Software Verification and Validation

Prof. Lionel Briand
Ph.D., IEEE Fellow
Introduction to Software Testing
Main Sources

• M. Pezze and M. Young, Software Analysis and Software Testing, Wiley, 2007
• Vahid Garousi, U. of Calgary
Examples of Software Failures

- **Communications:** Loss or corruption of communication media, non delivery of data.
- **Space Applications:** Lost lives, launch delays, e.g., European Ariane 5 shuttle, 1996:
  - From the official disaster report: “Due to a malfunction in the control software, the rocket veered off its flight path 37 seconds after launch.”
- **Defense and Warfare:** Misidentification of friend or foe.
- **Transportation:** Deaths, delays, sudden acceleration, inability to brake.
- **Electric Power:** Death, injuries, power outages, long-term health hazards (radiation).
Examples of Software Failures (cont.)

• **Money Management:** Fraud, violation of privacy, shutdown of stock exchanges and banks, negative interest rates.
• **Control of Elections:** Wrong results (intentional or non-intentional).
• **Control of Jails:** Technology-aided escape attempts and successes, failures in software-controlled locks.
• **Law Enforcement:** False arrests and imprisonments.
On June 4, 1996, the flight of the Ariane 5 launcher ended in a failure.

Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3,700 m, the launcher veered off its flight path, broke up and exploded.
Ariane 5 - Root Cause

• Source: ARIANE 5 Flight 501 Failure, Report by the Inquiry Board

A program segment for converting a floating point number to a signed 16 bit integer was executed with an input data value outside the range representable by a signed 16-bit integer. This run time error (out of range, overflow), which arose in both the active and the backup computers at about the same time, was detected and both computers shut themselves down. This resulted in the total loss of attitude control. The Ariane 5 turned uncontrollably and aerodynamic forces broke the vehicle apart. This breakup was detected by an on-board monitor which ignited the explosive charges to destroy the vehicle in the air. Ironically, the result of this format conversion was no longer needed after lift off.
Ariane 5 - Lessons Learned

• Adequate exception handling and redundancy strategies (real function of a backup system, degraded modes?)

• Clear, complete, documented specifications (e.g., preconditions, post-conditions)

• But perhaps more importantly: usage-based testing (based on operational profiles), in this case actual Ariane 5 trajectories

• Note this was not a complex, computing problem, but a deficiency of the software engineering practices in place ...
F-18 crash

• An F-18 crashed because of a missing exception condition:
  An if ... then ... block without the else clause that was thought could not possibly arise.

• In simulation, an F-16 program bug caused the virtual plane to flip over whenever it crossed the equator, as a result of a missing minus sign to indicate south latitude.
Fatal Therac-25 Radiation

• In 1986, a man in Texas received between 16,500-25,000 radiations in less than 10 sec, over an area of about 1 cm.

• He lost his left arm, and died of complications 5 months later.
Power Shutdown in 2003

508 generating units and 256 power plants shut down

Affected 10 million people in Ontario, Canada

Affected 40 million people in 8 US states

Financial losses of $6 Billion USD

The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system

© Lionel Briand 2010
Testing
Definitions & Objectives
Test Stubs and Drivers

• Test Stub: Partial implementation of a component on which a unit under test depends.

• Test Driver: Partial implementation of a component that depends on a unit under test.

• Test stubs and drivers enable components to be isolated from the rest of the system for testing.
Summary of Definitions

Test suite

Test case

Component

Correction

Test stub

Test driver

Failure

Fault

Error

exercises

is revised by

finds

repairs

is caused by

is caused by

is caused by

© Lionel Briand 2010
Motivations

• No matter how rigorous we are, software is going to be faulty
• Testing represents a substantial percentage of software development costs and time to market
• Impossible to test under all operating conditions - based on incomplete testing, we must gain confidence that the system has the desired behavior
• Testing large systems is complex - it requires strategy and technology and is often done inefficiently in practice
The Testing Dilemma

Available testing resources

All Software System functionality

Faulty functionality

Potentially thousands of items to test

© Lionel Briand 2010
Testing Process Overview

SW Representation (e.g., models, requirements)

Derive Test cases

Execute Test cases

SW Code

Get Test Results

Test Oracle

Compare

[Test Result==Oracle]

[Test Result! = Oracle]
Types of Testing

• **Functional Testing**: testing functional requirements.
  - Functional requirements specify specific behavior or functions of a software.
  - Functional testing is thus checking the correct functionality of a system.

• **Non-functional Testing**: testing non-functional requirements.
  - Non-functional requirements:
    • Specify criteria that can be used to judge the operation of a system, rather than specific behaviors.
    • Typical non-functional requirements are performance, reliability, scalability, and cost.
    • Non-functional requirements are often called the *ilities* of a system.
    • Examples next...
Non-functional SW Requirements

- Accessibility
- Availability
- Efficiency (resource consumption for given load)
- Effectiveness (resulting performance in relation to effort)
- Extensibility
- Maintainability
- Performance / Response time

- Resource constraints (required processor speed, memory, disk space, network bandwidth, etc.)
- Reliability (e.g. Mean Time Between Failures - MTBF)
- Robustness
- Safety
- Scalability
- Security
Qualities of Testing

• **Effective** at uncovering faults
• **Help** *locate* faults for debugging
• **Repeatable** so that a precise understanding of the fault can be gained
• **Automated** so as to lower the cost and timescale
• **Systematic** so as to be predictable in terms of its effect on dependability
Continuity Property

- Problem: Test a bridge ability to sustain a certain weight
- Continuity Property: If a bridge can sustain a weight equal to \( W_1 \), then it will sustain any weight \( W_2 \leq W_1 \)
- Essentially, continuity property = small differences in operating conditions should not result in dramatically different behavior

- BUT, the same testing property cannot be applied when testing software, why?
- In software, small differences in operating conditions can result in dramatically different behavior (e.g., value boundaries)
- Thus, the continuity property is not applicable to software
Subtleties of Software Dependability

- **Dependability**: Correctness, reliability, safety, robustness
- A program is correct if it obeys its specification.
- Reliability is a way of statistically approximating correctness.
- Safety implies that the software must always display a safe behavior, under any condition.
- A system is robust if it acts reasonably in severe, unusual or illegal conditions.
Subtleties of Software Dependability

- **Correct but not safe or robust**: the specification is inadequate
- **Reliable but not correct**: failures rarely happen
- **Safe but not correct**: annoying failures may happen
- **Reliable and robust but not safe**: catastrophic failures are possible
Traffic Light Controller

• Correctness, Reliability:
The system should let traffic pass according to the correct pattern and central scheduling on a continuous basis.

• Robustness:
The system should provide degraded functionality in the presence of abnormalities.

• Safety:
It should never signal conflicting greens.

An example degraded function: the line to central controlling is cut-off and a default pattern is then used by local controller.
Dependability Needs Vary

- Safety-critical applications
  - flight control systems have strict safety requirements
  - telecommunication systems have strict robustness requirements
- Mass-market products
  - dependability is less important than time to market
- Can vary within the same class of products:
  - reliability and robustness are key issues for multi-user operating systems (e.g., UNIX) less important for single users operating systems (e.g., Windows or MacOS)
Fundamental Principles
Exhaustive Testing

- Exhaustive testing, i.e., testing a software system using all the possible inputs, is most of the time impossible.

- Examples:
  - A program that computes the factorial function (n!=n.(n-1). (n-2)...1)
    - Exhaustive testing = running the program with 0, 1, 2, ..., 100, ... as an input!
  - A compiler (e.g., javac)
    - Exhaustive testing = running the (Java) compiler with any possible (Java) program (i.e., source code)
Input Equivalence Classes

- General principle to reduce the number of inputs
  - Testing criteria group input elements into (equivalence) classes
  - One input in selected in each class (notion of test coverage)
Test Coverage

Software Representation (Model)

Associated Criteria
Test cases must cover all the … in the model

Test Data

Representation of

• the **specification** ⇒ Black-Box Testing
• the **implementation** ⇒ White-Box Testing
Complete Coverage: White-Box

if \( x > y \) then
    Max := x;
else
    Max := x;  // fault!
end if;

\{x=3, y=2; x=2, y=3\} can detect the error, more “coverage”
\{x=3, y=2; x=4, y=3; x=5, y=1\} is larger but cannot detect it

- Testing criteria group input domain elements into (equivalence) classes (control flow paths here)
- Complete coverage attempts to run test cases from each class
Complete Coverage: Black-Box

• **Specification of Compute Factorial Number:** If the input value \( n \) is < 0, then an appropriate error message must be printed. If \( 0 \leq n < 20 \), then the exact value of \( n! \) must be printed. If \( 20 \leq n < 200 \), then an approximate value of \( n! \) must be printed in floating point format, e.g., using some approximate method of numerical calculus. The admissible error is 0.1% of the exact value. Finally, if \( n \geq 200 \), the input can be rejected by printing an appropriate error message.

• Because of expected variations in behavior, it is quite natural to divide the input domain into the classes \( \{ n < 0 \} \), \( \{ 0 \leq n < 20 \} \), \( \{ 20 \leq n < 200 \} \), \( \{ n \geq 200 \} \). We can use one or more test cases from each class in each test set. Correct results from one such test set support the assertion that the program will behave correctly for any other class value, but there is no guarantee!
Black vs. White Box Testing

System

Specification

Implementation

Missing functionality:
Cannot be revealed by white-box techniques

Unexpected functionality:
Cannot be revealed by black-box techniques
White-box vs. Black-box Testing: Pro’s and Con’s

• **Black-box**
  - Check conformance with specifications
  - It scales up (different techniques at different granularity levels)
  - It depends on the specification notation and degree of detail
  - Do not 'exactly' know how much of the system is being tested
  - Do not detect unspecified task.

• **White-box**
  - It allows you to be confident about test coverage
  - It is based on control or data flow coverage
  - It does not scale up (mostly applicable at unit and integration testing levels)
  - Unlike black-box techniques, it cannot reveal missing functionalities (part of the specification that is not implemented)
Theoretical Foundations of Testing:
Formal definitions

- **Let** \( P \) **be a program**
- **Let** \( D \) **and** \( R \) **denote its input domain and range**, i.e.:
  - \( P \) **is a function**: \( D \rightarrow R \)
- **Let** \( OR \) **denote the expected output values** (**ORacle**)
- **\( P \) is said to be correct** for all \( d \in D \) if \( P(d) \) satisfies \( OR \); if not, we have a **failure**
- **A test case** is an element \( d \) of \( D \) and the expected value of \( P \) in \( OR \) given \( d \)
- **A test set** (a.k.a. suite) \( T \) **is a finite set of test cases**
Theoretical Foundations of Testing:
Test Adequacy Criterion

• A test adequacy criterion $C$ is a subset of $P_D$, where $P_D$ is the set of all finite subsets of $D$ that we could target when devising test sets (note the two levels of subsets). (Recall: $D$ denotes the input domain of $P$).
• In simple words: how “much” of $D$ we should target by our test set.
• Example: Defining a criterion $C$ for a program model $M$
  - $M$: the Control Flow Graph (CFG) of a function
  - $C$: the set of all the edges in the CFG
• The coverage ratio of a test set $T$ is the proportion of the elements in $M$ defined by $C$ covered by the given test set $T$.
• A test set $T$ is said to be adequate for $C$, or simply $C$-adequate, when the coverage ratio achieves 100% for criterion $C$.
• We will have examples soon...
Theoretical Foundations of Testing: Hierarchy of Adequacy Criteria

- Let us define a subsumption relationship between different test criteria associated with a program model.
- Given a model M, and two criteria C1 and C2 for that model:
  - C1 subsumes C2 if any C1-adequate test set is also C2-adequate.

- Example: Consider criteria all-transitions and all-paths for finite state machines:
  - all-paths subsumes all-transitions

- If C1 subsumes C2, we assume:
  - Satisfying C1 is more “expensive” (e.g., # of test cases) than satisfying C2
  - C1 allows the detection of more faults (on average) than C2

© Lionel Briand 2010
Theoretical Foundations of Testing

Ideal Test Set

- A test set $T$ is said to be ideal if, whenever $P$ is incorrect, there exists a $d \in T$ such that $P$ is incorrect for $d$, i.e., $P(d)$ does not satisfy OR.

- If $T$ is an ideal test set and $T$ is successful for $P$, then $P$ is correct.

- $T$ satisfies a test adequacy criterion $C$ if its input values belong to $C$. 
Theoretical Foundations of Testing: Test Consistency and Completeness

• **A test adequacy criterion** $C$ is **consistent** if, for any pair of test sets $T_1$ and $T_2$, both satisfying $C$, $T_1$ is successful if and only $T_2$ is.

• **A test adequacy criterion** $C$ is **complete** if, whenever $P$ is incorrect, there is an unsuccessful test set that satisfies $C$.

• If $C$ is consistent and complete, any test set $T$ satisfying $C$ is **ideal** and could be used to decide $P$’s correctness.

• The problem is that it is not possible in general to derive algorithms that helps determine whether a criterion, a test set, or a program has any of the above mentioned properties ... they are undecidable problems 😞

• As we discussed, it is impossible to determine (find) consistent and complete test criteria from the theoretical standpoint.

• Also, exhaustive testing cannot be performed in practice.

• Therefore, we need test strategies that have been empirically investigated.

• A significant test case is a test case with high error detection probability - its execution increases our confidence in the program correctness.

• The goal is to run a sufficient number of significant test cases - and that number should be as small as possible (to save time and $$).
Practical Aspects
Many Causes of Failures

• The specification may be wrong or have a missing requirement
• The specification may contain a requirement that is impossible to implement given the prescribed software and hardware
• The system design may contain a fault
• The program code may be wrong
Test Organization

- Many different potential causes of failure,
  Large systems -> testing involves several stages
- Module, component, or unit testing
- Integration testing
- Function test
- Performance test
- Acceptance test
- Installation test
Unit test

Design descriptions
System functional specifications
Other software specifications
Customer requirements
User environment

Component code

Tested component

Integration test
Function test
Performance test
Acceptance test
Installation test

Integrated modules
Functioning system
Verified, validated software
Accepted system

Pfleeger, 1998

SYSTEM IN USE!

© Lionel Briand 2010
Unit Testing

• (Usually) performed by each developer.
• Scope: Ensure that each module (i.e., class, subprogram) has been implemented correctly.
• Often based on White-box testing.

• A unit is the smallest testable part of an application.
• In procedural programming, a unit may be an individual subprogram, function, procedure, etc.
• In object-oriented programming, the smallest unit is a method; which may belong to a base/super class, abstract class or derived/child class.
Deriving Unit Test Cases

Unit to be tested

local data structures

boundary conditions

error handling paths

interface

... test cases

© Lionel Briand 2010
Integration/Interface Testing

- Performed by a small team.
- **Scope**: Ensure that the interfaces between components (which individual developers could not test) have been implemented correctly, e.g., consistency of parameters, file format.

  Test cases have to be planned, documented, and reviewed.
  - Performed in a relatively small time-frame.
Integration Testing Failures

Integration of well tested components may lead to failure due to:

• Bad use of the interfaces (bad interface specifications / implementation)
• Wrong hypothesis on the behavior/state of related modules (bad functional specification / implementation), e.g., wrong assumption about return value
• Use of poor drivers/stubs: a module may behave correctly with (simple) drivers/stubs, but result in failures when integrated with actual (complex) modules.
System Testing

• Performed by a separate group within the organization (Most of the times).

• **Scope:** Pretend *we* are the end-users of the product.

• Focus is on functionality, but may also perform many other types of non-functional tests (e.g., recovery, performance).

• Black-box form of testing, but code coverage can be monitored.

• Test case specification driven by system’s use-cases.
Differences among Testing Activities

**Unit Testing**
- From module specifications
- Visibility of code details
- Complex scaffolding
- Behavior of single modules

**Integration Testing**
- From interface specifications
- Visibility of integr. Struct.
- Some scaffolding
- Interactions among modules

**System Testing**
- From requirements specs
- No visibility of code
- No drivers/stubs
- System functionalities

Pezze and Young, 2007
System vs. Acceptance Testing

- System testing
  - The software is compared with the requirements specifications (verification)
  - Usually performed by the developers, who know the system

- Acceptance testing
  - The software is compared with the end-user requirements (validation)
  - Usually performed by the customer (buyer), who knows the environment where the system is to be used
  - Sometime distinguished between α - β-testing for general purpose products
Testing throughout the Lifecycle

• Much of the life-cycle development artifacts provides a rich source of test data
• Identifying test requirements and test cases early helps shorten the development time
• They may help reveal faults
• It may also help identify early low testability specifications or design
Testing throughout the Lifecycle - in the “V” model
(half of this is about testing!)
Testing throughout the Lifecycle - in Agile Development

Even here!
Testing throughout the Lifecycle - in TDD
(Test-Driven Development)

The TDD Process

1. Add a test
2. Run tests; see new failure
3. Write some code
4. Run tests; see all pass
5. Refactor

In TDD, test cases are actually the requirements of the system.
Types of testing

One possible classification is based on the following four classifiers:

C1: Source of test case generation.

C2: Lifecycle phase in which testing takes place

C3: Goal of a specific testing activity

C4: Characteristics of the artifact under test
# C1: Source of test generation

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements (informal)</td>
<td>Black-box</td>
<td>Ad-hoc testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boundary value analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Category partition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classification trees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cause-effect graphs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Equivalence partitioning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Partition testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Predicate testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random testing</td>
</tr>
<tr>
<td>Code</td>
<td>White-box</td>
<td>Adequacy assessment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coverage testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data-flow testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Domain testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mutation testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Path testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Structural testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test minimization using coverage</td>
</tr>
<tr>
<td>Requirements and code</td>
<td>Black-box and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>White-box</td>
<td></td>
</tr>
<tr>
<td>Formal model:</td>
<td>Model-based</td>
<td>Statechart testing</td>
</tr>
<tr>
<td>Graphical or mathematical</td>
<td>Specification</td>
<td>FSM testing</td>
</tr>
<tr>
<td>specification</td>
<td></td>
<td>Pairwise testing</td>
</tr>
<tr>
<td>Component interface</td>
<td>Interface testing</td>
<td>Interface mutation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pairwise testing</td>
</tr>
</tbody>
</table>
C2: Lifecycle phase in which testing takes place

<table>
<thead>
<tr>
<th>Phase</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding</td>
<td>Unit testing</td>
</tr>
<tr>
<td>Integration</td>
<td>Integration testing</td>
</tr>
<tr>
<td>System integration</td>
<td>System testing</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Regression testing</td>
</tr>
<tr>
<td>Post system, pre-release</td>
<td>Beta-testing</td>
</tr>
</tbody>
</table>
## C3: Goal of specific testing activity

<table>
<thead>
<tr>
<th>Goal</th>
<th>Technique</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advertised features</td>
<td>Functional testing</td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>Security testing</td>
<td></td>
</tr>
<tr>
<td>Invalid inputs</td>
<td>Robustness testing</td>
<td></td>
</tr>
<tr>
<td>Vulnerabilities</td>
<td>Vulnerability testing</td>
<td></td>
</tr>
<tr>
<td>Errors in GUI</td>
<td>GUI testing</td>
<td>Capture/plaback</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Event sequence graphs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complete Interaction Sequence</td>
</tr>
<tr>
<td>Operational correctness</td>
<td>Operational testing</td>
<td>Transactional-flow</td>
</tr>
<tr>
<td>Reliability assessment</td>
<td>Reliability testing</td>
<td></td>
</tr>
<tr>
<td>Resistance to penetration</td>
<td>Penetration testing</td>
<td></td>
</tr>
<tr>
<td>System performance</td>
<td>Performance testing</td>
<td>Stress testing</td>
</tr>
<tr>
<td>Customer acceptability</td>
<td>Acceptance testing</td>
<td></td>
</tr>
<tr>
<td>Business compatibility</td>
<td>Compatibility testing</td>
<td>Interface testing</td>
</tr>
<tr>
<td>Peripherals compatibility</td>
<td>Configuration testing</td>
<td>Installation testing</td>
</tr>
</tbody>
</table>
## C4: Artifact under test

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application component</td>
<td>Component testing</td>
</tr>
<tr>
<td>Client and server</td>
<td>Client-server testing</td>
</tr>
<tr>
<td>Compiler</td>
<td>Compiler testing</td>
</tr>
<tr>
<td>Design</td>
<td>Design testing</td>
</tr>
<tr>
<td>Code</td>
<td>Code testing</td>
</tr>
<tr>
<td>Database system</td>
<td>Transaction-flow testing</td>
</tr>
<tr>
<td>OO software</td>
<td>OO testing</td>
</tr>
<tr>
<td>Operating system</td>
<td>Operating system testing</td>
</tr>
<tr>
<td>Real-time software</td>
<td>Real-time testing</td>
</tr>
<tr>
<td>Requirements</td>
<td>Requirement testing</td>
</tr>
<tr>
<td>Software</td>
<td>Software testing</td>
</tr>
<tr>
<td>Web service</td>
<td>Web service testing</td>
</tr>
</tbody>
</table>

© Aditya P. Mathur 2009
Testing Activities BEFORE Coding

- Testing is a time consuming activity
- Devising a test strategy and identify the test requirements represent a substantial part of it
- Planning is essential
- Testing activities undergo huge pressure as it is run towards the end of the project
- In order to shorten time-to-market and ensure a certain level of quality, a lot of QA-related activities (including testing) must take place early in the development life cycle
Testing takes creativity

• Many jobs out there in test automation
• To develop an effective test, one must have:
  • Detailed understanding of the system
  • Knowledge of testing techniques
  • Skill to apply these techniques in an effective and efficient manner
• Testing is done best by independent testers
• Programmer often stick to the data set that makes the program work
• A program often does not work when tried by somebody else.