

Visibility-related:
- **Access modifier change** - changes the visibility level of attributes and methods

Inheritance-related:
- **Hiding variable/method deletion** - deletes a declaration of an overriding or hiding variable/method
- **Hiding variable insertion** - inserts a member variable to hide the parent’s version
Polymorphism- and dynamic binding-related:

- Constructor call with child class type - changes the dynamic type with which an object is created

Various:

- Argument order change - changes the order of arguments in method invocations (only if there exists an overloading method that can accept the changed list of arguments)
- Reference assignment and content assignment replacement
  - example: list1 = list2.clone()
System test quality (STQ)

- $S$ - system composed of $n$ components denoted $C_i$
- $d_i$ - number of killed mutants after applying the unit test sequence to $C_i$
- $m_i$ - total number of mutants
- The mutation score $MS$ for $C_i$ being given a unit test sequence $T_i$:
  \[ MS(C_i, T_i) = \frac{d_i}{m_i} \]

\[
\text{STQ}(S) = \frac{\sum_{i=1,n} d_i}{\sum_{i=1,n} m_i}
\]

In general, STQ is a measure of test suite quality.

If contracts are used as oracles, STQ is a combined measure of test suite quality and contract quality.
Mutation tools

muJava - [http://ise.gmu.edu/~ofut/muja](http://ise.gmu.edu/~ofut/muja/)


Ma, Y.-S., Kwon, Y.-R., Offutt, J., *Inter-Class Mutation Operators for Java*, 13th International Symposium on Software Reliability Engineering, November 2002
Part 7:
Test case generation
Test case generation: topics

Automated test case generation
Scope
Challenges
Tools
  ➢ AutoTest
  ➢ TestEra
  ➢ Korat
Test case generation: topics

**Unit testing**
- testing of single unit in isolated environment

**Integration testing**
- testing parts of the system by combining modules

**System testing**
- testing of whole system

**Acceptance testing**
- testing if system does what user wants

**Regression testing**
- testing if a system still works
From a survey of 240 software companies in North America and Europe:

- 8% of companies release software to beta sites without any testing.
- 83% of organizations' software developers don't like to test code.
- 53% of organizations' software developers don't like to test their own code because they find it tedious.
- 30% don't like to test because they find testing tools inadequate.
Parts of a test case

Create input
- Instructions
- Data

Execute tests

Evaluate result *(Oracle)*
- Compare
- Compute

(Tear down)
Degrees of automation

- No automation
- Automated execution
- Automated input generation
- Automated oracle
Challenges of automated testing

Vast input space

Is this input good?

- Precondition

Is this output good?

- Postcondition

The quality of the test is only as good as the quality of the assertions
Vast input space

Input space typically unbounded

Even when finite, very large

Exhaustive testing impossible

Number of test cases increases exponentially with number of input variables

foo (c: CHARACTER) is
  do
    ...
  end

bar (c1: CHARACTER, c2: CHARACTER) is
  do
    ...
  end
Automatic testing tools

- TestEra (MIT)
- Korat (MIT)
- AutoTest (ETH)
Automatically generated test cases complement manually written ones:

- Developer may write precondition $A$, while intending precondition $B$.
- Implementation is unlikely to establish postcondition on input satisfying $A \rightarrow \neg B$.
- Test cases will likely only be written for $B$ (and not for $A \rightarrow \neg B$).
## Tool comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>xUnit</th>
<th>TestEra</th>
<th>Korat</th>
<th>AutoTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>generates test cases</td>
<td>✓*</td>
<td>✓*</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>generates oracle</td>
<td>✓*</td>
<td>✓*</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>runs test cases</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>coverage oriented</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓*</td>
</tr>
</tbody>
</table>
``TestEra: A Novel Framework for Automated Testing of Java Programs.''
TestEra: overall scheme
TestEra: characteristics

Requires

- Specification in Alloy
- Concretization function
  - Alloy -> Java
- Abstraction function
  - Java -> Alloy

Alloy: Constraint Solver
TestEra: structure

- **TestEra spec**
  - **Alloy input spec**
  - **Java tester**
  - **Alloy I/O spec**

  - **Alloy Analyzer**
  - **Concretization**
  - **Run Code**
  - **Abstraction**

  - **Alloy instances**
  - **Alloy input**
  - **Java input**
  - **Java output**
  - **Alloy output**

  - **counter example**
    - fail
    - pass

  - **Check correctness**
TestEra: example

Java:

class List {
    int elem;
    List next;
    static List mergeSort(List l) { ... }}

Alloy:

module list
import integer
sig List {
    elem: Integer,
    next: option List}
fun Sorted(l: List) { 
    all n: l.*next | some n.next => n.elem <= n.next.elem } 

fun Perm(l1: List, l2:List) { 
    all e: Integer | #(e.~elem & l1.*next) = 
                   #(e.~elem & l2.*next) } 

fun MergeSortOK(i:List, o:List) { 
    Acyclic(o) 
    Sorted(o) 
    Perm(i,o) } 

static sig Output ext List {} 

fact OutputOK { 
    MergeSortOk(Input, Output) }
fun Acyclic(l: List) {
    all n: l.*next | sole n.~next
    no l.~next  }

static sig Input extends List {}

fact GenerateInputs {
    Acyclic(Input)  }
Korat: overall scheme

1. Specification
2. Precondition
3. Test Case
4. Postcondition
5. Counter Example

Flow:
- Specification → Precondition
- Precondition → Test Case
- Test Case → Postcondition (Passed)
- Postcondition → Counter Example (Failed)
Korat: characteristics

Requires
- Executable predicates
- Bound

Backtracking

Candidate Inputs

Pruning
class BinaryTree
{
    private Node root;
    private int size;
    static class Node
    {
        private Node left;
        private Node right;
    }
    public boolean repOk()
    {
        // class invariant
    }
}
Korat: finitization

Stub generated automatically
Bound

Binary Tree  N0  N1  N2
root  size  left  right  left  right  left  right
4  *  1  *  4  *  4  *  4  *  4  *  4

$2^{14}$ Candidate Inputs
Korat: valid candidate
Korat: invalid candidate
Korat: search

Automatically explores state space

Executes \texttt{repOK()}

- Records fields accessed

Generates next vector through backtracking

Changes only accessed fields

Prunes search space

Implementation of \texttt{repOK()} relevant
AutoTest

- Fully automated testing framework
- Actual strategies are extensions
- Based on Design By Contract
- Robust execution
- Integration of manual unit tests
AutoTest: strategies

Random Strategy
- Use random input

Planning Strategy
- Employ information from postcondition to satisfy preconditions

...
AutoTest: robust execution
AutoTest: manual unit tests

Granularity of automated testing
- routine

Granularity of manual testing
- routine

Discover manual unit tests by ancestor
- Only execute those test cases that are relevant

Advantage
- Single tool
- Single coverage report
- Test case found bug — Manual test suite
Automatic test case generation: assessment

Testing is tedious
Automation can help
Challenges involved
Tools are getting there!
Automatic test case generation: literature

**TestEra**


**Korat**


**AutoTest**

Several articles and online descriptions available from [se.ethz.ch/research/tests.html](http://se.ethz.ch/research/tests.html)
Part 8: Debugging
What is debugging?
What is debugging?

Debugging is the work required to diagnose and correct a bug.

Debugging is not testing:

- Testing focuses on finding bugs
- Debugging is about fixing a bug that has already been identified
A failure is born (1)
A failure is born (2)
A failure is born (3)
Tracking problems

Large projects have many bugs reported
Bugs are not always fixed immediately
Need for Bug tracking system
  - Bugzilla
  - Trac
  - Origo
Classifying problems

- **Severity**
  - Blocker
  - Critical
  - Major
  - Normal
  - Minor
  - Trivial
  - Enhancement

- **Priority**

- **Identifier**

- **Comments**

- **Notifications**
Tom Van Vleck,
ACM SIGSOFT
Software
Engineering Notes,
14/5, July 1989
Testing and bug prevention

“Three questions about each bug you find' (Van Vleck):

- “Is this mistake somewhere else also?"
- “What next bug is hidden behind this one?"
- “What should I do to prevent bugs like this?”
Bug lifecycle

unconfirmed ➔ new ➔ assigned ➔ resolved ➔ verified ➔ closed

unconfirmed ➔ new ➔ assigned ➔ wontfix ➔ worksforme ➔ reopened

valid ➔ duplicate ➔ invalid

duplicate ➔ fixed

resolved ➔ verified ➔ closed
Can you find the bug?

```c
a = compute_value();
printf("a = %d\n", a);
```
Observe failure.

Invent hypothesis, consistent with observation.

Use hypothesis to make prediction.

Test prediction by experiment or observation:

- If prediction satisfied, then refine hypothesis.
- Otherwise, create alternative hypothesis.
Debugging techniques

Delta debugging
Static Slicing
<table>
  <tr>
    <td align=left valign=top>
      <SELECT NAME="op sys" MULTIPLE SIZE=7>
        <OPTION VALUE="All">All
        <OPTION VALUE="Windows 3.1">Windows 3.1
        <OPTION VALUE="Windows 95">Windows 95
        <OPTION VALUE="Windows 98">Windows 98
        <OPTION VALUE="Windows ME">Windows ME
        <OPTION VALUE="Windows 2000">Windows 2000
        <OPTION VALUE="Windows NT">Windows NT
        <OPTION VALUE="Mac System 7">Mac System 7
        <OPTION VALUE="Mac System 7.5">Mac System 7.5
        <OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1
        <OPTION VALUE="Mac System 8.0">Mac System 8.0
        <OPTION VALUE="Mac System 8.5">Mac System 8.5
        <OPTION VALUE="Mac System 8.6">Mac System 8.6
        <OPTION VALUE="Mac System 9.x">Mac System 9.x
        <OPTION VALUE="MacOS X">MacOS X
        <OPTION VALUE="Linux">Linux
        <OPTION VALUE="BSDI">BSDI
        <OPTION VALUE="FreeBSD">FreeBSD
        <OPTION VALUE="NetBSD">NetBSD
        <OPTION VALUE="OpenBSD">OpenBSD
        <OPTION VALUE="AIX">AIX
        <OPTION VALUE="BeOS">BeOS
        <OPTION VALUE="HP-UX">HP-UX
        <OPTION VALUE="IRIX">IRIX
        <OPTION VALUE="Neutrino">Neutrino
        <OPTION VALUE="OpenVMS">OpenVMS
        <OPTION VALUE="OS/2">OS/2
        <OPTION VALUE="OSF/1">OSF/1
        <OPTION VALUE="Solaris">Solaris
        <OPTION VALUE="SunOS">SunOS
        <OPTION VALUE="other">other
      </SELECT>
    </td>
    <td align=left valign=top>
      <SELECT NAME="priority" MULTIPLE SIZE=7>
        <OPTION VALUE="--">--
        <OPTION VALUE="P1">P1
        <OPTION VALUE="P2">P2
        <OPTION VALUE="P3">P3
        <OPTION VALUE="P4">P4
        <OPTION VALUE="P5">P5
      </SELECT>
    </td>
    <td align=left valign=top>
      <SELECT NAME="bug severity" MULTIPLE SIZE=7>
        <OPTION VALUE="blocker">blocker
        <OPTION VALUE="critical">critical
        <OPTION VALUE="major">major
        <OPTION VALUE="normal">normal
        <OPTION VALUE="minor">minor
        <OPTION VALUE="trivial">trivial
        <OPTION VALUE="enhancement">enhancement
      </SELECT>
    </td>
  </tr>
</table>
Bug example: Mozilla

Looking at the input it is hard to understand the real cause of the bug
Can we simplify the input?
Delta debugging: characteristics

Simplification algorithm for bug reproducing examples
Reduces size of input or program
Easy to implement and customize

Assumptions
- Input can be split into parts
- Working program
- Failing program
Delta debugging example (1)

Assume the following makes Mozilla crash:

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

Approach:
- Remove parts of input and see if it still crashes
Delta debugging example (2)

1. <SELECT NAME="priority" MULTIPLE SIZE=7> F
2. <SELECT NAME="priority" MULTIPLE SIZE=7> P
3. <SELECT NAME="priority" MULTIPLE SIZE=7> P
4. <SELECT NAME="priority" MULTIPLE SIZE=7> P
5. <SELECT NAME="priority" MULTIPLE SIZE=7> F
6. <SELECT NAME="priority" MULTIPLE SIZE=7> F
7. <SELECT NAME="priority" MULTIPLE SIZE=7> P
8. <SELECT NAME="priority" MULTIPLE SIZE=7> P
9. <SELECT NAME="priority" MULTIPLE SIZE=7> P
10. <SELECT NAME="priority" MULTIPLE SIZE=7> F
11. <SELECT NAME="priority" MULTIPLE SIZE=7> P
12. <SELECT NAME="priority" MULTIPLE SIZE=7> P
13. <SELECT NAME="priority" MULTIPLE SIZE=7> P

Bold parts remain in the input
Delta debugging example (3)

14  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
15  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
16  `<SELECT NAME="priority" MULTIPLE SIZE=7>` F
17  `<SELECT NAME="priority" MULTIPLE SIZE=7>` F
18  `<SELECT NAME="priority" MULTIPLE SIZE=7>` F
19  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
20  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
21  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
22  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
23  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
24  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
25  `<SELECT NAME="priority" MULTIPLE SIZE=7>` P
26  `<SELECT NAME="priority" MULTIPLE SIZE=7>` F
After 26 tries we found:

\[ \text{<SELECT>} \]

causes Mozilla to crash.

Delta debugging completely automates this process. Can this be applied only to input?
Delta debugging: limitations

Delta debugging does not guarantee smallest possible example.

- It only guarantees an example where every line is relevant.

We need to be able to replay inputs
We need to be able to split inputs
Empty input must not trigger failure
Program slicing

Decomposition technique that extracts statements relevant to a particular computation from a program.
Program slicing: applicability

Debugging
  ➢ Find relevant part of program

Testing
  ➢ Find relevant test cases after change

Formal Verification
  ➢ Find relevant part of program that needs to be proven
Static slicing

Program slice contains the statements in program that affect value of a variable computed at some point of interest.

Static slice
- Valid for all possible runs

Dynamic slice
- Assumes specific execution of program
Static slicing: steps involved

from
  s0
until p1 do
  s1
  if p2 then
    s2
  else
    s3
  end
end

Create Program Dependence Graph
Traverse Program Dependence Graph
Data dependence

Statement B is data dependent on statement A if:

- A writes to variable V that is being read by B
- There is at least one path in control flow graph from A to B in which V is not written to by other statement.

1. \( y := 2 + w \)
2. \( u := 2 \times w \)
3. \( x := 2 \times y \)
Control dependence

Statement S is directly control dependent on predicate P if:

- Value of P determines whether or not S will be executed.

```
from
s0
until p1 do
  s1
  if p2 then
    s2
  else
    s3
  end
end
```
Backward slice example

\[
\begin{align*}
\text{n} & := \text{read} \\
\text{i} & := 1 \\
\text{sum} & := 0 \\
\text{product} & := 1 \\
\text{while } i \leq n \text{ do} \\
& \quad \text{sum} := \text{sum} + 1 \\
& \quad \text{product} := \text{product} + i \\
& \quad i := i + 1 \\
\text{end} \\
\text{print (sum)} \\
\text{print (product)}
\end{align*}
\]
Static slice

Slicing Criterion:
- Program Point \((S_i)\)
- Set of variables \((V)\)

Static Slice \(S(S_i, V)\):
- Transitive closure of data and control dependences of \(V\) at \(S_i\)
- Set of statements on which the values of \(V\) at \(S_i\) are directly or indirectly data or control depend.
Debugging: assessment

Debugging Failures
Problem Management
Scientific Debugging Techniques
  - Delta debugging
  - Program Slicing
Debugging: literature


Debugging: resources

Trac: http://www.issue-tracker.com
Bugzilla: http://www.bugzilla.com
Delta debugging:
  - http://www.st.cs.uni-sb.de/dd
  - http://freshmeat.net/projects/delta
Slicing: http://www.codesurfer.com
AskIgor: http://www.st.cs.uni-sb.de/askigor/
Part 9:
Quality Assurance & testing in the development process
Quality assurance: other aspects

Concurrent and real-time systems

Assessing non-functional properties:
- Performance
- Reliability
- Complexity

Techniques not studied: symbolic analysis, abstract interpretation
In the lifecycle

Quality assurance is:
- A continuous activity
- The exclusive target of several phases
- A mindset

Who performs quality assurance?
- Developers
- Quality assurance team
- Users
Some key lessons

A panoply of complementary techniques
- Static: proofs, static analysis, model-checking
- Dynamic: testing
- Methodology-oriented: Design by Contract, proof-based development
- Process-oriented: CMMI, open-source, XP

They are converging!

Components are the best place to apply all this

Quality has been improving, but we are not there yet, by far

Quality is not free, but non-quality is more expensive
("There’s never any time to do it right, but there’s always time to do it over")

Management role is essential: Software Quality Policy
The verification Grand Challenge

Source: Tony Hoare, now Microsoft Research
VSTTE conference (Verified Software: Tools, Theories, Experiments), ETH Zurich, October 2005

vstte.ethz.ch

See also NZZ:

www.nzz.ch/2005/10/14/em/articleD825L.print.html

Goal: within the next 15 years, produce a “Verifying Compiler” which, when compiling a program, guarantees that it is correct.