

# *Black Box Software Testing* *(Academic Course - Fall 2001)*

**Cem Kaner, J.D., Ph.D.**

Florida Institute of Technology

**Section: 22 :**

Stochastic or Random Testing

Contact Information:

**kaner@kaner.com**

**www.kaner.com (testing practitioners)**

**www.badsoftware.com (software law)**

**www.testingeducation.org (education research)**

Copyright (c) Cem Kaner 2001.

I grant permission to make digital or hard copies of this work for personal or classroom use, without fee, provided that (a) copies are not made or distributed for profit or commercial advantage, (b) copies bear this notice and full citation on the first page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion. Abstracting with credit is permitted. The proper citation for this work is Cem Kaner, *A Course in Black Box Software Testing (Academic Version)*, Fall 2001, [www.testing-education.org](http://www.testing-education.org). To copy otherwise, to republish or post on servers, or to distribute to lists requires prior specific permission and a fee. Request permission to republish from kaner@kaner.com.

# *Black Box Software Testing*

## Stochastic or Random Testing

Assigned Reading:

Kaner, *Architectures of Test Automation*

Nyman, *Application Testing With Dumb Monkeys*

Recommended Reading:

Becker & Berkemeyer, *The Application of a Software Testing Technique to Uncover Data Errors in a Database System*

Robinson, *Finite State Model-Based Testing on a Shoestring*—available at [http://www.geocities.com/model\\_based\\_testing/](http://www.geocities.com/model_based_testing/)

Whittaker, *Stochastic Software Testing*—available at [http://www.geocities.com/model\\_based\\_testing/](http://www.geocities.com/model_based_testing/)

# *Random / Statistical Testing*

- **Tag line**
  - “High-volume testing with new cases all the time.”
- **Fundamental question or goal**
  - Have the computer create, execute, and evaluate huge numbers of tests.
    - The individual tests are not all that powerful, nor all that compelling.
    - The power of the approach lies in the large number of tests.
    - These broaden the sample, and they may test the program over a long period of time, giving us insight into longer term issues.

# *Random / Statistical Testing*

- Paradigmatic case(s)
  - Some of us are still wrapping our heads around the richness of work in this field. This is a tentative classification
    - NON-STOCHASTIC RANDOM TESTS
    - STATISTICAL RELIABILITY ESTIMATION
    - STOCHASTIC TESTS (NO MODEL)
    - STOCHASTIC TESTS USING ON A MODEL OF THE SOFTWARE UNDER TEST
    - STOCHASTIC TESTS USING OTHER ATTRIBUTES OF SOFTWARE UNDER TEST

# *Random / Statistical Testing:*

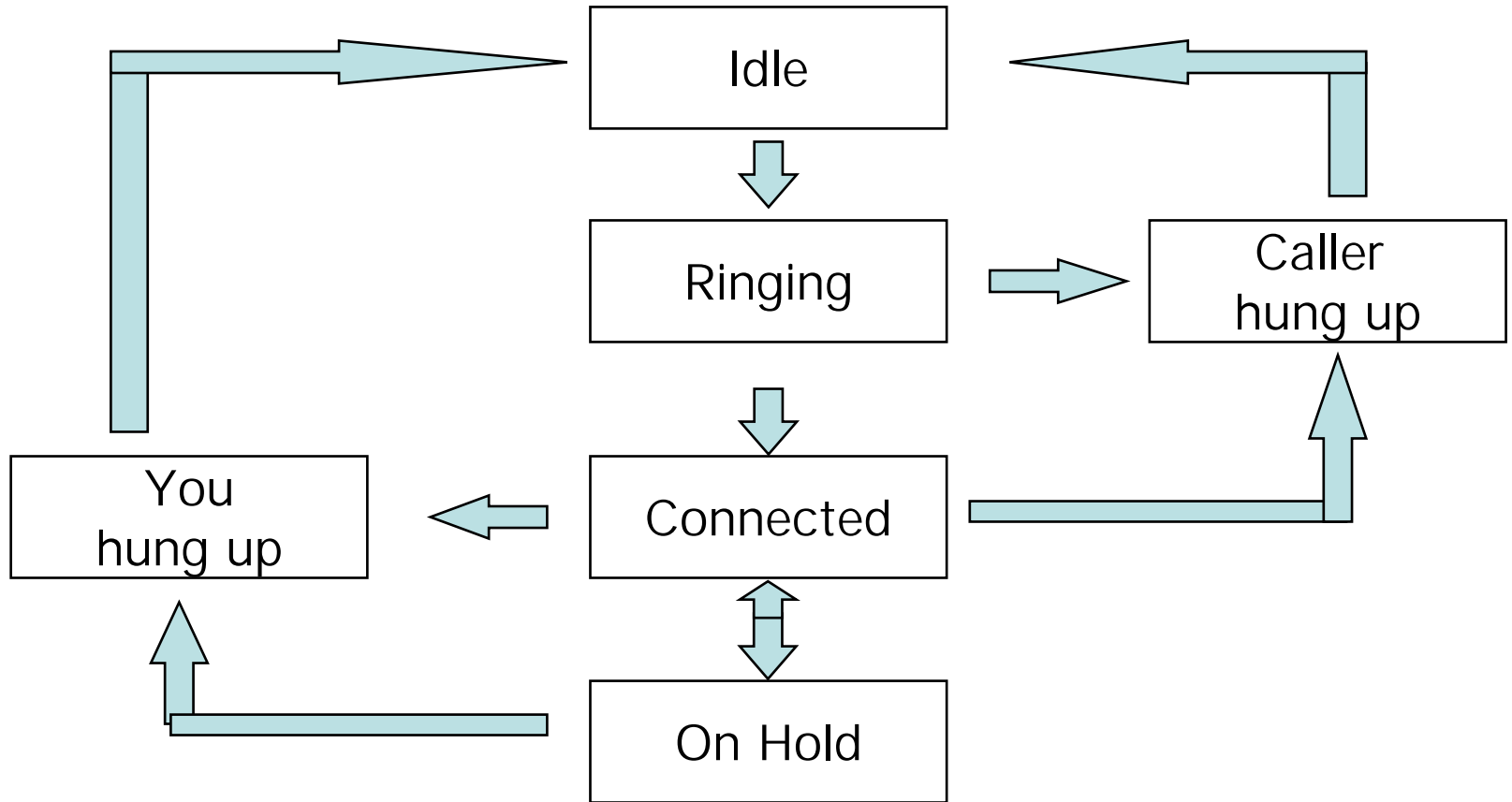
## *Non-Stochastic*

- **Fundamental question or goal**
  - The computer runs a large set of essentially independent tests. The focus is on the results of each test. Tests are often designed to minimize sequential interaction among tests.
- **Paradigmatic case(s)**
  - Function equivalence testing: Compare two functions (e.g. math functions), using the second as an oracle for the first. Attempt to demonstrate that they are not equivalent, i.e. that they achieve different results from the same set of inputs.
  - Other test using fully deterministic oracles (see discussion of oracles, below)
  - Other tests using heuristic oracles (see discussion of oracles, below)

# *Random / Statistical Testing: Statistical Reliability Estimation*

- **Fundamental question or goal**
  - Use random testing (possibly stochastic, possibly oracle-based) to estimate the stability or reliability of the software. Testing is being used primarily to qualify the software, rather than to find defects.
- **Paradigmatic case(s)**
  - Clean-room based approaches

# *The Need for Stochastic Testing: An Example*



# *Random Testing: Stochastic Tests-- No Model: “Dumb Monkeys”*

- **Fundamental question or goal**
  - High volume testing, involving a long sequence of tests.
  - A typical objective is to evaluate program performance over time.
  - The distinguishing characteristic of this approach is that the testing software does not have a detailed model of the software under test.
  - The testing software might be able to detect failures based on crash, performance lags, diagnostics, or improper interaction with other, better understood parts of the system, but it cannot detect a failure simply based on the question, “Is the program doing what it is supposed to or not?”

# *Random Testing: Stochastic Tests-- (No Model: “Dumb Monkeys”)*

- **Paradigmatic case(s)**
  - Executive monkeys: Know nothing about the system. Push buttons randomly until the system crashes.
  - Clever monkeys: More careful rules of conduct, more knowledge about the system or the environment. See Freddy.
  - O/S compatibility testing: No model of the software under test, but diagnostics might be available based on the environment (the NT example)
  - Early qualification testing
  - Life testing
  - Load testing
- **Notes**
  - Can be done at the API or command line, just as well as via UI

# *Random Testing:* *Stochastic, Assert or Diagnostics Based*

- **Fundamental question or goal**
  - High volume random testing using random sequence of fresh or pre-defined tests that may or may not self-check for pass/fail. The primary method for detecting pass/fail uses assertions (diagnostics built into the program) or other (e.g. system) diagnostics.
- **Paradigmatic case(s)**
  - Telephone example (asserts)
  - Embedded software example (diagnostics)

# *Random Testing: Stochastic, Regression-Based*

- **Fundamental question or goal**
  - High volume random testing using random sequence of pre-defined tests that can self-check for pass/fail.
- **Paradigmatic case(s)**
  - Life testing
  - Search for specific types of long-sequence defects.

# *Random Testing: Stochastic, Regression-Based*

## **Notes**

- Create a series of regression tests. Design them so that they don't reinitialize the system or force it to a standard starting state that would erase history. The tests are designed so that the automation can identify failures. Run the tests in random order over a long sequence.
- This is a low-mental-overhead alternative to model-based testing. You get pass/fail info for every test, but without having to achieve the same depth of understanding of the software. Of course, you probably have worse coverage, less awareness of your actual coverage, and less opportunity to stumble over bugs.
- Unless this is very carefully managed, there is a serious risk of non-reproducibility of failures.

# *Random Testing: Sandboxing the Regression Tests*

- In a random sequence of standalone tests, we might want to qualify each test, T1, T2, etc, as able to run on its own. Then, when we test a sequence of these tests, we know that errors are due to interactions among them rather than merely to cumulative effects of repetition of a single test.
- Therefore, for each  $T_i$ , we run the test on its own many times in one long series, randomly switching as many other environmental or systematic variables during this random sequence as our tools allow.
- We call this the “sandbox” series— $T_i$  is forced to play in its own sandbox until it “proves” that it can behave properly on its own. (This is an 80/20 rule operation. We do want to avoid creating a big random test series that crashes only because one test doesn’t like being run or that fails after a few runs under low memory. We want to weed out these simple causes of failure. But we don’t want to spend a fortune trying to control this risk.)

# *Random Testing: Model-based Stochastic Tests*

- **Fundamental Question or Goal**
  - Build a state model of the software. (The analysis will reveal several defects in itself.) Generate random events / inputs to the program. The program responds by moving to a new state. Test whether the program has reached the expected state.
- **Paradigmatic case(s)**
  - *I haven't done this kind of work. Here's what I understand:*
    - Works poorly for a complex product like Word
    - Likely to work well for embedded software and simple menus (think of the brakes of your car or walking a control panel on a printer)
    - In general, well suited to a limited-functionality client that will not be powered down or rebooted very often.
    - Maintenance is a critical issue because design changes add or subtract nodes, forcing a regeneration of the model.

# *Random Testing:*

## *Model-based Stochastic Tests*

- Alan Jorgensen, Software Design Based on Operational Modes, Ph.D. thesis, Florida Institute of Technology:
- **“The applicability of state machine modeling to mechanical computation dates back to the work of Mealy [Mealy, 1955] and Moore [Moore, 1956] and persists to modern software analysis techniques [Mills, et al., 1990, Rumbaugh, et al., 1999]. Introducing state design into software development process began in earnest in the late 1980’s with the advent of the cleanroom software engineering methodology [Mills, et al., 1987] and the introduction of the State Transition Diagram by Yourdon [Yourdon, 1989].**
- **“A deterministic finite automata (DFA) is a state machine that may be used to model many characteristics of a software program. Mathematically, a DFA is the quintuple,  $M = (Q, \Sigma, \delta, q_0, F)$  where  $M$  is the machine,  $Q$  is a finite set of states,  $\Sigma$  is a finite set of inputs commonly called the “alphabet,”  $\delta$  is the transition function that maps  $Q \times \Sigma$  to  $Q$ ,  $q_0$  is one particular element of  $Q$  identified as the initial or starting state, and  $F \subseteq Q$  is the set of final or terminating states [Sudkamp, 1988]. The DFA can be viewed as a directed graph where the nodes are the states and the labeled edges are the transitions corresponding to inputs. . . .**

# *Random Testing:*

## *Model-based Stochastic Tests*

- **“When taking this state model view of software, a different definition of software failure suggests itself: “The machine makes a transition to an unspecified state.” From this definition of software failure a software defect may be defined as: “Code, that for some input, causes an unspecified state transition or fails to reach a required state.”**
- **“Recent developments in software system testing exercise state transitions and detect invalid states. This work, [Whittaker, 1997b], developed the concept of an “operational mode” that functionally decomposes (abstracts) states. Operational modes provide a mechanism to encapsulate and describe state complexity. By expressing states as the cross product of operational modes and eliminating impossible states, the number of distinct states can be reduced, alleviating the state explosion problem.**

# *Random Testing: Model-based Stochastic Tests*

- “Operational modes are not a new feature of software but rather a different way to view the decomposition of states. All software has operational modes but the implementation of these modes has historically been left to chance. When used for testing, operational modes have been extracted by reverse engineering.”
- Alan Jorgensen, Software Design Based on Operational Modes, Ph.D. thesis, Florida Institute of Technology

# *Random / Statistical Testing*

- **Strengths**

- Regression doesn't depend on same old test every time.
- Partial oracles can find errors in young code quickly and cheaply.
- Less likely to miss internal optimizations that are invisible from outside.
- Can detect failures arising out of long, complex chains that would be hard to create as planned tests.

- **Blind spots**

- Need to be able to distinguish pass from failure. Too many people think “Not crash = not fail.”
- Executive expectations must be carefully managed.
- Also, these methods will often cover many types of risks, but will obscure the need for other tests that are not amenable to automation.

# *Random / Statistical Testing*

- **Blind spots**

- Testers might spend much more time analyzing the code and too little time analyzing the customer and her uses of the software.
- Potential to create an inappropriate prestige hierarchy, devaluating the skills of subject matter experts who understand the product and its defects much better than the automators.