Software Testing Research: Achievements, Challenges, Dreams

Antonia Bertolino

– paper at ICSE 2007, FSE track –

presented by Marius Minea

December 2, 2009

Starting point

Testing today is still:

- ad hoc
- expensive
- unpredictable in effectiveness

Roadmap

- where to go: *dreams*
- how to get closer: *challenges*

Recap: what is testing ?

observing software *execution*
to *validate* intended behavior
and *identify errors*

Testing is an instrument for *quality assurance*:

- most *direct* and *realistic* feedback
- ⇒ will always be needed

What are the hard problems?

Software is increasingly *complex, pervasive, critical*

We want higher *quality* and *dependability*

⇒ Testing becomes *difficult* and *expensive*

The Questions (1 - 2)

WHY: *test objective*

look for faults ?
decide on product release ?
evaluate usability ?

The Questions (1 - 2)

WHY: *test objective*

look for faults ?
decide on product release ?
evaluate usability ?

HOW: *test selection*

ad hoc
at random
systematically (algorithmic or statistical)

Important: influences test *efficacy*

The Questions (3 - 4)

HOW MUCH: *test adequacy*

coverage analysis
reliability measures

The Questions (3 - 4)

HOW MUCH: *test adequacy*

coverage analysis
reliability measures

WHAT: *levels of testing*

test whole system
or a part (unit/component/subsystem; integration)

The Questions (5 - 6)

WHERE do we observe ?

in a *simulated* environment

in the *real* (target) context

Example: embedded systems (test with emulators / with hardware)

The Questions (5 - 6)

WHERE do we observe ?

in a *simulated* environment

in the *real* (target) context

Example: embedded systems (test with emulators / with hardware)

WHEN it in the product lifecycle ?

earliest is cheapest

but some tests must await *deployment*

The four dreams

1. Universal test theory

2. Test-based modeling

3. 100% automatic testing

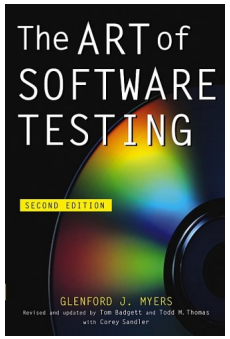
4. Efficacy-maximized test engineering

Achievements: early years

70's: testing is *an art*

destructive (execute with intent to find errors)

design is *constructive*



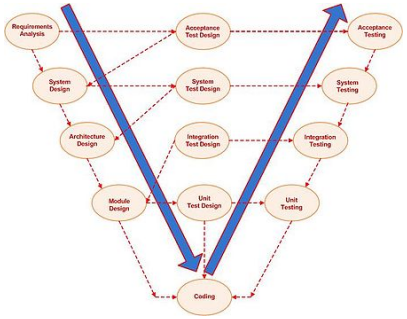
80's: testing is *an engineered discipline*

positive view: prevention

a broad and continuous activity *throughout the development* process [Hetzel]

the act of *designing* tests is one of the best bug preventers known [Beizer]

Achievements: Testing process [what? when?]



process models

systematize "test design thinking"

levels: unit, integration, system testing
⇒ *V model* ubiquitous (is it?)

Critics: excess of form = bureaucracy

⇒ *agile testing*
⇒ *test-driven development*

What is a *suitable testing process* ?
⇒ still a fundamental *research topic*

Achievements: Test criteria [how? how much?]

Well studied and classified (black-box/white-box, coverage, etc.)
Challenge: *make a* justified *choice* / combination
Buzzword: *model-based* testing ?

Comparing test criteria

from *analytical* towards *empirical*
systematic vs. *random* testing

Demonstrating *effectiveness* of testing techniques
⇒ still a fundamental *challenge*

Object-oriented testing

90's: everything is object-oriented!
but: new *problems* and *risks* for testing

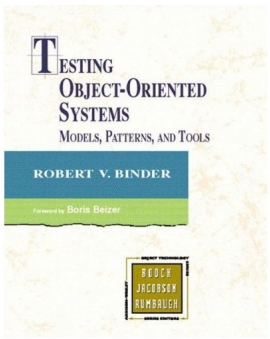
encapsulation can *hide bugs*

need to *re-test inherited* code

need new *coverage* models

for *polymorphism* and dynamic binding

need effective *incremental integration* testing



1248 pages!

Component-based testing

late 90's: components are the ultimate!

but: interface info not enough for functional testing
need to *re-test in deployed context* (just like O-O!)

Solutions:
built-in testing (tests packaged with component)
explicit *contracts* for verification

Compositional testing still a fundamental theoretical challenge

Protocol testing

Protocols are *precisely specified* (good for testing!)
may have *standard conformance test suite*
or formal methods to check conformance
⇒ *different* from general software testing

Trends:
software could adopt *standardized specifications*
protocols are increasingly *complex*
} ⇒ getting closer

Reliability testing [how? how much?]

Testing can't find all bugs
⇒ eliminate bugs hit *most often* and/or *most critical* ones

How ?
Intuitively: tester mimics user
Practically: use *reliability models*

Status:
Theory is good (software reliability models)
Practice not so good (perceived as expensive, hard to identify usage profile)

⇒ still a challenge ...

Dream 1: Universal test theory

A *theory*, what for ?
understand strengths and limitations of test *techniques*
choose the most adequate one

Dream (as concept):
despite *negative* theory (testing can never be exact)
be *positive* in practice
what do we know after applying a given testing technique ?
how can we dynamically tune our testing stragegy ?

Dream (as tool):
given *goal* of maximum test effectiveness (cf. dream 4)
decide combination of *techniques to adopt* (+ assumptions)

Challenge: Explicit test hypotheses [why?]

Guarantees are never absolute (in testing and real life...)
only under certain conditions (read the fine print)
⇒ to be sure what we're saying, need to *make assumptions explicit*

Assumptions:

execution context (closed world assumption)
software is "correct". But are the libraries ? the OS ? the hardware ?

is test representative for a class ?
e.g. *uniformity hypothesis* within a black-box partition

Similar to *fault model* in protocol testing / fault-tolerance

Summary: explicitating assumptions refines **WHY** a test is relevant

Challenge: Test effectiveness [why? how? how much?]

Old challenges, still current:

How *effective* is a test selection criterion for finding faults ?
evidence: analytical, statistical, empirical

For what classes of faults is a criterion useful ?

Wisdom: use *combination* of techniques
saturation effect affects single technique

e.g. *systematic vs. random* testing

Challenge: Compositional testing [what?]

Divide et impera also in testing
do unit testing first
Integration order / test order: extensively investigated

Compositional: what do component tests say about system behavior ?
what can we *reuse* ?
what tests must we *add* ?

Current work:

component-based software reliability (quantitative estimation)
assume-guarantee reasoning
composition of protocols
fault models for component integration

Empirical body of evidence

Controlled experimentation is an indispensable research methodology

but is hard to do

Need meaningful experiments, in terms of
scale (large)
subjects used (replicated)
context (real-world)

Solution: collaborative open experiments
data repositories: bugs, flawed versions, tests

Software-artifact Infrastructure Repository `http://sir.unl.edu/`

Cooperative Bug Isolation Project `http://www.cs.wisc.edu/cbi/`

Dream 2: Test-based modeling

Buzzword: *model-based testing*
use whatever design model and adapt a testing technique to it

Reverse is better: *test-based modeling*
build models that allow effective testing
similar to *design for testability*

Example: models already *instrumented for testing*
assertions: check internal state at runtime
contracts: can be used for test generation (JMLUnit)

Challenge: Model-based testing [how? how much?]

MBT: an old idea (since Moore and FSMs)

Promised benefits are high, but adoption barrier still there:
lack of formal modeling skills

Insight, the hard way: forcing users to new notations does not work
⇒ *combine* different models
transition-based, contract-based, scenario-based

Combine with other approaches (testing over simulations)

Integrate in current software processes

Special case: *conformance-based testing* (model to specification)
again, mostly theory, not practice ...

Challenge: Anti-model-based testing [how? how much?]

Reverse: *derive* models from test executions (chosen or passively recorded)
useful when models unavailable

Models can be:
finite-state machine
data invariants / pre- / postconditions
sequence diagrams

Approaches:
dynamic invariant generation
learning automata / extracting interfaces (for components/services)

Models need to be refined if proven inaccurate (counterexample-based)

Challenge: Test oracles [why?]

Basics: can't do testing without knowing good output

Status:
few alternatives to human observation
can become bottleneck to test automation
balance cost and effectiveness (missed errors / false positives)

Dream 3: 100% automatic testing

Utopia: a program that automatically tests itself
(generates instrumentation and test cases, runs them, removes test code, produces reports)

Promising: automated *intelligent input generation* (for unit tests)

Directed Automated Random Testing
(statically extract interface, produce test driver,
generate random inputs, exercise alternating paths)

software agitation (commercial, similar)

parameterized unit tests (Microsoft PEX for .NET)
tests characterized by symbolic constraints between inputs

Challenge: Test input generation

Again, lots of theory, limited industrial impact. But good hope.

model-based test generation

combine state-based approaches with data models

symbolic execution: exploit advances in theorem proving/constraint solving

random test generation

sophisticated techniques may outperform systematic test generation
promising: *concolic* (concrete + symbolic) execution

search-based test generation

direct search towards most promising areas of input space

Challenge: Domain-specific test approaches [what?]

So far, case studies rather than methodologies:

- databases
- GUI usability
- web applications
- avionics
- telecom systems

Current work: frameworks for generating domain-specific test drivers

Challenge: On-line testing [where? when?]

Testing not just pre-release, but also in operation
$\Rightarrow$ monitor system behavior, using dynamic analysis and self-test

Goals: detect malfunctions, performance problems; possibly recover
Called *passive testing*: needs no test suite
Less powerful than proactive testing

Related:
actively stimulate application after deployment
testing for resource-constrained applications (e.g. adaptive code unloading)

Dream 4: Efficacy-maximized test engineering

Ultimate goal: *"practical testing methods, tools and processes for development of high quality software"*

good process
+ powerful methods (*efficiency* and *effectiveness*)
+ easy-to-use tools / environments

Main obstacle: *complexity*

Main strategy: *design for testability*

Challenge: Controlling evolution

Fact: Most testing is *regression testing*

To reduce cost, need to
- reduce amount of retesting
- prioritize test cases
- automate *re*-execution

Issues:
testing global properties as parts are modified
testing component evolution in line with architecture
testing product families

Related: *test factoring*
- split complex system test into many unit tests

Challenge: Testing patterns

Recall: theory goal of relating faults to appropriate tests
$\Rightarrow$ need to collect pragmatic evidence for test effectiveness

$\Rightarrow$ organize proven solutions into *catalogue of test patterns*

Patterns document problem-solving expertise

e.g. patterns for testing object-oriented software [Binder]

Challenge: Understanding the costs of testing

Everyone keeps citing that testing cost is 50% of total software cost ...

Problem: real data is often company-confidential

Problem: most statistics assume all bug costs are equal

value-based software engineering [Boehm]
- quantitative frameworks to support manager decisions

Key question: how to use a fixed testing budget most effectively ?

Challenge: Education of software testers

... this lecture ...

Transversal challenges

Testing within the emerging development paradigm Current paradigm: from components to services

Testing process different for service developer, provider, and integrator
- black-box for the latter two
- off-line or on-line

Services have to *expose an interface* (e.g. WSDL)
⇒ can use it for testing
⇒ but method signatures only are poor information

Coherent testing of functional and extrafunctional properties

timing, performance, resource usage, workload ...

approaches: model-based and genetic
extend conformance and interface theory

Conclusions (personal)

Programs will always have bugs.

The obvious ones are found, the hidden ones remain.

Software testing has *many challenges*.

The easy ones are solved, the hard ones remain.

⇒ lots of *interesting research* to do !

Exploit connections to other areas (debugging, static analysis, modeling, constraint solving, formal methods ...)