SOFTWARE TESTING - STATE OF THE ART, METHODS, AND LIMITATIONS

MONIKA HEINER
monika.heiner@b-tu.de
http://www.informatik.tu-cottbus.de

PRELIMINARIES

- natural fault rate of seasoned programmers
  -> about 1-3 % of produced program lines

- fault-avoidant software construction techniques ?
  -> built-in quality, quality by construction

- validation techniques seem to be unavoidable !
**VALIDATION versus TESTING**

- **VALIDATION**
  - any confidence-increasing method to trust in the software’s quality

- undecidability of basic questions in software validation
  - program termination
  - equivalence of programs
  - program verification
  - . . .

- validation == testing

- testing portion of total software production effort
  - standard system: ≥ 50 %
  - extreme availability demands: ≈ 80 %

---

**CORRECT SOFTWARE**

- A software product is formally correct, if the following three items correspond:
  - specification
    - the expected properties
  - software behavior
    - the observed properties
  - documentation
    - the product description for application and maintenance

- 100% totally correct software is possible !!!
  - holds by definition for the empty specification

- How to validate the correspondance ?
  - using the software itself
  - using a model of the software instead

  . . . model-based software validation
**Testing Approach**

- checking properties
  - of the real implementation of the software
  - in the real environment
  - against its specification / documentation
- by reading it
  - **STATIC TESTING (HUMAN TESTING)**
- by executing it
  - **DYNAMIC TESTING**

---

**Properties**

- properties
  - (functional) correctness
  - safety, security
  - usability, stability, ...
  - robustness, reliability, availability
  - portability, maintainability, readability
  - extendability, ...
  - performance/throughput
  - real time behavior/ deadline conformance
  - resource consumption, ...
- special properties
  - **specification, usually checked by dynamic testing**
- general properties
  - **guidelines, usually checked by static testing**
- testing (as any kind of validation)
  - can only be as good as the specifications (guidelines) do be
**Two Famous Quotes**

- **E. W. Dijkstra, 1972:**
  
  “Program testing can be used to show the presence of bugs, but never to show their absence!”

- **G. J. Myers, 1979:**
  
  “Testing means the execution of a program in order to find bugs.”

  -> *if a test run discovers unknown bugs*
  
  *then it is called successful*
  
  *else unsuccessful*

  *endif*

  -> *testing is an inherently destructive task*

  -> *most programmers are unable to test their own programs*

  -> *ask your favourite enemy to test your programs*

---

**Terminology I**

- **BUG** - deviation from expected behavior
  
  -> *fault*

  -> *error*

  -> *failure*

- **TESTING** - discover the bug

- **DEBUGGING** - fix the bug

- **testing ≠ debugging**

  -> *done at different times*

  -> *by different people*

  -> *using different techniques*
**Terminology II**

- **TEST DATA** - values for all input data
- **TEST CASE** - complete set of values for all input data + corresponding output data values
  - A good test case answers one or several questions concerning the test object.
  - Testing is a highly sophisticated task!
  - Test data may be generated, test cases not!
    - The generator would have to have the same function as the software being tested.
- **TEST SUITE** - a representative set of test cases
  - Table-like test case notation
- **TEST ORACLE** - assesses a given test case

**General Procedure**

- **test steps**
  1. derivation of test cases (from a suitable system specification)
    - The outcome is predicted and documented before the test is run!
  2. execution of these test cases
  3. assessment of the test results

- what was in the beginning?
  - Test object, i.e. software
  - Test cases

- simultaneous design of software and its test cases!
## Test Case Selection

- **exhaustive testing impossible**
  - all valid inputs -> correctness, . . .
  - maybe theoretically finite, but mostly practically infinite
  - all invalid inputs -> robustness, security, reliability, . . .
  - infinite
  - state-preserving software (operating/information systems):
    a (trans-) action depends on its predecessors
  - all sequences of (trans-) actions had to be regarded !?

- **test case design strategy**
  - finding good test suites,
  - good = sufficiently small, but high bug discover rate

## Basic Strategies

- **structure testing** (1)
  - white-box testing,
    developer testing
  - basis:
    inner structure of the test object

- **function testing** (2)
  - black-box testing,
    user testing
  - basis:
    behavior given by the specification

- **diversified testing** (3)
  - back-to-back testing, mutation testing, perturbation testing
(1) **Structure Testing**

- based on control structure model (= control flow model)

<table>
<thead>
<tr>
<th>Program Elements</th>
<th>Control Flow Graph</th>
<th>Petri Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>Nodes</td>
<td>Transitions</td>
</tr>
<tr>
<td>Control Flow</td>
<td>Arcs</td>
<td>Places</td>
</tr>
</tbody>
</table>

- control flow - based testing

- data flow - based testing (defs/uses methods)

**Test Coverage**

- relation of executed to existing statements/branches/paths . . .
- easy to compute by code instrumentation
- side-effect: hot spots are revealed -> tuning

- main drawback: specification is not checked!

(2) **Function Testing**

- considerations on the input space
  - equivalence partitioning
effective selection depends on the skills and experience of the tester
  - boundary value testing
  - special value testing

- random testing, statistical testing
  - estimation of residual defects
  - suitable combination with equivalence partitioning

- testing against some model
  - state automaton
  - cause effect graph
  - fault tree, . . .

test coverages similar to structure testing node/branch/path coverage
(3) DIVERSIFIED TESTING I

- back to back testing

```
golden prototype
program version A
result A
```

```
program version B
result B
result comparator
```

test data

greenhorn

Remark:
Usually, not applicable.

(3) DIVERSIFIED TESTING II

- mutation testing
  - make small changes (mutations) to the program
  - run the mutated program using the same test suite as the program being tested
  - the test suite is adequate, if it finds all mutations

- perturbation testing (fault injection)
  - implementing anomalies for inputs, outputs, and everything in between
  - impact of component bugs on the entire system
  - fault tolerance
**Recommended Procedure**

- **function testing**
  - code instrumentation to observe test coverage
  - design test suite using equivalence classes
  - execute test suite neglecting any reached coverage

- **mutation test**
  - test suite assessment

- **regression testing**
  - each debugging requires re-execution of complete test suite

- **structure testing**
  - evaluate reached test coverage
  - design additional test cases to increase test coverage
  - execute additional test cases
  - repeat as long as the specified degree has not been reached

- **Remark:**
  Usually, test suites growth step-wise over time by careful bookkeeping what has been tested before.

---

**Incremental Testing**

- **most programs are too complicated to understand all details at a glance**

- **white-box testing becomes more and more impractical with increasing size of the test component**

- **way out:** modular programming with sound interfaces (ADT), **BUT:** all interfaces are sources of confusion

- **consequences:** step-wise bottom up / top down testing
  - unit testing procedures, . . .
  - module testing set of procedures + interface
  - integration testing interaction of several modules
  - system testing complete software product
step-wise testing requires
- test **DRIVERS**
  simulating the calling modules
- test **STUBS**
  simulating the called modules

these test environments must be programmed and tested too,

<table>
<thead>
<tr>
<th>criteria</th>
<th>test method</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind of test execution</td>
<td>inspection of program code</td>
<td>review, walk-through, . . .</td>
</tr>
<tr>
<td></td>
<td>running of executables</td>
<td></td>
</tr>
<tr>
<td>kind of knowledge of the test object</td>
<td>structure test (white box test, developer test)</td>
<td>basis: inner structure of the test object</td>
</tr>
<tr>
<td></td>
<td>function test (black box test, user test)</td>
<td>basis: behavior given by the specification</td>
</tr>
<tr>
<td>size of the test object</td>
<td>unit testing</td>
<td>procedures, . .</td>
</tr>
<tr>
<td></td>
<td>module testing</td>
<td>set of procedures + interface</td>
</tr>
<tr>
<td></td>
<td>integration testing</td>
<td>interaction of several modules</td>
</tr>
<tr>
<td></td>
<td>system testing</td>
<td>complete software product</td>
</tr>
</tbody>
</table>
Special Challenges

- testing of alternative programming paradigms using
  - declarative programming languages
  - functional programming languages
  - object-oriented programming languages

- programs which can hardly be described by an IO function
  - GUI
  - state-preserving software
  - reactive systems’s software

- systematic testing of concurrent programs
  - is much more complicated than of sequential ones

Criteria to Finish Testing

- common
  - time is over (time-to-market pressure)
  - all test cases successful

- better (?)
  - Discover a given amount of bugs !
  - Reach a specified degree of test coverage(s) !
  - Reach a specified fault rate ! (number of found bugs per time)

Optimistic view

Pessimistic view

Realistic view (?)

Ageing model
**STATE OF THE ART**

- effective testing is still a challenge in real-life software development
- validation needs knowledgeable professionals
  - -> study / job specialization
  - -> profession of “software tester”
- testing is very time and resource consuming
  - -> ’external’ quality pressure
- There is no such thing as a fault-free program!
  - -> sufficient dependability for a given user profile
  - -> how to characterize a user profile?
- sophisticated testing is not manageable without tool support
  -> exercises

**LIMITATIONS OF TESTING**

- Testing (as any kind of validation) is no substitute for thinking!
- testing can only be as good as the specification
  - -> readable <-> unambiguous
  - -> complete <-> limited size
- (dynamic) testing needs an executable
- “Program testing can be used to show the presence of bugs, but never to show their absence!” [Dijkstra 72]
  - sophisticated static analyses (CONTEXT CHECKING)
    to prove the absence of certain types of bugs
  - correctness proofs (VERIFICATION),
    similar to the proof of a mathematical theorem

next slide
VALIDATION METHODS

- Context Checking
  - Analysis of static semantics
  - Data flow analysis
  - Control flow analysis
  - Pragmatic aspects
  - Data flow anomalies
  - Control flow anomalies

- Verification
  - Prototyping (functional simulation)
  - Symbolic execution
  - Program proving
  - Functionality
  - Robustness
  - Safety ...

- Evaluation
  - Analytical evaluation
  - Simulative evaluation
  - Performance
  - Reliability
  - Availability ...

- Testing
  - Qualitative testing
  - Quantitative testing
  - Functionality
  - Robustness
  - Performance ...

GENERAL SEMANTIC PROPERTIES

- Qualitative properties
  - Time-less properties

SPECIAL SEMANTIC PROPERTIES

- Quantitative properties
  - Time-based properties

VALIDATION PROPERTIES

- Time-free properties
- Time-based properties