

It's Too Darn Big:

Test Techniques for Gigantic Systems

WSA 2005



Keith Stobie

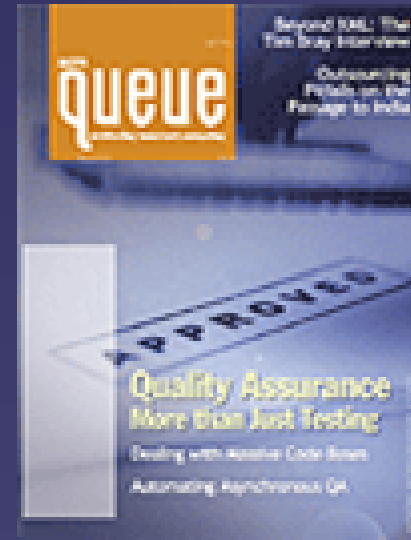
Microsoft

Test Architect, XWS

Windows Communication Foundation
(WCF, previously code name "Indigo")

Too Darn Big to Test

Testing large systems is a daunting task, but there are steps we can take to ease the pain.



- increasing size and complexity of software
- concurrency and distributed systems,
→ ineffectiveness of using only handcrafted tests

We must start again

Re-engineer Your Testing

- Begin with good design
(Employ dependency analysis for test design)
- Good static checking
(including model property static checking)
- Robust unit testing (including good input selection)
- Use “all pairs” for configuration testing
- Employ coverage to select and prioritize tests
- Use random testing including fuzz testing security
- Make models of systems
 - act as test oracles for automation.

Objectives

After attending this presentation, you will:

- Use test checklists
to help you and your developer's testing efforts
- Choose test cases which target a change
based on dependency analysis
- Test with less effort
by making appropriate use of code coverage
- Automate race condition detection

Dependency Analysis

- *Choose test cases which target a change.*
- Simple analysis: **structural** code **dependencies**
Who calls who , Fan-in and Fan-out
can be very effective
- More sophisticated: **data flow** analysis
more accurate dependency determination
but greater cost
- big wins over just running everything all the time.
- **test design implication**
→ *create relatively small test cases*
 - reduce extraneous testing
 - factor big tests into little ones.

Robust Unit Testing

- Get high quality units before integration!
key to make big system testing tractable.
- Test Driven Development
- 100% statement coverage [as per IEEE 1008-**1987**].
- Training & Generic checklists [see references]
- Tools
 - Xunit family (JUnit, NUnit, ...) or **Team System**
 - FIT : Framework for Integrated Test (Ward Cunningham)
 - IDEs : e.g. Eclipse, Rational, and **Team System** generate unit test outlines.

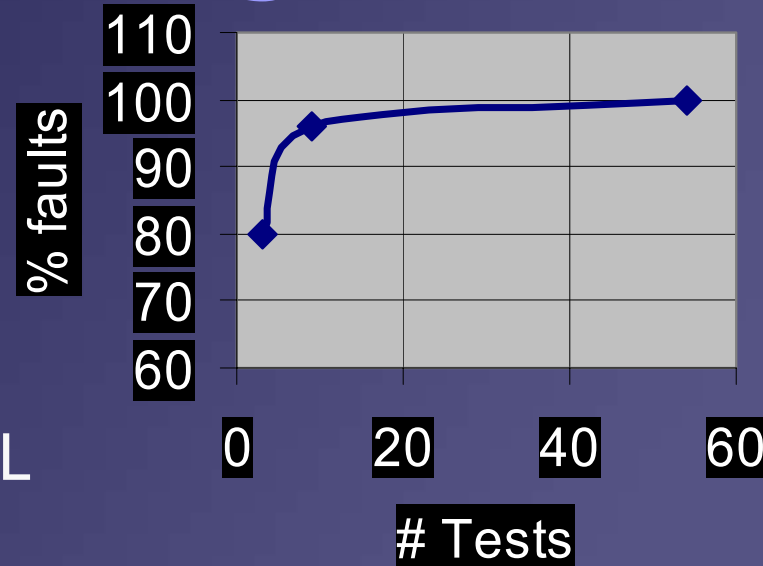
Automated Unit Testing

- Current automatic unit generation test techniques:
 - Simple known failure modes (e.g. null parameters as inputs)
 - analysis of the data structures used.
Data structure generation can be based on Constraint Solvers, Models, and symbolic execution
- You may see *100% code coverage, with no results checking!*
Be aware of results checking thoroughness; not just code coverage.
- Eclat creates approximate test oracle from a test suite, then uses it to aid in generating test inputs likely to reveal bugs or expose new behavior while ignoring those that are invalid inputs or that exercise already-tested functionality

“all pairs” for configuration testing

Example:

- **CPU:** x86, x86-64
- **Browser:** IE 5.5, IE 6.0, Firefox
- **OS:** WinXP, Win2000, Win2003
- **Security:** NTLM, Kerberos, SAML



Single value fault : 80% of time (3 tests for above)

Pair of values fault: 80% of the remainder
(96% with 9 tests (3x3) – next slide)

All combinations : example below $2 \times 3 \times 3 \times 3 = 54$ tests

All Pairs test cases example

<u>CPU</u>	<u>Browser</u>	<u>OS</u>	<u>Security</u>
x86-64	IE 5.5	Win2003	Kerberos
x86-64	IE 6.0	Win2000	NTLM
x86-64	Firefox	WinXP	SAML
x86	IE 5.5	WinXP	NTLM
x86	Firefox	Win2003	NTLM
x86	IE 6.0	Win2003	SAML
x86	Firefox	Win2000	Kerberos
x86	IE 6.0	WinXP	Kerberos
x86-64	IE 5.5	Win2000	SAML

Model Property Static Checking

Convergence of

- static analysis tools (e.g. Lint, Compiler Warnings, etc.)
- formal methods (e.g. Zed, ASML, etc.)

Example: PreFix -- simulation tool for C/C++
simulates program execution state along a selected set of program paths, and queries the execution state in order to identify programming errors

Static checking w/ Annotations

- Add annotations to describe the intent.
- Emulate checks using compiler or language constructs like assert().
- Eiffel : Precondition, Postcondition
- Sal assertions. Spec# <http://research.microsoft.com/specsharp/>
- Nasty issues that would take days or weeks to debug found effortlessly upfront
- Help independent testers

Defects found by static analysis are early indicators of pre-release system defect density.

Static analysis defect density can be used to discriminate between components of high and low quality (fault prone and not fault prone components).

Coverage: the Bad assumptions

Logical Fallacies

Not low coverage, therefore not poor tests

High coverage, therefore good tests

Coverage: the Ugly reality

- Coverage \neq Verification
- Test \rightarrow Verification
- results of real MS software study compared code coverage and defect density (defects/kilo-lines of code)
 - \rightarrow using *coverage* measures *alone* is *not accurate as predictor* of defect density (software quality/reliability).
- Uncovered areas might not be your highest priority to add new tests for!

Coverage: Good select and prioritize tests

- If you know the coverage of each test case:
prioritize the tests
to run in the *least amount of time*
to get the *highest coverage*.
 - Frequently running the minimal set of test cases
(that gave the same coverage as all of the test cases)
Finds virtually all of the bugs
(that running all of the test cases would have)
 - major problems
 - getting the coverage data (per test case)
 - keeping it up to date
 - Merge coverage data from different builds helps
- » Time to calculate minimization can be a deterrent.

Coverage Minimization Example

- **Original** Tests: 134
Calls: 2,942 Blocks: 20,929 Segments: 29,003

- **Same Call coverage** Tests: 12 (**9%**)
Calls: 2,942 (100%)
Blocks: 20,774 (**99.26%**)
Segments: 28,785 (99.25%)

- **Same Block coverage** Tests: 41 (**30.6%**)
Calls: 2,942 (100%)
Blocks: 20,929 (100%)
Segments: 28,999 (**99.99%**)

Coverage Prioritization

- Industrial experience at several companies appears to confirm early academic studies:
Dramatic reduction in # of tests while very little reduction in defect finding ability.
- You can easily run the experimental comparison.
 1. Run minimal set of tests providing same coverage
 2. Run remaining tests to see how many additional defects are revealed.
- Combine with dependency analysis!
Target only the changed code (structural or data flow)
Find fewest tests to coverage the changed code.

Random Testing

- Partitioning based on structural code coverage isn't necessarily better than random. Faults may be data or state based and exist in covered code that random testing might reveal, where further coverage might not.
 - Recent academic study^[1.] (with fault seeding): [under some circumstance] the **all-pairs** testing technique applied to function parameters was **no better than random testing** at fault detection ability.
1. "Pairwise Testing: A Best Practice That Isn't", J. Bach and P. Schroeder, 22nd Annual Pacific Northwest Software Quality Conference, Portland, OR. <http://www.pnsgc.org/proceedings/pnsgc2004.pdf>

Myth: Random testing is ineffective

- Handcrafted partitioned tests must have *reliable* knowledge about *increased fault likelihood to do better* than random testing
Think Stock Indexing vs Market pickers/timers
- Testing discipline is to understand where we have reliable knowledge of increased fault likelihood
- e.g. classic boundary value analysis technique succeeds because off-by-one errors increase the fault likelihood at boundaries

Random Testing Issues

- Verification of the result.
Easier to have static tests with pre-computed outputs.
- Very low probability sub-domains
 - likely to be disregarded by random testing
but cost of failures in those sub-domains may be very high
 - good reason for using it as a complementary rather than the sole testing strategy.

Random (Fuzz) Testing Security

Goal: push invalid data **as far** into system as possible. [3]

- Unexpected input : length and content
- Both Dumb and Smart fuzzing valuable!
 - Dumb generates data with no regard to the format
 - Smart requires knowledge of data format or how data is consumed. The more detailed the knowledge – the better.
- Generation technique creates new data from scratch
- Mutation technique transforms sample input data to create new input data : add/change/delete data
- Most fuzzing tools are a mix of each approach

After failure found, how to report?

Use Delta-Debugging [4] to reduce to minimal input.

Models of Systems

- Models encompasses the behavior from invocation of the software to termination.
- “dumb” or “smart” Monkeys
- Finite State Machines (FSM) e.g. Goldilocks
UML2
Abstract State Machines e.g. ASML
- Models can be used to generate all relevant variations for limited sizes of data structures
- finite-state, discrete parameter Markov chain
[FSM with probabilities] provide stochastic model.

Issues with Models

- Models describe at a very **high level of abstraction**. Many testers, especially adhoc ones, have difficulty with describing abstractions succinctly.
- Many traditional Test Management progress metrics are based on **number of tests cases** and Models make this metric almost irrelevant.
- Complex models without tools.
While even simple models and manual models can prove very useful, skilled modelers quickly yearn to create complex models which are only manageable with tools.

Managing Model Size

- **Exclusion**
e.g. You may chose to not model invalid inputs
- **Equivalence**
e.g. You may chose to treat different objects with the same values (although created differently) as the same object.
- **Partitioning** Variable Input (Equivalence Classes)
e.g. Select values of strings, rather than all possible strings.
- **Abstraction**
e.g. You may model the internal state as empty, partial, and full rather than all aspects of “partial”.

Models as test Oracles

- Simple models generate inputs.
- Good models also predict the expected output.
e.g. FSMs predict which state you end up in after input from current state.
ASMs allow non-deterministic behavior which is critical for real distributed systems where ordering isn't predictable.

Test checklists for large, complex, distributed systems

- Busy, Timing, multiple CPUs, etc.
System Test Pattern Language [2]
- Always assume failures
Architecture Achilles Heel Analysis [1]
- Latency (see concurrency next slides)
- Stress Attack Points and Stress Threat Analysis

Identify : Stress Attack Points

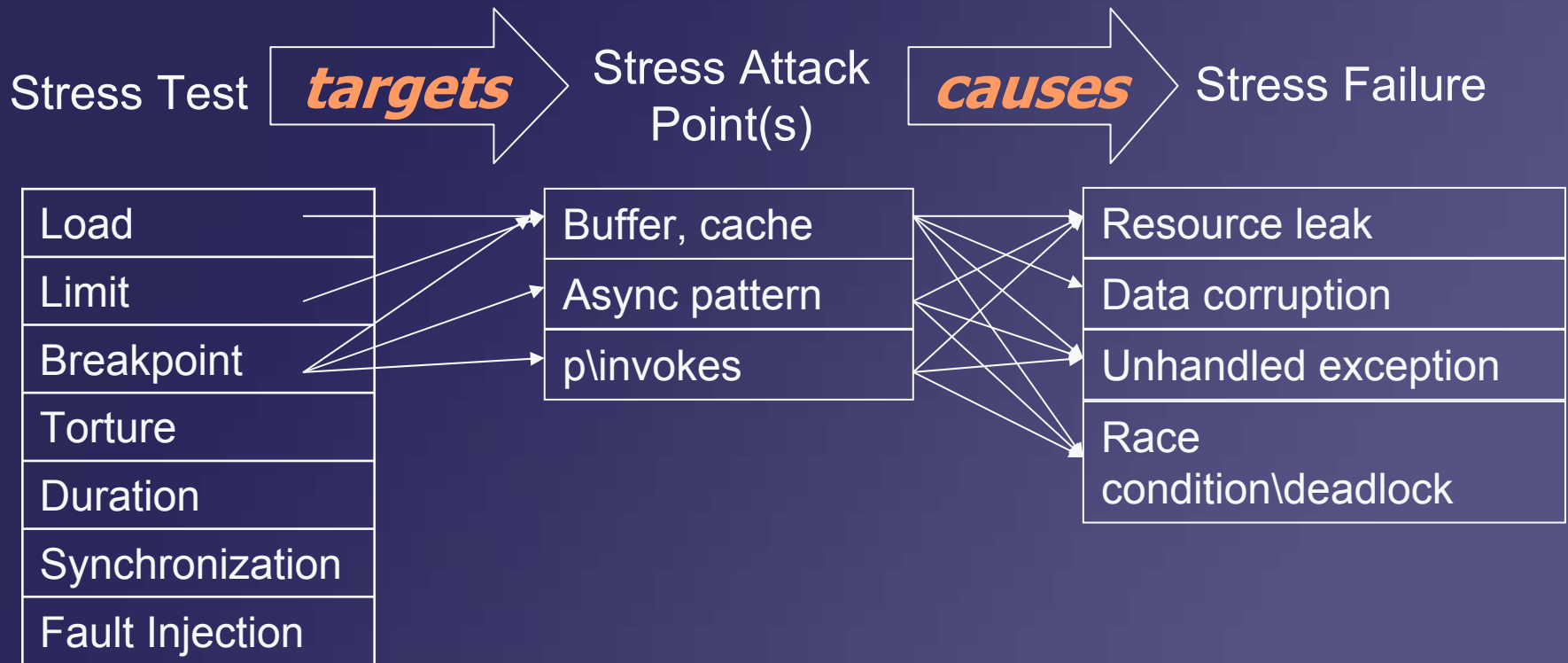
Definition: Areas that have potential for stress failures (where's the weakest link?)

- Resources
 - buffers, caches, object (e.g., thread) pooling
 - Collections, quotas, limits
- Synchronization\Timing
 - concurrent access, threading
 - locks, events, timeouts, lifetimes
 - async (BeginX, EndX) : cancels, aborts
- Integration points
 - cross-component, cross-system
 - P\Invokes or JNI , managed<->native

Flavors of Stress Tests

Load	Ensures that system can perform “acceptably” under Input and Resource load both Constant and Varying
Limit	When capacity\limits are known. Equivalent to boundary testing
Breakpoint	When capacity\limits are not known. Find heaviest load the component can handle, before failure
Torture	Taking component beyond known or pre-defined min\max limits
Duration	To run for some amount of time without undue load or failures.
Synchronization	Flush out timing\synchronization\concurrency issues e.g., multi-thread tests
Fault Injection	Deterministically inject faults, simulate error conditions

Stress Threat Analysis Modeling



Race Condition Detection

- **Primitive** (yet somewhat effective):
Random testing with “lots of things” going on concurrently.
- **Simple** :
Marick’s race condition coverage in GCT
Test Hooks or Fault Injection
 - in the SUT code (e.g. <http://www.research.ibm.com/journal/sj/411/edelstein.html>)
 - in the Thread scheduler
- **Advanced** :
 - Formal Methods (static)
 - Lockset tracking (dynamic)

Advanced Concurrency Approaches

- Formal Methods Properties (static – models)
 - Liveness (system eventually arrives in a “good” state)
 - Livelock / deadlock
 - Safety (system is never in a “bad” state)

TLA Temporal Logic of Actions

<http://research.microsoft.com/users/lamport/tla/tla.html>

SPIN

<http://spinroot.com/spin/whatispin.html>

ZING!

<http://www.research.microsoft.com/Zing/>

- Erase Lockset algorithm adaptations (dynamic)
 - Java PathExplorer (JPAX)
(<http://webcourse.cs.technion.ac.il/236801/Winter2004-2005/ho/WCFiles/Java-Path-Explorer.pdf>)
 - RaceTrack (<http://research.microsoft.com/research/sv/racetrack/>)

Summary

- Account for dependency analysis in test design
- Robust unit testing and static analysis
- Use “all pairs” for configuration testing
- Employ coverage to select and prioritize tests
- Use random testing including fuzz testing security
- Make models of systems
 - act as test oracles for automation.

References -- CheckLists

- “Common Software Errors”
from Testing Computer Software by Cem Kaner
- “Attacks”
from How to Break Software by James Whittaker
- “Test Catalog”
from Craft of Software Testing by Brian Marick
www.testing.com/writings/short-catalog.pdf
- On the web , for example
<http://blogs.msdn.com/jledgard/archive/2003/11/03/53722.aspx>

Tools and resources

1. *Architecture Achilles Heel Analysis*
by Elisabeth Hendrickson and Grant Larson
<http://www.testing.com/test-patterns/patterns/Architecture-Achilles-Heels-Analysis.pdf>
2. Testing Strategy in *System Test Pattern Language*
by David DeLano and Linda Rising
<http://members.cox.net/risingl1/articles/systemtest.htm>
3. *Fuzz Revisited: A Re-examination of the Reliability of UNIX ...*
by BP Miller, et al
<http://www.opensource.org/advocacy/fuzz-revisited.pdf>
4. *Simplifying and Isolating Failure-Inducing Input*
Andreas Zeller and Ralf Hildebrandt
IEEE Trans. on Software Engineering, 28/2, February 2002
<http://www.st.cs.uni-sb.de/papers/tse2002/>

It's Too Darn Big:

Test Techniques for Gigantic Systems

WSA 2005

Questions?