Part IV

Introduction to Software Testing

Hélène WAESELYNCK
Software: first failure cause of computing systems

Size: from some (tens) of thousands of code lines to some millions of code lines

Development effort:
- 0.1-0.5 person.year / KLOC (large software)
- 5-10 person.year / KLOC (critical software)

Share of the effort devoted to fault removal: 45-75%

Fault density:
- 10-200 faults / KLOC created during development
- static analysis
- proof
- model-checking
- testing
- 0.01-10 faults / KLOC residual in operation
- Issues of **controllability** and **observability**
  - **Examples:**
    - Manual computation of expected results, executable specification, back-to-back testing of different versions, output plausibility checks, ...

- **Oracle** problem = how to decide about the correctness of the observed outputs?
  - At least one test input activates the fault and creates an error
  - The error is propagated until an observable output is affected
  - The erroneous output violates an oracle check

- To reveal a fault, the following chain of conditions must be met:
If \( x > 0 \): output \( \frac{x+1}{3} + 1 \nabla \)
Else: output 0

Example_function (int x)
BEGIN
    int y, z ;
    IF x \leq 0 THEN
        z = 0
    ELSE
        y = x-1 ; /* y = x+1 */
        z = (y/3) +1 ;
    ENDIF
    Print(z) ;
END

- Activation of the fault if \( x > 0 \)
- Error propagation: incorrect output if \((x \mod 3) \neq 1\)
- Violation of an oracle check:
  - Expected result correctly determined by the operator \(\nabla\) fault revealed
  - Back-to-back testing of 2 versions, V2 does not contain this fault \(\nabla\) fault revealed
  - Plausibility check \(0 < 3z-x < 6\) \(\nabla\) fault revealed if \((x \mod 3) = 0\)
Explicit consideration of testing in the software life cycle

Example: V life cycle
• Whatever the adopted life-cycle model, it defines a testing process, in interaction with the software development process
  – Planning test phases associated with development phases
  – Progressive integration strategy (e.g., top-down design, bottom-up testing)
  – Tester/developer independence rules (according to software phase and criticality)
  – Rules guiding choice of test methods to employ (according to software phase and criticality)
  – Procedures for coordinating processes

• Documents are produced at each testing step
  – Employed test methods
  – Test sets + oracle
  – Test platform: host machine, target machine emulator, target machine, external environment simulator
  – Other tools: compiler, test tools, drivers and stubs specifically developed
  – Test reports
• **Unit testing** = testing of an isolated component

![Diagram showing unit testing](image)

- Test driver
- Test inputs
- Unit under test
- Calls
- Stub
- Outputs
- Oracle

• **Integration testing** = gradual aggregation of components

E.g.: bottom-up strategy

- Test \{C\}, test \{D\}
- Test \{B, C, D\}
- Test \{A, B, C, D\}

😊 no stub to develop

😊 High-level components are tested late, while it may be important to test them early
  - because they are major components of the system (e.g., GUI)
  - to reveal high-level faults (e.g., inadequate functional decomposition)

➡️ **Other strategies**: top-down, sandwich
Test design methods: problem

Exhaustive testing is impractical!

- Very large, or even infinite, input domain
  - Testing a simple program processing three 32 bits integers: $2^{96}$ possible inputs
  - Testing a compiler: infinite input domain

- Relevance of the very notion of exhaustiveness?
  - Elusive faults (Heisenbugs): activation conditions depend on complex combinations of internal state x external requests

Partial verification using a (small) sample of the input domain

Adequate selection?

No model of all possible software faults
Classification of test methods

<table>
<thead>
<tr>
<th>②</th>
<th>①</th>
<th>STRUCTURAL MODEL</th>
<th>FUNCTIONAL MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>②</td>
<td>SELECTIVE CHOICE</td>
<td>deterministic structural</td>
<td>deterministic functional</td>
</tr>
<tr>
<td>②</td>
<td>RANDOM CHOICE</td>
<td>statistical structural</td>
<td>statistical functional</td>
</tr>
</tbody>
</table>

① criterion
The model synthesizes information about the program to be tested. The criterion indicates how to exploit the information for selecting test data: it defines a set of model elements to be exercised during testing.

② input generation process
Deterministic: selective choice of inputs to satisfy (cover) the criterion.
Probabilistic: random generation according to a probabilistic distribution over the input domain; the distribution and number of test data are determined by the criterion.
Fault finding test: test aimed at uncovering fault

Conformance test: functional test aimed at checking whether a software complies with its specification (integrated testing level, required traceability between test data and specification)

Robustness test: test aimed at checking the ability of a software to work acceptably in the presence of faults or of stressing environmental conditions

Regression test: after software modification, test aimed at checking that the modification has no undesirable consequence
Introduction

Structural testing

Functional testing

Mutation analysis

Probabilistic generation of test inputs
Control flow graph

Oriented graph giving a compact view of the program control structure:

• built from the program source code
• A node = a maximal block of consecutive statements $i_1, \ldots, i_n$
  – $i_1$ is the unique access point to the block
  – the statements are always executed in the order $i_1, \ldots, i_n$
  – the block is exited after the execution of $i_n$
• edges between nodes = conditional or unconditional branching

Based on:
- control flow graph
- control flow graph + data flow annotations
POWER function:
computes $Z = X^Y$, where $X$ and $Y$ are two integers ($X \neq 0$)

BEGIN
  read $(X,Y)$ ;
  $W = \text{abs} (Y) ;$
  $Z = 1;$
  WHILE ($W <> 0$) DO
    $Z = Z \times X ;$
    $W = W-1 ;$
  END
  IF ($Y<0$) THEN
    $Z = 1/Z ;$
  END
  print $(Z) ;$
END

Program execution = activation of a path in the graph
Structural Criterion: guides the selection of paths
○ All Paths
  ➥ Non-executable path:
    1 → 2 → 4 → 5 → 6
  ➥ Infinite (or very large) number of paths:
    number of loop iterations 2 → (3 → 2)*
    determined by |Y|

○ All Branches
  ➥ Two executions are sufficient
    Y < 0 : 1 → 2 → (3 → 2)⁺ → 4 → 5 → 6
    Y ≥ 0 : 1 → 2 → (3 → 2)* → 4 → 6

○ All Statements
  ➥ Covered by a single execution
    Y < 0 : 1 → 2 → (3 → 2)⁺ → 4 → 5 → 6
Other criteria

• Criteria for covering loops
  – Intermediate between “all paths” and “all branches”
  – E.g., pick paths that induce 0, 1 and n>1 loop iterations (you may, or not, consider all subpaths for the loop body at each iteration)

• Criteria for covering branch predicates
  – Refinement of “all branches” (and also possibly “all paths”) when the branch predicate is a compound expression with Boolean operators.
  – Example: $A \land (B \lor C)$
    Test every possible combination of truth values for conditions $A$, $B$ et $C \rightarrow 2^3$ cases
    Test a subset of combinations such that each condition independently affects the outcome of the decision to False and True ...
    ...
... Test a subset of combinations such that each condition independently affects the outcome of the decision to F and T...

**Principle**

\[ A \land \neg A \]

2 test cases F T

\[ A \lor B \]

3 test cases FT TF TT

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FF</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>FT</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>TF</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>TT</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FF</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>FT</td>
<td>A (3)</td>
</tr>
<tr>
<td>2</td>
<td>TF</td>
<td>B (3)</td>
</tr>
<tr>
<td>3</td>
<td>TT</td>
<td>A (1), B (2)</td>
</tr>
</tbody>
</table>

\[ A_1 \land A_2 \ldots \land A_n \Rightarrow n+1 \text{ test cases} \]

\[ A_1 \lor A_2 \ldots \lor A_n \Rightarrow n+1 \text{ test cases} \]

MC/DC criterion = Modified Condition / Decision Coverage

Very much used in the avionics domain: required by DO-178B for Level A software

Test a subset of combinations such that each condition independently affects the outcome of the decision to F and T...

\[ A \land \neg A \]

3 test cases FF FT TF

\[ A \lor B \]

3 test cases FF FT TF

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FF</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>FT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>TF</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>TT</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FF</td>
<td>A (2), B (1)</td>
</tr>
<tr>
<td>1</td>
<td>FT</td>
<td>B (0)</td>
</tr>
<tr>
<td>2</td>
<td>TF</td>
<td>A (0)</td>
</tr>
<tr>
<td>3</td>
<td>TT</td>
<td>–</td>
</tr>
</tbody>
</table>

\[ A_1 \land A_2 \ldots \land A_n \Rightarrow n+1 \text{ test cases} \]

\[ A_1 \lor A_2 \ldots \lor A_n \Rightarrow n+1 \text{ test cases} \]

\[ A_1 \land A_2 \ldots \land A_n \Rightarrow n+1 \text{ test cases} \]

\[ A_1 \lor A_2 \ldots \lor A_n \Rightarrow n+1 \text{ test cases} \]
Ex: $A \land (B \lor C)$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TTT</td>
<td>T</td>
<td>A (5)</td>
</tr>
<tr>
<td>TTF</td>
<td>T</td>
<td>A (6), B (4)</td>
</tr>
<tr>
<td>TFT</td>
<td>T</td>
<td>A (7), C (4)</td>
</tr>
<tr>
<td>TFF</td>
<td>F</td>
<td>B (2), C (3)</td>
</tr>
<tr>
<td>FTT</td>
<td>F</td>
<td>A (1)</td>
</tr>
<tr>
<td>FTF</td>
<td>F</td>
<td>A (2)</td>
</tr>
<tr>
<td>FFT</td>
<td>F</td>
<td>A (3)</td>
</tr>
<tr>
<td>FFF</td>
<td>F</td>
<td>—</td>
</tr>
</tbody>
</table>

Take a pair for each operand
A : (1,5) or (2,6) or (3,7)
B : (2,4)
C : (3,4)

Hence two minimal sets for covering the criterion
{2, 3, 4, 6} or {2, 3, 4, 7}
i.e., 4 cases to test (instead of 8)

Generally: $[n+1, 2n]$ instead of $2^n$

Remark: MC/DC can be applied to instructions involving Boolean expressions, in addition to branching conditions

If $(A \land (B \lor C))$  
res := $A \land (B \lor C)$;
MC /DC and coupled conditions

« If a condition appears more than once in a decision, each occurrence is a distinct condition »

• Example: \((A \land B) \lor (A \land C)\)
  2 occurrences of A: 2 pairs for A
  Occurrence 1 -> pair (6,2)  Occurrence 2 -> pair (5,1)
  \[
  \begin{array}{c c c c}
  A & B & C & res \\
  6 & T & T & F & T \\
  2 & F & T & F & F \\
  \end{array}
  \quad
  \begin{array}{c c c c}
  A & B & C & res \\
  5 & T & F & T & T \\
  1 & F & F & T & F \\
  \end{array}
  \]

• Some cases may be impossible to cover when conditions are not independent
  Example 1: \((A \land B) \lor \neg A\)
  1st occurrence of A cannot affect the decision outcome F
  Example 2: \((A \land x \leq y) \lor (x > y - 10)\)
  \(x \leq y\) cannot affect the decision outcome F
MC / DC in practice

- A priori approach via complete truth table often infeasible (e.g., number of operands > 4)

- A posteriori evaluation of an existing (functional) test set
  - Coverage analysis tools: Rational Test RealTime, IPL Cantata, LDRA Testbed

- According to results, complement, or not, the test set
Data flow

Annotating the control graph:

• Node i
  – Def (i) = set of variables defined at node i, which can be used externally to i
  – C-use (i) = set of variables used in a calculus at node i, not defined in i

• Edge (i,j)
  – P-use (i, j) = set of variables appearing in the predicate conditioning transfer of control from i to j

For each variable v defined in node i, selection of subpaths between this definition and one or several subsequent uses of v

Associated criteria
Example: definition of Z at node 1 --> covering all uses?

- use at node 3: test with |y| > 0
- use at node 6: test with y = 0
- use at node 5 impossible, as path 1 → 2 → 4 → 5 → 6 is infeasible
criteria:

- **All definitions**
  - Selection of a subpath for each variable definition, for some use (equally in a calculus or predicate)

- **All C-uses / some P-Uses (resp. all P-uses / some C-Uses)**
  - Use in calculation (resp. in predicate) is favored

- **All Uses**
  - Selection of a subpath for each use

- **All DU paths**
  - Selection of all possible subpaths without iteration between definition and each use
Ordering of structural criteria (subsumption)

All paths

All DU paths

All uses

All C-Uses / some P-Uses

All C-Uses

All definitions

All P-Uses / some C-Uses

All P-Uses

Branches

Instructions

MC/DC
Structural Criteria – conclusion

• Criteria defined in a homogeneous framework
  – Model = control flow graph (+ possibly data flow)

• Mainly applicable to the first test phases (unit testing, integration of small sub-systems)
  – Complexity of analysis rapidly grows
  – Note: for integration testing, a more abstract graph may be used (call graph)

• Tool support (except for data-flow)
  – Automated extraction of control-flow graph, coverage analysis

• Structural coverage is required by standards
  – Typically: 100% branch coverage
  – Can be more stringent: MC/DC for DO-178B
Introduction

Structural testing

Functional testing

Mutation analysis

Probabilistic generation of test inputs
Equivalence classes + boundary values

Principle

Partition the input domain into equivalence classes to be covered
Classes determined from the functional requirements (set of values for which functional behavior is the same), and/or from the data types (e.g., for int, positive, negative)
Consider both valid and invalid classes (robustness testing)
Identify boundary values for dedicated tests
E.g., -1, 0, 1, +/- MAXINT

Example

“The price entered by the operator must be a strictly positive integer. If price ≤ 99€, then … From 100€ and above, the processing …”

Valid classes: Invalid class: Boundary values:
1 ≤ price ≤ 99 price ≤ 0 price = -1, 0, 1, 98, 99, 100, 101, +/- MAXINT
100 ≤ price + possibly: price = real number nothing entered (↲)
price = string
• Informal approach, as based on natural language specification
  – Differing partitions and limit values can be obtained from same specification analysis

• Analysis granularity?
  – Separate analysis of input variables ➞ considering all case combinations?
    For *price* input variable
    \[1 \leq \text{price} \leq 99 \quad \text{and} \quad 100 \leq \text{price}\]
    For *in_stock* input variable
    \[\text{in\_stock} = \text{TRUE} \quad \text{or} \quad \text{in\_stock} = \text{FALSE}\]
  – Class defined by composite predicate ➞ breaking up in disjoint cases?
    discount = 10% or discount = 20%
  – Classes with non-empty intersection ➞ breaking up in disjoint cases?
    \[1 \leq \text{price} \leq 99 \quad \text{and} \quad \text{prix} \times (1 - \text{discount}) \leq 50\]

😊 Rapid explosion of the number of created classes!

• Examples of selective choice strategies aimed at covering the classes
  – Selecting one test input for each class, without considering possible intersections
  – Minimizing the number of test inputs while favoring the patterns covering several valid classes; contrarily, testing separately each invalid class
  – Pairwise strategies for covering combinations of values from classes
## Decision Table

<table>
<thead>
<tr>
<th>LIST OF CONDITIONS</th>
<th>rule 1</th>
<th>rule 2</th>
<th>rule 3</th>
<th>rule 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C 1</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>C 2</td>
<td>X</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>C 3</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LIST OF ACTIONS</th>
<th>A 1</th>
<th>A 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

For a combinatorial function

- $C_i$: input conditions
  - $A_i$: disjoint actions, order in which they are executed does not matter

- completeness and consistency of the rules (exactly one rule is eligible)
  - document the impossible cases

[*] Rule coverage = 1 test case for each rule
Finite State Machine

For a sequential function

- $S$ : finite set of states
- $S_0$ : initial state
- $\Sigma$ : finite input alphabet
- $\Omega$ : finite output alphabet
- $\delta$ : transition relation, $\delta : S \times \Sigma \rightarrow S$
- $\lambda$ : output relation, $\lambda : S \times \Sigma \rightarrow \Omega$

**Graphical representation**

**Tabular representation**

<table>
<thead>
<tr>
<th></th>
<th>$\alpha$</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2/r</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>1/s</td>
<td>2/r</td>
</tr>
</tbody>
</table>

**Preliminaries:**

- Check completeness. E.g., add self-loop ‘$\beta/$’ on State 1
- Deterministic? Minimal? Strongly connected?
coverage of states, transitions, switches (pairs of transitions)

- Testing wrt original machine or completed one
  Impact on the input domain + oracle

- controllability: reach State i to test transition i->j?
  Availability of a ‘reset’ function in the test driver

Other strategies based on fault models
faults leading to output errors  

faults leading to transfer errors  

→ Testing all transitions + checking the arrival state  

• observability: observing state j?  
  Existence of a ‘status’ function?
Generally, direct observation of state $i$ is impossible
Selective choice strategy often used:

$$T_{i,j} = \text{preamble}_i \cdot e_{ij} \cdot \text{SC}_j$$

With: $T_{ij} =$ test sequence for transition $i \rightarrow j$
$preamble_i =$ input sequence starting at initial state, and arriving at $i$
$e_{ij} =$ input activating transition $i \rightarrow j$
$\text{SC}_j =$ characterising sequence for state $j$

Input $\alpha$ enabling to distinguish between ① and ②:
① $\rightarrow r$ output ② $\rightarrow s$ output

Testing transition ② $\frac{\alpha}{s}$ ① via sequence:
Reset $\cdot \alpha \cdot \alpha \cdot \alpha$
Generally, direct observation of state $j$ is impossible
Selective choice strategy often used:

$$T_{i,j} = \text{preamble}_i \cdot e_{ij} \cdot \text{SC}_j$$

With:
- $T_{ij}$ = test sequence for transition $i \rightarrow j$
- $\text{preamble}_i$ = input sequence strating at initial state, and arriving at $i$
- $e_{ij}$ = imput activating transition $i \rightarrow j$
- $\text{SC}_j$ = caracterising sequence for state $j$

- Distinguishing sequence DS (same $\forall j$)
- Sequence UIO (unique for each $j$)
- Set $W$ (set of sequences enabling to distinguish states 2 by 2)

Do not always exist

Exists for each minimal and complete MEF
Example: Cho’s W Method

General case

\[ [1 \ 2 \ \ldots \ n] \]

\[ \downarrow \]

\[ W = \{\alpha\} \]

(Here, DS)

\[ [1 \ 3 \ [2, 4 \ \ldots \ n] \]

\[ \downarrow \]

\[ W = \{\alpha, \beta \cdot \beta\} \]

\[ [1] [3] [2, \ldots] [4] [5, \ldots] \]

Test tree

Test sequences

Test tree

Test sequences
W method:

For a completely specified, minimal, deterministic FSM of n states

If implementation is also deterministic, with n states at most, the driver implements correctly the Reset,

Then the W method guarantees to reveal faults producing output errors (according to the output alphabet) and transfer errors.
Transition systems

LTS (Labelled transition system) = low level formalism for describing process systems / communicating automata
Example: specifications in LOTOS, SDL, Estelle, … can be translated in LTS

• Basic LTS: input and output events are not distinguished
  ⇒ IOLTS (Input/Output LTS) for testing

  ![Diagram](image)

  Input event: a
  Output event: x, y, z
  Internal actions: \(\tau_1, \tau_2\)

  non-determinism, quiescence

• Test approaches referring to relations between LTSs
  Chosen relation = conformance relation between spec and impl
Example: ioco relation

Implementation not in conformity if it can exhibit outputs or quiescences not present in specification

Spec

Impl1 not in conformity

Impl2 not in conformity

Impl3 in conformity
Test cases

Test case = IOLTS with specific trap states (Pass, Fail, Inconclusive) --> incorporates inputs, expected outputs, and verdict

Example:

! x ! y ! other

? a

• enables non conformity of impl1 to be uncovered
Observation of !z after ?a
• enables non conformity of impl2 to be uncovered
Observation of a quiescence after ?a !x

FAIL PASS

(Actually, the mirror image ?↔! is taken, because inputs/outputs of the system are outputs/inputs of the tester)

Test case automatically synthesized from a test purpose

Test purpose = IOLTS

Spec = IOLTS

Test case = IOLTS
Test purpose

- handwritten
- automatically generated from a specification coverage criterion
  Example: transition coverage of the SDL model

Example: purpose = produce !z

(Actually, the mirror image of this)
Functional Testing

• No homogeneous framework, methods depend on the used model
  – Equivalence classes
  – Decision tables
  – Finite state machines
  – Statecharts
  – SDL
  – Logic-based specifications (algebraic, model-oriented)
  – ...

• Model abstraction level & criterion stringency depend on the complexity (integration level) of the tested software

• Formal models used for specification and design ⇒ makes it possible to (partially) automate testing = input generation, oracle
Introduction

Structural testing

Functional testing

Mutation analysis

Probabilistic generation of test inputs
Assessing the fault-revealing power of test methods?

- **Feedback from past projects**
  - Assessment wrt real faults
  - But reduced sample of faults

- **Mutation analysis = controlled experiments**
  - Assessment wrt seeded faults
  - Large sample of faults
Mutation analysis (1)

- introduction of a simple fault (= mutation)
  “C” → “C +1”  “true” → “false”  “+” → “-”  “<” → “≤”  “low” → “mid”

- execution of each mutant with test set T  
  – killed (fault revealed by the test set)  
  – alive (fault not revealed)

- measurement = mutation score

<table>
<thead>
<tr>
<th>Number of mutants killed by T</th>
<th>Total number of mutants</th>
</tr>
</thead>
</table>

- mutations are syntactically not representative of development faults  
  but produce similar errors  
  (it is as difficult to reveal mutations as to reveal real faults)

- Tools  
  – Mothra (Georgia Inst. of Techn.): Fortran  
  – Sesame (LAAS): C, Pascal, Assembly, Lustre  
  – JavaMut (LAAS): Java  
  – and many others…
Mutation analysis (2)

😊 Comparing test methods (academic research)

😊 Identifying imperfections of a structural or functional test set
   --> complement test set with additional cases

😊 Evaluating software propensity to mask errors
   --> locally more stringent test

But …

😊 Explosion of the number of mutants

😊 Identification of equivalent mutants partly by hand
   equivalent mutant = mutant which cannot be distinguished from
   the original programme by any test input
Effectiveness of structural criteria: some experimental results

- **FCT3: 1416 mutations**
  - Mutation score
    - All uses, P-uses, branches
    - All defs

- **FCT4: 587 mutations**
  - Mutation score
    - All uses
    - P-uses, branches
    - All defs

- **All paths: score < 100%**
- **For a given criterion, strong impact of the chosen input values**
- **Criterion stringency → no guarantee**
Imperfection of test criteria

**Deterministic approach: criterion refinement**

Examples:
- instructions → branches → paths → paths + decomposition of branch conditions
- states → transitions → W method → W+1 method (number of implemented states ≤ number of specified states +1) → … → W+n method

- exhaustiveness according to fault assumptions
- explosion of test size in order to account for weaker assumptions

**Probabilistic approach**

Random, less focused, selection: exhaustiveness according to fault assumptions not searched for, reasoning in terms of failure rate according to a sampling profile.

Remark: increase of the test size in both cases
- Necessity to automate input selection and oracle procedure
Introduction

Structural testing

Functional testing

Mutation analysis

Probabilistic generation of test inputs
Probabilistic approaches

Random testing

Uniform distribution over the input domain

Easy to implement

“blind” selection, usually inefficient… usually not recommended!

Operational testing

Input distribution is representative of operational usage

Removal of faults that will have the greatest impact on reliability + software reliability assessment

inadequate for revealing, or assessing the consequences of, faults that induce small failure rates (e.g., $< 10^{-4}/h$)

Statistical testing $=$ criterion + random selection

Imperfect but relevant information

compensates imperfection of criterion
Operational testing

- Population of users ⇒ test according to several operational profiles

- Definition of operational profile(s): functional modeling
  - Identifying operating modes, functions which can be activated in a mode, input classes
  - Quantifying probabilities associated to the model according to operating data for similar systems, or to projected estimates ⇒ introducing measurement capabilities in the system (« log » files)

- Some figures (Source: AT&T)
  - Operational profile(s) definition = 1 person.month for a project involving 10 developers, 100 KLOC, 18 months
  - Definity project: duration of system test %2, maintenance costs %10
Criterion-based statistical testing

(criterion → {elements to activate})
(structural or functional)

(i) Search for an input distribution
maximizing P, prob. of least probable element
→ balance of element weight according to
selected profile

(ii) Calculation of the number of inputs
to generate

\[ N \geq \frac{\ln(1-Q_N)}{\ln(1-P)} \]
\[ Q_N = \text{test quality wrt criterion} \]
\[ Rq : Q_N = 0.999 \rightarrow 7 \text{ activations on average} \]
\[ Q_N = 0.9999 \rightarrow 9 \text{ activations on average} \]

Structural statistical testing

Probabilitic analysis of control graph and data flow

Functional statistical testing

Probabilitic analysis of behavior models
Finite state machine, decision table, Statechart
Example

Analytical techniques

\[ C = \{ \text{paths} \} \rightarrow \text{Sc} = \{ k_1, k_2 \} \]

\[ p_1 = \text{Prob}[0 \leq Y \leq 99] \]

\[ p_2 = \text{Prob}[99 < Y \leq 9999] \]

\[ p_1 = p_2 = 0.5 \]

Structural distribution / \( C \)

Empirical techniques

Successive refinements of an initial distribution

\[ Q_N = 0.999 \rightarrow N = 10 \text{ inputs to generate} \]
## Results for 4 Unit Functions and 2816 Mutations

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Live Mutants</th>
<th>Mutation Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural Deterministic</td>
<td>from 312 to 405</td>
<td>[85.6% – 88.9%]</td>
</tr>
<tr>
<td>Uniform Random</td>
<td>from 278 to 687</td>
<td>[75.6% – 90.1%]</td>
</tr>
<tr>
<td>Structural Statistical</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99.8%</td>
</tr>
</tbody>
</table>

**Typical cases for limit value testing**

- **Contribution of probabilistic approach?**
  - Compensates imperfection of structural criteria

- **Contribution of criterion to efficiency of random inputs?**
  - Adequate test profiles
Functional statistical testing: mastering complexity?

- adoption of weak criteria (e.g., states, ...) and possibly definition of several profiles
- high test quality (e.g., $q_N = 0.9999$)

Example: component from a nuclear control system

Functional modeling: hierarchy of Statecharts
⇒ Statistical testing according to two complementary profiles $N = 85 + 356 = 441$

Efficiency wrt actual faults?

« Student » version: 12 faults (A, ..., L)
« industrial » version: 1 fault Z (hardware compensation)

Efficiency wrt injected faults?

« Student » version: mutation score = 99.8 - 100%
« industrial » version: mutation score = 93 - 96.1%

Necessity of initialisation process specific test
General conclusion

• Numerous test methods
  – General approach: take a (structural or functional) model and define model elements to be covered

• Choice
  – Depends on available models and their complexity
  – Complementarity of “global” coverage and testing of specific cases (boundary values, transient modes, …)
  – Deterministic or probabilistic selection

• (Semi-)automation of testing is highly recommended
  – B. Beizer: “About 5% of the software development budget should be allocated to test tool building and/or buying”

• Beyond this introductive lecture...
  – Not covered here: specificities wrt testing OO, concurrent, distributed, real-time software systems