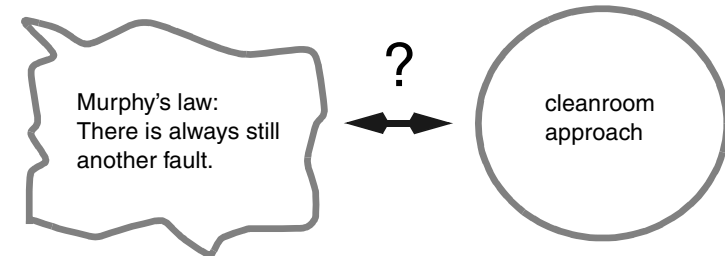


SOFTWARE TESTING - STATE OF THE ART, METHODS, AND LIMITATIONS

MONIKA HEINER

mh@informatik.tu-cottbus.de
<http://www.informatik.tu-cottbus.de>

PRELIMINARIES



- natural fault rate of experienced programmers - about 1-3 % of produced program lines
- testing consumes at least half of the labor expended to produce a working program
 - > extreme availability demands - 80% of the total effort
- G. J. Myers:
"Testing means the execution of a program in order to find bugs."
 - > A test run is called successful, if it discovers unknown bugs, else unsuccessful.
- testing is an inherently destructive task,
 - > most programmers are unable to test their own programs

SOFTWARE TESTING

- ☐ checking *properties* of the real *implementation* of the software against its *specification*
 - by reading it -> **STATIC TESTING**
 - by executing it -> **DYNAMIC TESTING**
- ☐ properties
 - (functional) correctness
 - security
 - reliability
 - usability/robustness
 - performance
 - portability, maintainability
 - ...
- ☐ a software product is correct formally, if the followings correspond:
 - specification (i.e. the expected properties),
 - software behavior (i.e. the observed properties),
 - documentation (i.e. the product description for application and maintenance).

TERMINOLOGY

- ☐ **BUG**: derivation from expected behavior (fault - error - failure)
- ☐ **TESTING** - discover the bug
DEBUGGING - fix the bug
- ☐ **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,
 - > test case table
- ☐ test steps
 - derivation of test cases (from a suitable system specification)
The outcome is predicted and documented before the test is run !
 - execution of these test cases
 - assessment of the test results

TEST CASE SELECTION

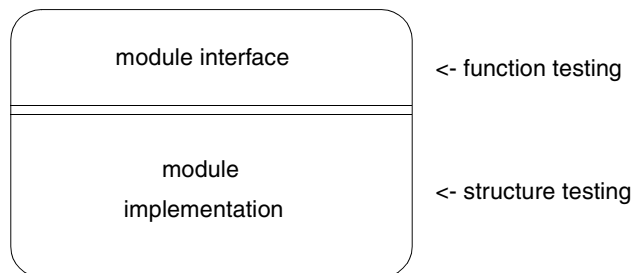
- exhaustive testing impossible
 - all valid inputs -> correctness, . . .
(maybe theoretically finite, but mostly practically infinite)
 - all invalid inputs -> robustness, security, reliability, . . .
(infinite)
 - state-preserving software (operating/information systems):
a (trans-) action depends 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
 - **(1) STRUCTURE TESTING** (white-box testing, developer testing)
basis: inner structure of the test object,
 - **(2) FUNCTION TESTING** (black-box testing, user testing)
basis: behavior given by the specification



- **(3) DIVERSIFIED TESTING**

(1) STRUCTURE TESTING

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

	control flow graph	Petri net
statements:	nodes	transitions
control flow:	branches	places

- control flow - based testing
 - statement coverage testing (C0)
 - branch coverage testing (C1)
 - path coverage testing (C2)
(complete, structured, boundary interior)
 - condition coverage test (C3)
(simple, minimal multiple, multiple)

Remark: C0 < C1 < C2; C1 < C3 / (min) multiple

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

def	- assignment of a value
computational use (c-use)	- use to compute expression
predicate use (p-use)	- use to evaluate condition

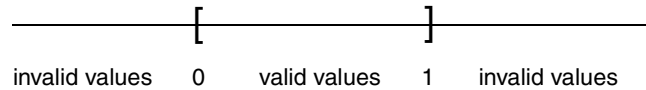
 - all defs criterion
 - all p-uses, all c-uses, all uses criterion
 - all c-uses/some p-uses criterion
 - all p-uses/some c-uses criterion

- 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

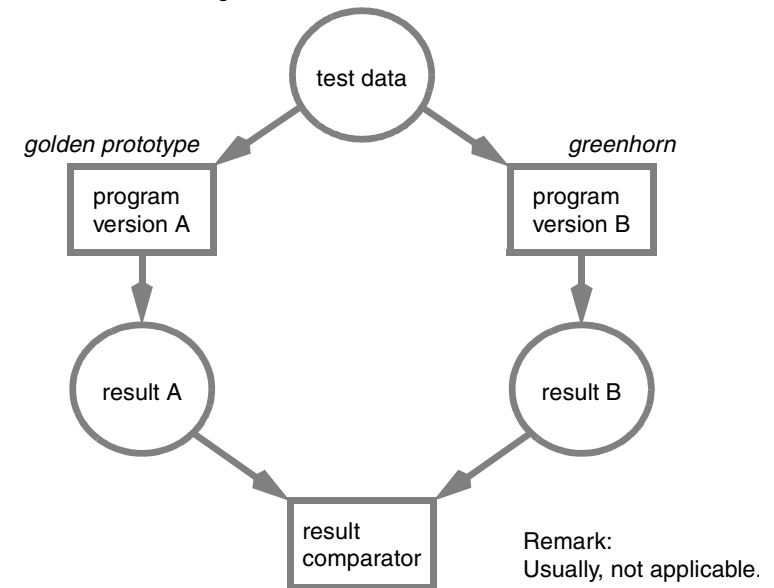
- equivalence partitioning
 - break the (infinite) data space into a finite set of equivalence classes of input data with common properties
 - assumption: testing with one randomly chosen value of each class is equal to testing with any other value of the same class



- boundary value testing
- special value testing
 - effective selection depends on the skill and experience of the tester
- random testing, statistical testing
 - estimation of residual defects
 - suitable combination with equivalence partitioning
- testing of state automaton
 - specification is given as state automaton
 - test coverages similar to structure testing: node / branch / path coverage
- cause effect analysis, fault tree analysis

(3) DIVERSIFIED TESTING

- back to back testing



- 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

- 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

- mutation test
 - test suite assessment

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

-> SUPPORT BY SUITABLE TEST TOOLS !!

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

INCREMENTAL TESTING

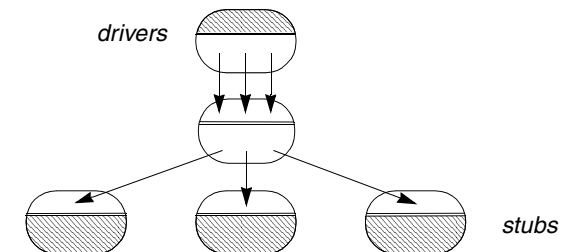
- most programs are too complicated to understand in detail

- 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

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

- step-wise testing requires
 - test **DRIVERS** simulating the calling modules
 - test **STUBS** simulating the called modules



- these test environments must be programmed and tested again,
...
...
... ..

CLASSIFICATION OF TEST METHODS, SUMMARY

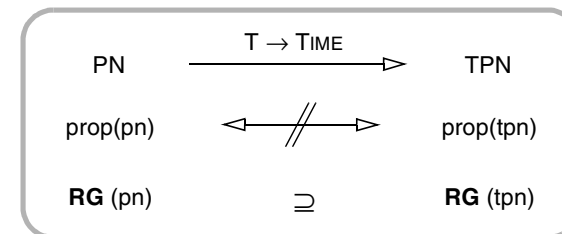
criteria	test method	remarks
kind of test execution	inspection of program code	review, walk-through, . . .
	running of executables	
kind of knowledge of the test object	structure test (white box test, developer test)	basis: inner structure of the test object
	function test (black box test, user test)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

TESTING OF CONCURRENT SOFTWARE

- state space explosion, worst-case: product of the sequential state spaces

PROBE EFFECT

- system exhibits in test mode other (less) behavior than in standard mode, -> test means (debugger) affect timing behavior

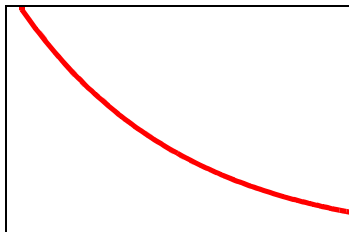


- result: masking of certain types of bugs:
 - DSt (pn) -> not DSt (tpn)
 - live(pn) -> not live (tpn)
 - not BND (pn) -> BND (tpn)

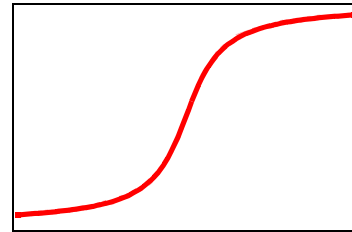
- non-deterministic behavior, pn: time-dependent dynamic conflicts
- dedicated testing techniques to guarantee reproducibility, e. g. Instant Replay [LeBlanc 87]
 - record phase - writing history tapes
 - replay phase - reading the history tapes
- combination with reachability graph/concurrent automaton: -> test coverages similar to structure testing: node / branch / path coverage

CRITERIA TO FINISH TESTING

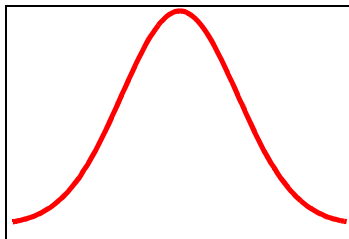
- common
 - time is over (time-to-market pressure)
 - all test cases successful
- better (?)
 - Discover a given amount of bugs !
 - Reach a specified degree of test coverage(s) !
 - Reach a specified fault rate !
(number of found bugs per time)



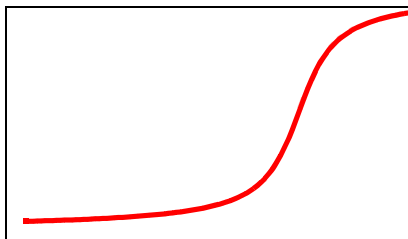
optimistic view



pessimistic view



realistic view (?)



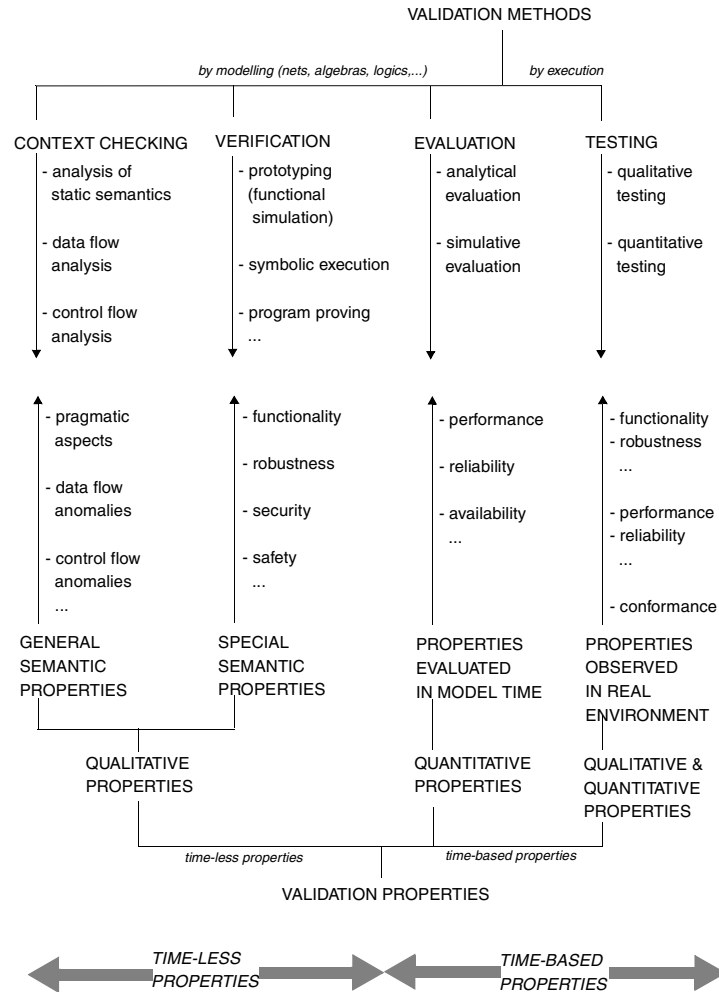
ageing model

LIMITATIONS OF TESTING

- Effective testing is still a challenge in real-life software development.
- Testing is very time and resource consuming.
- Sophisticated testing is not manageable without tool support.
- Systematic testing of concurrent programs is much more complicated than of sequential ones.
- "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
- Testing (as any kind of validation) can only be as good as the specification does be.
- Testing (as any kind of validation) is no substitute for thinking !
- THERE IS NO SUCH THING AS A FAULT-FREE PROGRAM !**

} next slide

COMPUTER-AIDED SOFTWARE VALIDATION TECHNIQUES



REFERENCES

[Balzert 98]
Balzert, H.:
Lehrbuch der Software-Technik, Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung;
Spektrum Akademischer Verlag GmbH, Heidelberg - Berlin, 1998.

[Beizer 90]
Beizer, B.:
Software Testing Techniques;
Int. Thomson Computer Press, 1990.

[Beizer 95]
Beizer, B.:
Black-Box Testing;
John Wiley & Sons, Inc., 1995.

[Dijkstra 72]
Dijkstra, E. W.:
Notes on Structured Programming;
Academic Press, London, 1972, pp. 1-82.

[Heiner 95]
Heiner, M.:
Petri Net Based Software Dependability Engineering;
Tutorial Notes, Int. Symposium on Software Reliability Engineering (ISSRE '95), Toulouse, Oct. 1995.

[LeBlanc 87]
Leblanc, T.; Mellor-Crummey, J. M.:
Debugging Parallel Programs with Instant Replay;
IEEE Trans. on Computers 36(87)3, pp. 471-482.

[McDermid 91]
McDermid, J.:
Software Engineer's Reference Book;
Butterworth-Heinemann Ltd., 1991.

[Myers 79]
Myers, G. J.:
The Art of Software Testing;
John Wiley & Sons, New York, 1979.

[Siegel 96]
Siegel, S.:
Object-oriented Software Testing;
John Wiley & Sons 1996.

[Sommerville 92]
Sommerville, I.:
Software Engineering;
Addison-Wesley Publishers Ltd., 1992.

[Voas 98]
Voas, J. M.; McGraw, G.:
Software Fault Injection, Inoculating Programs Against Errors;
John Wiley & Sons 1998.