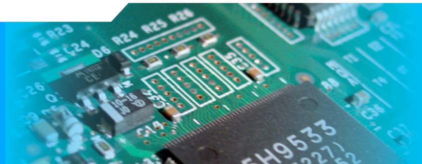
Generalities on software validation

Noël PLOUZEAU
IRISA/ISTIC



Testing is vital

- Developing a system without testing it seriously is asking for trouble
  - Personal example: the *La deuxième tête* startup case
- Software testing has always been lagging behind other domains (eg electronics, mechanical engineering)
  - But it is now a well accepted practice
- This is even a specific trade
  - Third party testing/validation

Size of modern software

- Common car: 5 millions of lines of code (MLOC)
- Airliner: 50 MLOC
- High end car: 100 MLOC
- Estimation of 2000 to 7000 defects per MLOC
  - No need to panic
  - Severity x probability x population
  - This is typical of an engineering approach

Problems at every scale

- Testing trivial code is difficult
  - read n (positive integer)
  - while n > 1 do
    - if n is even then n := n / 2
    - else n := 3*n + 1
  - end while
  - print “Finished!”

Problems at every scale (cont'd)

- Can we prove that for every n "Finished!" will be printed at some time?
- Answer: no! Undecidable problem…
- Let's try all possibilities then
  - for a 32 bits integer : 2 billions of value
  - we now have 64 bits processors, even in smartphones

Programming in the large

- Cars
  - now they have dozens of processors (ECUs)
  - concurrent and real time execution
  - Life critical (braking, steering)
- Google
  - number of servers: about a million
  - several billions of search requests, doc edition, etc per day

Programming in the duration

- About 25% of system are more than 10 years old
- Average age of a system: 7 years
- This means that these systems have been modified extensively in the past 10 years
- Therefore we must apply life long testing

Code reuse

- Very good reasons to reuse code
  - reliability: well-tried code is more reliable
  - economy: no need to start all over
- However one cannot reuse a code module that was not carefully designed for this

An example (famous Ariane example)

- Ariane 501 maiden flight (1996-06-04 T 12:34Z)
- Self destruction triggered at H0+39s
- Cost: 500 millions of euros

Chain of events in Ariane's case

- The rocket has two inertial reference system for navigation
- Conversion error in inertial reference system #2
  - Because the value read from the accelerometer was off bounds for a 16 bits variable
- Handling of this error
  - exception raised
  - while debug mode was still active: memory dumped on the main rocket hardware bus!

Chain of events (cont'd)

- Then autopilot reads phony values from the dump
- It then gives erroneous orders to nozzle motors
- Ariane 5 starts to steer crabwise and beyond acceptable limits of trajectory angle
- Self destruction is triggered to shred the rocket before it falls back on earth

Why all this?

- Code from Ariane 4 was used verbatim in Ariane 5’s software (“it has always worked well”)
- At the time of failure this code should not have been running (design left over from Ariane 4)
- Tests of the code reused were not exhaustive and realistic enough: bug could have been easily discovered
- Preconditions for this code were not explicit in documentation (not following CBD)
- In short: major process and design flaws because of lack of training and experience in software engineering

Other major software failures

- Therac 25 (1985-1987): radiotherapy
- USS Yorktown (1998): engine shutdown due to a zero divide error
- Mars Climate Orbiter (1999): lost because of unspecified heterogeneous unit use
- Sciaparelli Mars probe: crashed because input data requirements in navigation were not specified properly (and of course not checked before using it)

Therac 25 process problems

- The software code was not independently reviewed
- The software design was not documented with enough detail to support reliability modeling
- The software was written in assembly language. While this was more common at the time than it is today, assembly language is harder to debug than high-level languages.

Design problems

- No hardware interlocks to prevent the electron- beam from operating in its high-energy mode without the target in place
- **Software from older models had been reused without properly considering the hardware differences**
- The software assumed that sensors always worked correctly, since there was no way to verify them (see open loop)
- Arithmetic overflows could cause the software to bypass safety checks.

Usability problems

- The system documentation did not adequately explain error codes.
- AECL personnel were at first dismissive of complaints.

Costs of validation, per phase

- Requirements definition: 10%
- Design: 20%
- Coding: 20%
- Validation (testing, code review, etc): 50%
- When taking maintenance cost into account: 80% of total costs, often more (technical debts are costly)

Confidence vs cost

- Validation aims at checking conformity of a product with respect to its specification
  - verification & proof: no bugs left
  - testing: an acceptable number of bugs left

Cost of validation

- Low cost
- Short time
- High confidence
- **Pick two out of three only!**

The 50% rule

- 50% of software engineers begin their career as testers
- 50% of startups fail because of testing problems
  - poorly designed testing process
  - problematic maintenance
  - frequent regression
  - all in all a huge quality management problem

What is testing?

- Testing is
  - a manual or automated process,
  - used to check that a system satisfies properties required by its **specification**,
  - or to exhibit differences between results
    - **produced** by the system and
    - results **expected** from the specification

Key points in this definition

- Specification of expected properties
  - No specs, no tests, no sale
  - Requirements engineering (a trade by itself)
- Expected versus actual results
- Manual or automated process
  - who really wants to do it manually...

Try to see if it works

- Try
  - how does the system work
  - how to run it
  - how to monitor it

Try to see if it works (cont'd)

- See
  - what properties should be monitored
    - there are many properties in a real system
  - how to monitor them
    - how to get access to them
    - without inducing side-effects that spoil the tests

Try to see if it works (cont'd)

- If it works
  - how do we know "it works"
  - what are the possible differences between expected and actual properties
  - how can we detect them

Software quality (eg ISO 9126)

- In the large testing a system implies evaluation of the score of it with respect to norm such as ISO 9126
  - Usability evaluation is a specific domain that involves ergonomics, psychology,...
  - Robustness involves cybersecurity engineering
  - Deployment testing involve system engineering, etc

A remark on devops

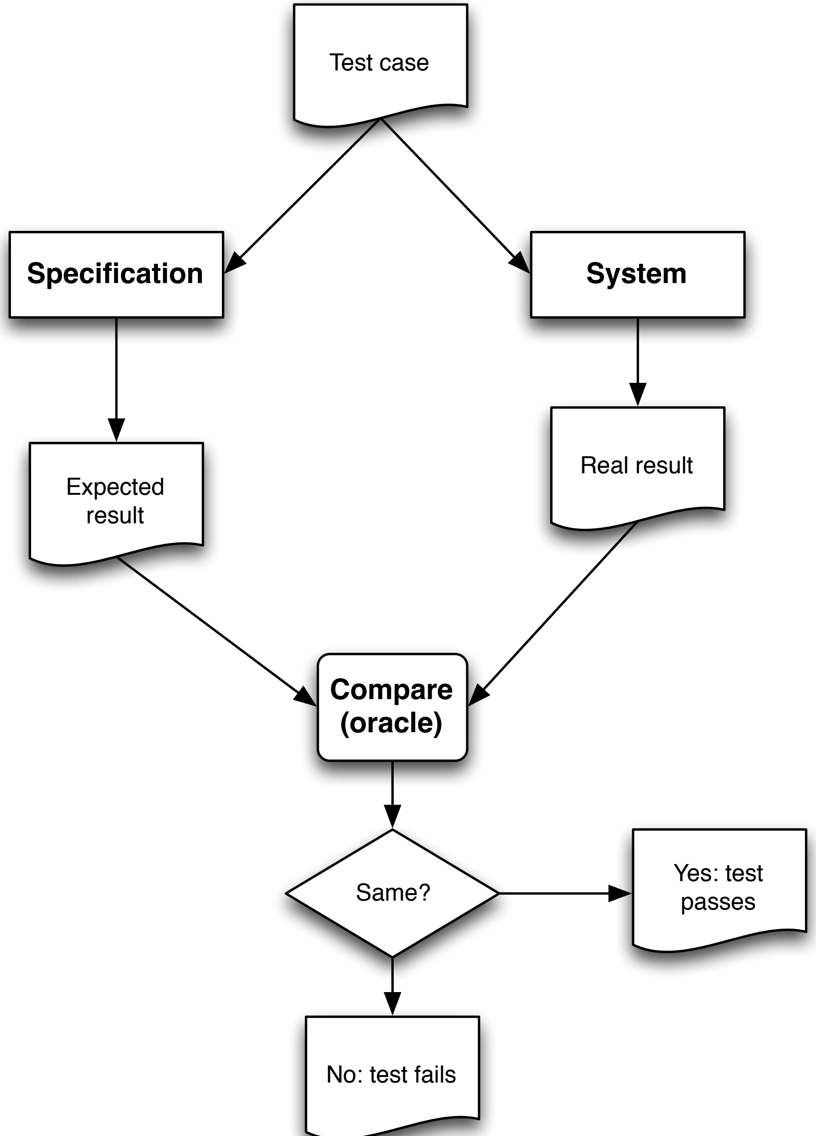
- Large and modern companies rely on fast development to operation cycles (devops)
- For instance the booking.com company's engineers
  - deploy a new version of the site every 15 minutes
  - monitor continuously and automatically the behaviour of the system and its users
  - roll back or correct the system within minutes
- That is an extreme continuous integration approach (the users are the end testers)

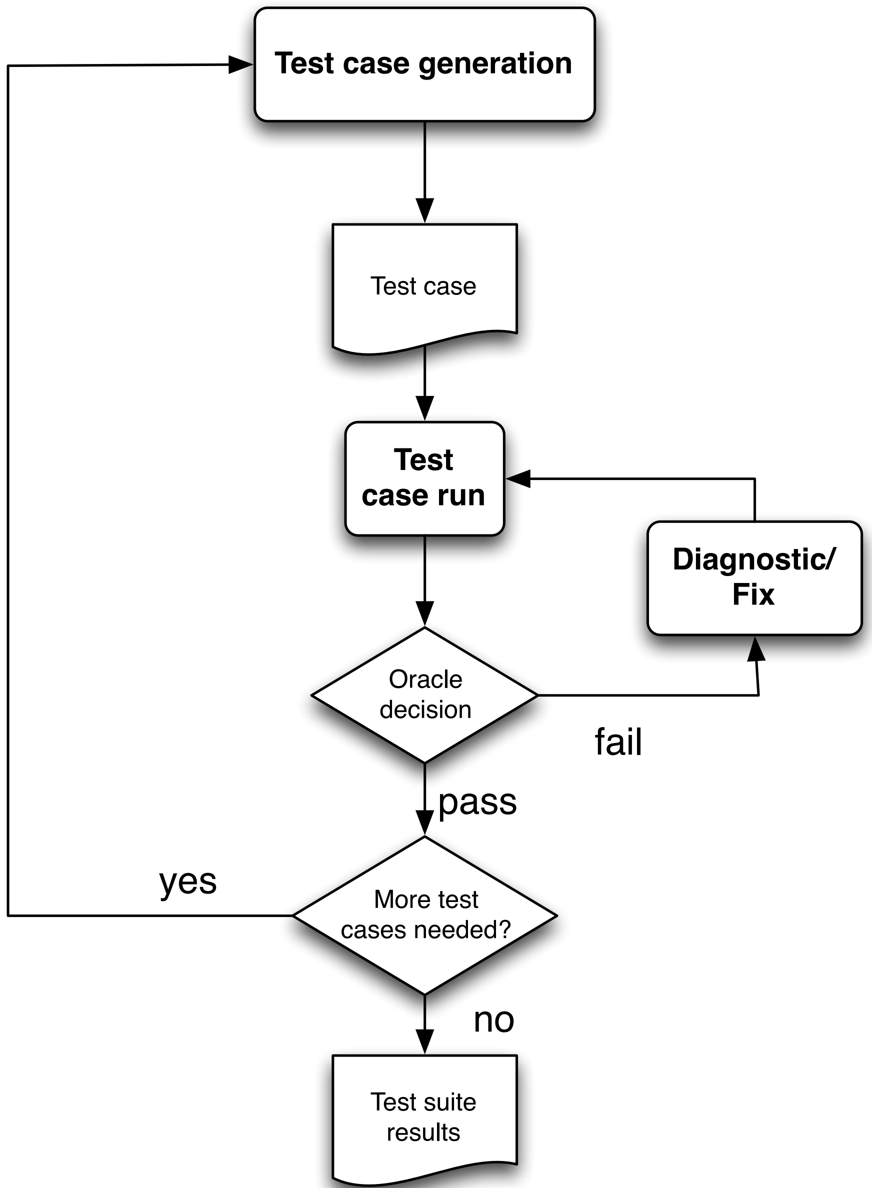
Categories of testing techniques

- Based on the internal structure of the system under test (SUT)
  - static testing, eg structural testing
- Based on executing the SUT
  - dynamic testing, eg functional testing

First phase of the testing process

- Get precise specifications of the expected behavior
  - requirements in a natural language
  - comments in the code
  - contract based specifications (pre/post)
  - formal specification (statecharts, B model, OCL)


Test case
Specification
System
Expected result
Real result
Compare (oracle)
Same?
Yes: test passes
No: test fails


Test case generation
Test case
Test
case run
Diagnostic/
Fix
Oracle
decision
fail
pass
yes
More test
cases needed?
no
Test suite
results

Different levels of testing

- **Unit** testing
  - class by class
- Integration testing
  - groups of classes working together, subsystems
- System testing
  - the whole system, involves end users (for interactive systems)

Implementation level vs test level

- In a classical V cycle (ie **waterfall**)
  - requirements matches system tests (user level)
  - coarse grain design matches integration testing
  - fine grain design matches unit testing

Testing on a large scale: the Google case

- 6000 developers with about 1500 projects
- dev cycles last one or a few weeks
- 50% of code changes in a month
- what about testing all this?

Testing on a large scale: the Google case (cont'd)

- 120.000 test suites
- giving 7.5 millions of test suites run each day
- 120 millions of unit tests
- 1400 continuous integration builds

Basic setups

- Unit testing requires test drivers (engine that loads, runs and monitors the SUT), oracle code, coverage measure
- Integration requires stubs plus unit testing setup
- System testing requires a whole setup including user interface

Static testing

- No execution required
- Inspection/evaluation of the static structure (design, code) brings insight on the quality of the implementation with respect to specifications

Code inspection

- 4 people
  - code writer
  - code inspector
  - designer
  - moderator
- goal: to find and list defects
- defects are not corrected at once

Pros and cons of code inspection

- Tedious
  - coding is more fun
  - source of tension between people
- Not automatised therefore
  - costly
  - cannot be done continuously like continuous integration

Source analysis

- Relies on simple criteria
  - methodical
  - is the code suitable for further reuse and adaptation?
  - does structure stand out clearly?
  - does it follow design diagrams
  - find out redundancies and dissymetries

Basic rules

- Header for operations
  - Purpose
  - Preconditions/postconditions
  - Parameters
  - Exceptions

Basic rules (cont'd)

- Dependency minimization
- Clear interfaces
- Usage of predocumented collaborations (design patterns)

Support

- To ease the application of rules
  - Use of an integrated development environment
  - completion, skeleton generation
  - dead code detection, uninitialised and unused variables, etc
- Use of a code analyser to evaluate compliance to the rules

Project code structuration

- Defined and follow a standard for file organisation
  - headers with adequate data: authors, date, purpose, version,…
- Maintain change logs
  - with the help of a versioning system
  - but needs proper comments when commiting

Dynamic testing

- Based on executing a system using a test suite for trying out different possibilities
- A test suite is a set of test cases
- A test case contains input data and service requests
- An oracle checks that the actual behavior is equivalent to the expected behavior

Restriction of the input domain

- A real input domain D is reaaaally huge
  - we cannot try all possible values (too long and costly)
- Therefore we take a subset T of D
  - if the system works ok on T then we **trust** it for D

Choosing a set of test cases

- Hand picking input data
- Automatic random generation
- Constrained automatic random generation
- Automatic generation based on requirements models (Model Based testing)

Partition based generation

- Principles
  - use the input domain specification to find subdomains
  - pick some values in the subdomain, which will represent all values in the subdomain (equivalence class)

Example

- If an operation specification uses a divide by N of an input parameter x
  - use values 0, 1, ...,N-1 for x
  - value v will represent all multiples of v

Limit testing

- Variant of the previous technique
- Find critical values by analyzing the specification of the operation

Example #2

- // pre : (v >=1) and (v <= 100)
- void doIt(int v)
- Use the following critical values
  - 1, 100, 2, 99, 0, 101, 50, 49, 51
- Singularities are often weak points

Example #3

- The triangle test
  - takes floats (v1, v2, v3) as input
  - return true if and only there exists a triangle with side lengths of v1, v2 and v3

Example #3 (cont'd)

- List of values
  - 1, 1, 2 : not a triangle
  - 0, 0, 0 : one point only
  - 4, 0, 3 : flat
  - 1, 2, 3.00000001 : almost a triangle
  - 0.0001, 0.0001, 0.0001 : tiny triangle
  - 888888, 888888, 888888 : huge triangle
  - 1, 1, A : undefined value
  - -1, -1, -1 : negative lengths

Mutation based test case generation

- How can we trust our test cases?
  - Are they precise enough?
  - Do we have enough test cases?
- Answering these questions is important
  - “who watches the watchers”...

Principles of mutation

- Let ST be the set of test cases for a program P to be tested
- We generate a set of “bad mutants” Mi from P
- We run the ST on each Mi
- Intuitively ST should detect the mutants

More precisely

- During mutation analysis ST is fixed
- We use each Mi in turn
- If a test case t in ST fails for Mi
  - Mi is “killed”: mutant detected
  - t has some testing quality

More precisely (cont'd)

- If a test case t in ST succeeds for Mi
  - Mi is alive, t did not detect the mutant
  - this can mean two things
    - Mi is equivalent to P (innocuous mutant), we cannot conclude on t's quality
    - Mi is not equivalent to P and t has failed to spot the "bad" mutant

Quality definition

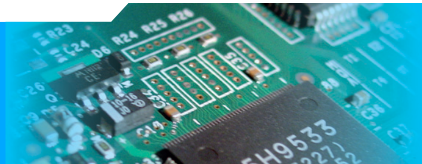
- Quality score for a program P
  - (number of mutants killed)/(number of mutants, excluding equivalent mutants)

Mutant generation

- How does one generate automatically a set of mutants Mi?
  - by randomly altering the source code, or the internal form (tree form) of a program

Examples of mutation operator

- + becomes -, and vice versa
- AND becomes OR, NOT are injected
- logical expressions are replaced by constants (TRUE, FALSE)
- comparisons are inverted, restricted or expanded (e.g. > becomes >=)



Examples of mutation operator (cont'd)

- Suppression of statements
- Anything that keeps the compiler happy to prevent broken mutants

Mutation of OO programs

- OO programs have more features than non OO ones
  - therefore specific mutation operators are needed

OO mutation operators

- Throw an exception
- Alter operation visibility (private to public)
- Alter objects through aliases
- Remove cloning instruction

OO mutation operators (cont'd)

- Add spurious calls to visible operations of neighbor objects
- Suppress call to super
  - And in general wreck havoc in the protocol with the inherited class

Structural testing

Functional vs structural testing

- Functional testing relies on outside properties that must be checked
- Structural testing relies on insider's knowledge to produce better tests

Structural tests

- Aim at integration testing
  - based on the structure of the system
- Aim at unit testing
  - to improve the quality of the test suite

Control graph

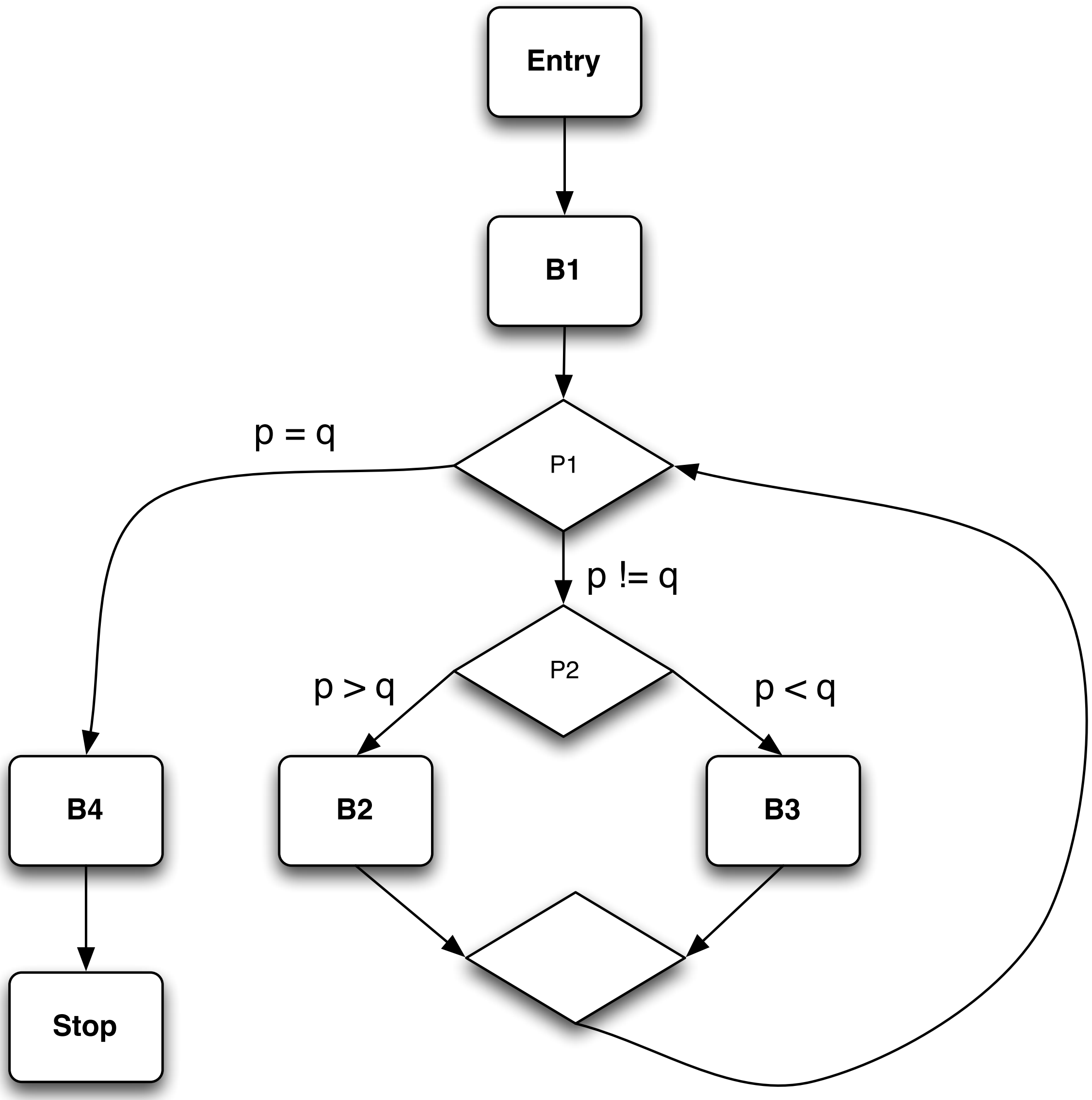
- An abstraction of the source code
- Contains control flow information
- Nodes
  - block of sequential statements
  - predicates (in control statements)
  - junctions (empty nodes)

Control graph (cont'd)

- Edges
  - connect nodes to represent the flow of control

An example

- // Compute the PGCD of p and q
- read(p,q) // Block B1
- while p<>q do // Predicate P1
  - if p>q then // Predicate P2
    - p := p - q; // Block B2
    - else q := q - p; // Block B3
  - end // if
- end // while
- result := p; // Block B4


Entry
B1
P1
p = q
p != q
P2
p > q
p < q
B4
B2
B3
Stop

Paths

- Execution path
  - a sequence of edges from E to S (often we use the sequence of nodes)
- Path predicate
  - the conjonction of control predicates that make execution follow a given path

Path examples

- E B1 P1 P2 B2 S
  - p0 > q0 and p1 = q1
- E B1 P1 P2 B3 S
  - p0 <= q0 and p1 = q1

Coverage criteria

- Are all paths are “covered” by a test suite?
  - too many paths in general
- k paths
  - includes paths that take every loop 0..k times
- elementary paths
  - at most one loop execution, for every loop

Data flow

- Control flow graphs do not take data into account
  - predicate coverage (missing cases?)
  - variable dependency (initialization?)
  - input domain coverage

What is the best technique?

- Structural and functional techniques are complementary
- For unit testing
  - start with functional testing
  - use structural techniques to improve confidence (e.g. coverage)