Correctness, partial correctness.

Syntax checking determines, whether a program is syntactically correct or not. Syntax checking is done on a whole system. Syntax checking usually is based on a grammar of the programming language.

Model-Based Testing

— tests a whole system.

Integration testing

— a single program unit is tested (function, sub-routine, method, class, etc.)

Unit testing

— tests a single test item.

Complexity

— somebody (not author!) derives test cases, defines input/soll, documents test coverage from coverage measures.

Testing of the test item:

— invent input and judge output on-the-fly (throw-away-test).

Testing of the specification:

— testing of the requirements that the program shall implement (using requirements specifications).

Choosing Test Cases

• Considering the structure of the test item (using model, source code, concepts from coverage measures).

• From the design of the program.

Some more vocabulary

• Boolean content: covered (true positive), not covered (false negative), false positive.

• Observation: Software Usually Has Many Inputs

Software examination

• Software Examination

• Choosing Test Cases

• Choosing Test Cases

• Choosing Test Cases

• Choosing Test Cases

• Choosing Test Cases
• Which property is tested?
  • function test — functionality as specified by the requirements documents,
  • installation test — is it possible to install the software with the provided documentation and tools?
  • recommissioning test — is it possible to bring the system back to operation after operation was stopped?
  • availability test — does the system run for the required amount of time without issues,
  • load and stress test — does the system behave as required under high or highest load? ... under overload?

"Hey, let's try how many game objects can be handled!" — that's an experiment, not a test.

• resource tests — response time, minimal hardware (software) requirements, etc.

• regression test — does the new version of the software behave like the old one on inputs where no behaviour change is expected?

• Some more vocabulary

Choosing Test Cases

• Generic requirements on good test cases
  • Approaches:
    • Statistical testing
    • Expected outcomes: Test Oracle
    • Habitat-based
    • Glass-Box Testing
    • Statement/Branch/term coverage
    • Conclusions from coverage measures

• When To Stop Testing?
  • Model-Based Testing
  • Testing in the Development Process
  • Formal Program Verification

• Deterministic Programs
  • Syntax, Semantics, Termination, Divergence
  • Correctness of deterministic programs
    • partial correctness, total correctness.

How to Choose Test Cases?

• A first rule-of-thumb: "Everything, which is required, must be examined/checked. Otherwise it is uncertain whether the requirements have been understood and realised."
  (Ludewig and Lichter, 2013)

  In other words:
  • Not having at least one (systematic) test case for each (required) feature is (grossly?) negligent.
  (Dt.: (grob?) fahrlässig).

  In even other words:
  • Without at least one test case for each feature, we can hardly speak of software engineering.

  • Good project management: document for each test case which feature(s) it tests.

A test case is a good test case if it discovers — with high probability — an unknown error.

An ideal test case (In, Soll) would be
  • of low redundancy, i.e. it does not test what other test cases also test.
  • error sensitive, i.e. has high probability to detect an error, (Probability should at least be greater than 0.)
  • representative, i.e. represent a whole class of inputs, (i.e., software S passes (In, Soll) if and only S behaves well for all In′ from the class)

The idea of representative:

12345678
+ 27
7 8 9 0
4 5
+ 1
2 3

• If (12345678, 27; 12345705) was representative for (0, 27; 27), (1, 27; 28), etc.
• then from a negative execution of test case (12345678, 27; 12345705)
• we could conclude that (0, 27; 27), etc. will be negative as well.

Is it / can we?
Thus: The wish for representative test cases is problematic:

• In general, we do not know which inputs lie in an equivalence class with respect to a certain error.
• Yet there is a large body of literature on how to construct representative test cases, assuming we know the equivalence classes.

Of course: *If* we *know* equivalence classes, we should exploit that knowledge to optimise the number of test cases. But it is perfectly reasonable to test representatives of equivalence classes induced by the specification, e.g.

• valid and invalid inputs (to check whether input validation works at all),
• different classes of inputs considered in the requirements, like "C50", "E1" coins in the vending machine → have at least one test case with each.

Recall: one should have at least one test case per feature.
Where Do We Get the "Soll"-Values From?

Recall: A test case is a pair \((\text{In}, \text{Soll})\) with proper expected (or "soll") values.

- In an ideal world, all "soll"-values are defined by the (formal) requirements specification and effectively pre-computable.
- In this world, the formal requirements specification may only reflectively describe acceptable results without giving a procedure to compute the results.
- There may not be a formal requirements specification, e.g.
  - "the game objects should be rendered properly",
  - "the compiler must translate the program correctly",
  - "the notification message should appear on a proper screen position",
  - "the data must be available for at least 10 days",
  - etc.

Then: need another instance to decide whether the observation is acceptable.

The testing community prefers to call any instance which decides whether results are acceptable a (test) oracle.

I'd prefer not to call automatic derivation of "soll"-values from a formal specification an "oracle"...;-) ("person or agency considered to provide wise and insightful [...]
precognition of the future, inspired by the gods." says Wikipedia)

Some more vocabulary

Choosing Test Cases

Generic requirements on good test cases

Approaches:

- Statistical testing
  - Expected outcomes: Test Oracle
    - Habitat-based
    - Glass-Box Testing
    - Statement/Branch/term coverage
  - Conclusions from coverage measures

When To Stop Testing?

Model-Based Testing

Testing in the Development Process

Formal Program Verification

Deterministic Programs

Syntax, Semantics, Termination, Divergence

Correctness of deterministic programs

partial correctness, total correctness.

Glass-Box Testing: Coverage

Choosing Test Cases Habitat-based
Coverage Example

Statements and Branches by Example
Testing in the Development Process

In other words: "for each condition (term), there is one computation path that satisfies $\phi$ necessarily true where the condition (term) evaluates to $\phi$ satisfying.

Currently, the standard moves towards accepting certain verification or coverage measures in certification. Some more vocabulary:

Coverage Measures in Certification

Conclusions from Coverage Measures

Term Coverage

Unreachable Code

Term Coverage Example
There need to be defined criteria for when to stop testing; project planning should consider these criteria (and previous experience).

Possible "testing completed" criteria:

- All (previously) specified test cases have been executed with negative result. (Special case: All test cases resulting from a certain strategy, like maximal statement coverage have been executed.)
- Testing effort time sums up to \( x \) (hours, days, weeks),
- Testing effort sums up to \( y \) (any other useful unit),
- \( n \) errors have been discovered,
- No error has been discovered during the last \( z \) hours (days, weeks) of testing,

Values for \( x \), \( y \), \( n \), \( z \) are fixed based on experience, estimation, budget, etc.

Of course: not all criteria are equally reasonable or compatible with each testing approach.

Another Criterion

Another possible "testing completed" criterion:

- The average cost per error discovery exceeds a defined threshold \( c \).

\[
\frac{\text{cost per discovered error}}{\text{number of discovered errors}} > c
\]

Value for \( c \) is again fixed based on experience, estimation, budget, etc.

Some more vocabulary

- Choosing Test Cases
- Generic requirements on good test cases
- Approaches:
  - Statistical testing
  - Expected outcomes: Test Oracle: -/
  - Habitat-based
  - Glass-Box Testing
  - Statement/Branch/term coverage
- Conclusions from coverage measures
- When To Stop Testing?
- Model-Based Testing
- Testing in the Development Process
- Formal Program Verification
  - Deterministic Programs
  - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.
Testing in the Software Development Process

Correctness.

• Total correctness, partial correctness of deterministic programs

- Deterministic Programs

Formal Program Verification

• Testing in the Development Process

Model-Based Testing

When to stop testing?

- Synonym: test harness.

IEEE 610.12 (2013), Ludewig and Lichter (2013):

Oracle expected outcomes: Test

- Generic approaches:

Choosing test cases

• Testing on good test cases

Test cases:

- Some more vocabulary

Test Conduction: Activities & Artefacts

Forward: Test Gear

Test Cases: Test Planning, Test Preparation, Test Execution, Test Evaluation, Test Analysis, Test Report

Test Conduction: Activities & Artefacts

Backward: Test Driver & Monitor

Test Gear: Test harness, test driver & monitor, test driver & monitor role, test driver role, monitor role, driver role, monitor role

Testing in the Software Development Process

• Model-Based Testing

- Some more vocabulary

Choosing test cases

• Testing on good test cases

Test cases:

- Some more vocabulary

Test Conduction: Activities & Artefacts

Forward: Test Gear

Test Cases: Test Planning, Test Preparation, Test Execution, Test Evaluation, Test Analysis, Test Report

Test Conduction: Activities & Artefacts

Backward: Test Driver & Monitor

Test Gear: Test harness, test driver & monitor, test driver & monitor role, test driver role, monitor role, driver role, monitor role

Testing in the Software Development Process

• Model-Based Testing

- Some more vocabulary

Choosing test cases

• Testing on good test cases

Test cases:
Concepts of Software Quality Assurance

Software Quality Assurance

Project Management

Organisational Software Examination

Analytic Examination by Humans

Non-mech. Inspection

Review

Manual Proof


Semi-mech. Inspection

E.g. Interactive Prover

Examination with Computer

Mechanical Static Checking

Analyse

Check Against Rules

Consistency Checks

Quantitative Examination

Dynamic Checking (Test)

Execute

Formal Verification

Prove

Constructive Software Engineering

Constructive E.g. Code Generation

(Ludewig and Lichter, 2013)

Content

Some more vocabulary

Choosing Test Cases

Generic requirements on good test cases

Approaches:

Statistical testing

Expected outcomes: Test Oracle

Habitat-based Glass-Box Testing

Statement/Branch/term coverage

Conclusions from coverage measures

When to Stop Testing?

Model-Based Testing

Testing in the Development Process

Formal Program Verification

Deterministic Programs

Syntax, Semantics, Termination, Divergence

Correctness of deterministic programs

Partial correctness, total correctness.
Deterministic Programs

Syntax:

```
S ::= skip | u = t | S_1 ; S_2 |
     if B then S_1 else S_2 fi |
     while B do S_1 do
```

where

- $u \in V$ is a variable,
- $t$ is a type-compatible expression,
- $B$ is a Boolean expression.

Tell Them What You've Told Them...

• There is a vast amount of literature on how to choose test cases. A good starting point:
  - at least one test case per feature,
  - corner-cases, extremal values,
  - error handling, etc.
• Glass-box testing considers the control flow graph, defines coverage measures.
• Other approaches:
  - statistical testing, model-based testing,
  - Define criteria for "testing done" (like coverage, or cost per error).
• Process: tester and developer should be different persons.

Formal Verification:

• There are more approaches to software quality assurance than (just) testing.
• For example, program verification.

References

