

# High Volume Test Automation

**Cem Kaner**

**Professor of Software Engineering**

**Walter P. Bond**

**Associate Professor**

**Pat McGee**

**Doctoral Student**

**Florida Institute of Technology**

**October 2003**



**CENTER FOR SOFTWARE TESTING  
EDUCATION AND RESEARCH**

# Acknowledgements

- Many of the ideas in this presentation were initially jointly developed with Doug Hoffman, as we developed a course on test automation architecture, and in the Los Altos Workshops on Software Testing (LAWST) and the Austin Workshop on Test Automation (AWTA).
  - LAWST 5 focused on oracles. Participants were Chris Agruss, James Bach, Jack Falk, David Gelperin, Elisabeth Hendrickson, Doug Hoffman, Bob Johnson, Cem Kaner, Brian Lawrence, Noel Nyman, Jeff Payne, Johanna Rothman, Melora Svoboda, Loretta Suzuki, and Ned Young.
  - LAWST 1-3 focused on several aspects of automated testing. Participants were Chris Agruss, Tom Arnold, Richard Bender, James Bach, Jim Brooks, Karla Fisher, Chip Groder, Elisabeth Hendrickson, Doug Hoffman, Keith W. Hooper, III, Bob Johnson, Cem Kaner, Brian Lawrence, Tom Lindemuth, Brian Marick, Thanga Meenakshi, Noel Nyman, Jeffery E. Payne, Bret Pettichord, Drew Pritsker, Johanna Rothman, Jane Stepak, Melora Svoboda, Jeremy White, and Rodney Wilson.
  - AWTA also reviewed and discussed several strategies of test automation. Participants in the first meeting were Chris Agruss, Robyn Brilliant, Harvey Deutsch, Allen Johnson, Cem Kaner, Brian Lawrence, Barton Layne, Chang Lui, Jamie Mitchell, Noel Nyman, Barindralal Pal, Bret Pettichord, Christiano Plini, Cynthia Sadler, and Beth Schmitz.
- We're indebted to Hans Buwalda, Elisabeth Hendrickson, Noel Nyman, Harry Robinson, James Tierney, and James Whittaker for additional explanations of test architecture and/or stochastic testing.
- We also appreciate the assistance and hospitality of "Mentsville," a well-known and well-respected, but can't-be-named-here, manufacturer of mass-market devices that have complex firmware. Mentsville opened its records to us, providing us with details about a testing practice (Extended Random Regression testing) that has been evolving at the company since 1990.
- Finally, we thank Alan Jorgenson for explaining hostile data stream testing to us and providing equipment and training for us to use to extend his results.

# Typical Testing Tasks

- Analyze product & its risks
  - market
  - benefits & features
  - review source code
  - platform & associated software
- Develop testing strategy
  - pick key techniques
  - prioritize testing foci
- Design tests
  - select key test ideas
  - create test for the idea
- Run test first time (often by hand)
- Evaluate results
  - Report bug if test fails
- Keep archival records
  - trace tests back to specs
- Manage testware environment
- If we create regression tests:
  - Capture or code steps once test passes
  - Save “good” result
  - Document test / file
  - Execute the test
    - Evaluate result
      - Report failure or
      - Maintain test case

# Automating Testing

- No testing tool covers this range of tasks
- We should understand that
  - “Automated testing” doesn’t mean automated testing
  - “Automated testing” means  
*Computer-Assisted Testing*

# Automated GUI-Level Regression Testing

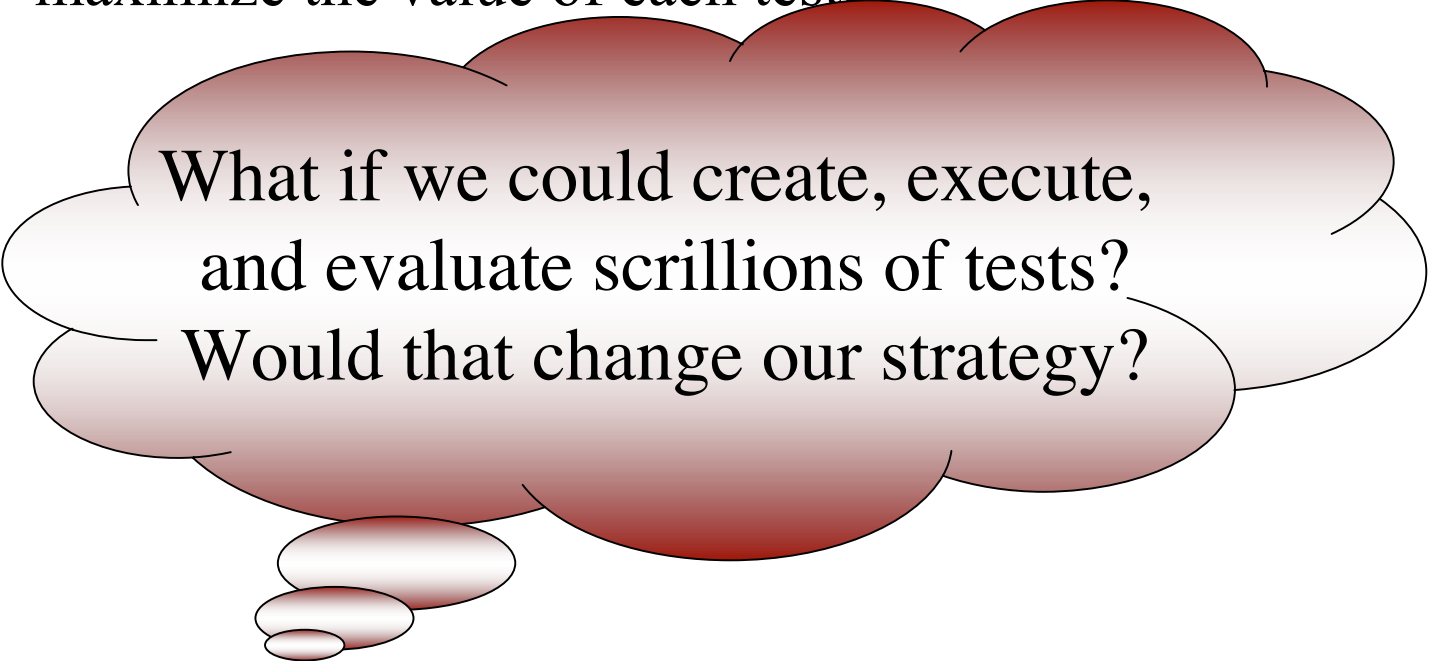
- Re-use old tests using tools like Mercury, Silk, Robot
- Low power
- High maintenance cost
- Significant inertia

## **INERTIA**

*The resistance to change that  
our development process  
builds into the project.*

# The Critical Problem of Regression Testing

- Very few tests
- We are driven by the politics of scarcity:
  - too many potential tests
  - not enough time
- Every test is lovingly crafted, or should be, because we need to maximize the value of each test.



What if we could create, execute,  
and evaluate scillions of tests?  
Would that change our strategy?

# Case Study: Extended Random Regression

- Welcome to “Mentenville”, a household-name manufacturer, widely respected for product quality, who chooses to remain anonymous.
- Mentenville applies wide range of tests to their products, including unit-level tests and system-level regression tests.
  - We estimate > 100,000 regression tests in “active” library
- Extended Random Regression (ERR)
  - Tests taken from the pool of tests ***the program has passed in this build***
  - The tests sampled are run in random order until the software under test fails (e.g crash)
  - These tests add nothing to typical measures of coverage.
  - Should we expect these to find bugs?

# Extended Random Regression Testing

- Typical defects found include timing problems, memory corruption (including stack corruption), and memory leaks.
- Recent release: 293 reported failures exposed 74 distinct bugs, including 14 showstoppers.
- Mentsville's assessment is that **ERR exposes problems that can't be found in less expensive ways.**
  - troubleshooting of these failures can be very difficult and very expensive
  - wouldn't want to use ERR for basic functional bugs or simple memory leaks--too expensive.
- ERR has gradually become one of the fundamental techniques relied on by Mentsville
  - gates release from one milestone level to the next.



# Implications of ERR for Reliability Models

- Most models of software reliability make several common assumptions, including:
  - Every fault (perhaps, within a given severity class) has the same chance of being encountered as every other fault.
  - Probability of fault detection in a given period of time is directly related to the number of faults left in the program.(Source (example) Farr (1995) “Software Reliability Modeling Survey,” in Lyu (ed.) *Software Reliability Engineering*.)
- Additionally, the following ideas are **foreign** to most models:
  - a) There are different kinds of faults (different detection probabilities)
  - b) There are different kinds of tests (different exposure probabilities)
  - c) The power of one type of test can diminish over time, without a correlated loss of power of some other type of test.
  - d) The probability of exposing a given kind of fault depends in large part on which type of test you’re using.

ERR demonstrates (d).

# Summary So Far

- Traditional test techniques tie us to a small number of tests.
- Extended Random Regression exposes bugs the traditional techniques probably won't find.
- The results of Extended Random Regression provide another illustration of the weakness of current models of software reliability.



OK, so it finds some bugs.  
So what?  
Are there any types of software  
for which  
tests like this are really important  
or is this just another  
technique for the textbooks?

# Ten Examples of HVAT

1. Extended random regression testing
2. Function equivalence testing (comparison to a reference function)
3. Comparison to a computational or logical model
4. Comparison to a heuristic prediction, such as prior behavior
5. Simulator with probes
6. State-transition testing without a state model (dumb monkeys)
7. State-transition testing using a state model (terminate on failure rather than after achieving some coverage criterion)
8. Functional testing in the presence of background load
9. Hostile data stream testing
10. Random inputs to protocol checkers

# A Structure for Thinking about HVAT

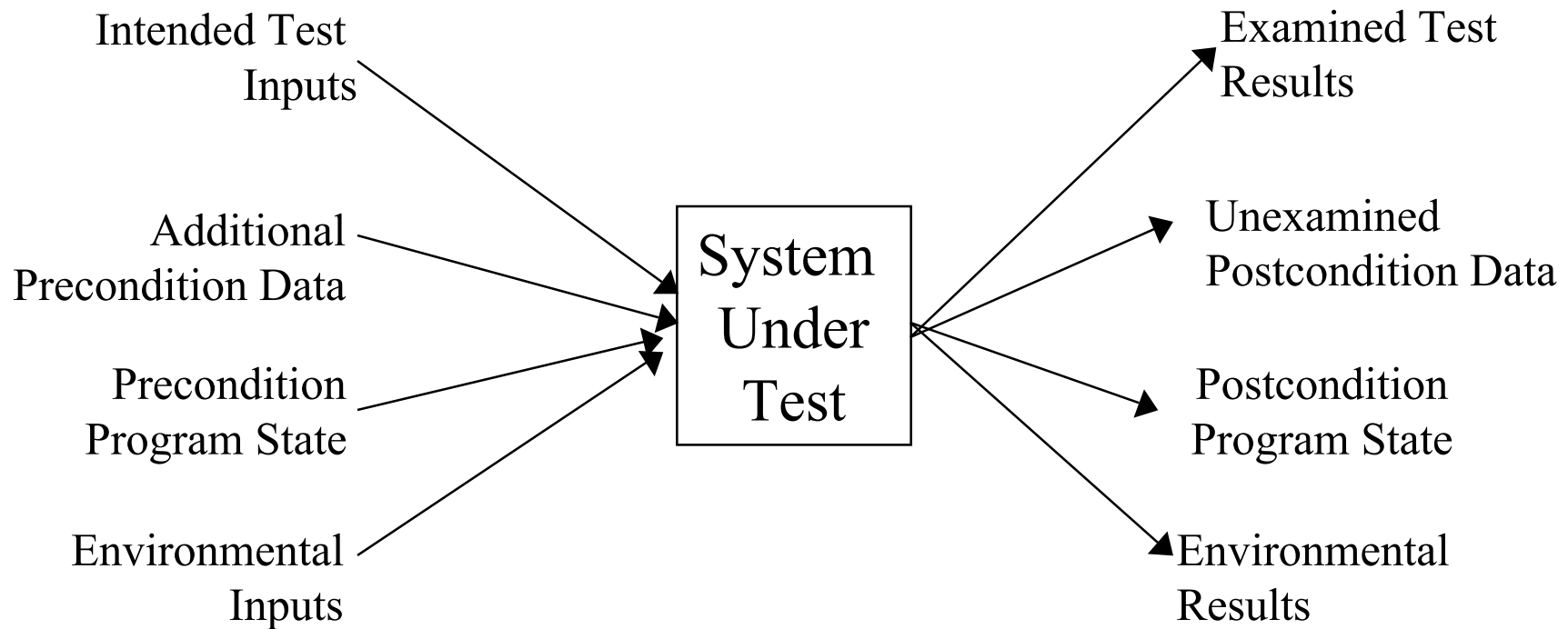
- INPUTS
  - What is the source for our inputs?  
How do we choose input values for the test?
  - (“Input” includes the full set of conditions of the test)
- OUTPUTS
  - What outputs will we observe?
- EVALUATION
  - How do we tell whether the program passed or failed?
- EXPLICIT MODEL?
  - Is our testing guided by any explicit model of the software, the user, the process being automated, or any other attribute of the system?
- WHAT ARE WE MISSING?
  - The test highlights some problems but will hide others.
- SEQUENCE OF TESTS
  - Does / should any aspect of test N+1 depend on test N?
- THEORY OF ERROR
  - What types of errors are we hoping to find with these tests?
- TROUBLESHOOTING SUPPORT
  - What data are stored? How else is troubleshooting made easier?
- BASIS FOR IMPROVING TESTS?
- HOW TO MEASURE PROGRESS?
  - How much, and how much is enough?
- MAINTENANCE LOAD / INERTIA?
  - Impact of / on change to the SUT
- CONTEXTS
  - When is this useful?

# Mentsville ERR and the Structure

- INPUTS:
  - taken from existing regression tests, which were designed under a wide range of criteria
- OUTPUTS
  - Mentsville: few of interest other than diagnostics
  - Others: whatever outputs were interesting to the regression testers, plus diagnostics
- EVALUATION STRATEGY
  - Mentsville: run until crash or other obvious failure
  - Others: run until crash or until mismatch between program behavior or prior results or model predictions
- EXPLICIT MODEL?
  - None
- WHAT ARE WE MISSING?
  - Mentsville: Anything that doesn't cause a crash
- SEQUENCE OF TESTS
  - ERR sequencing is random
- THEORY OF ERROR
  - bugs not easily detected by the regression tests: long-fuse bugs, such as memory corruption, memory leaks, timing errors
- TROUBLESHOOTING SUPPORT
  - diagnostics log, showing state of system before and after tests

## *What Are We Missing?*

We Miss Much of the Actual Test Input and Results



Adapted, with permission of Doug Hoffman

# Function Equivalence Testing

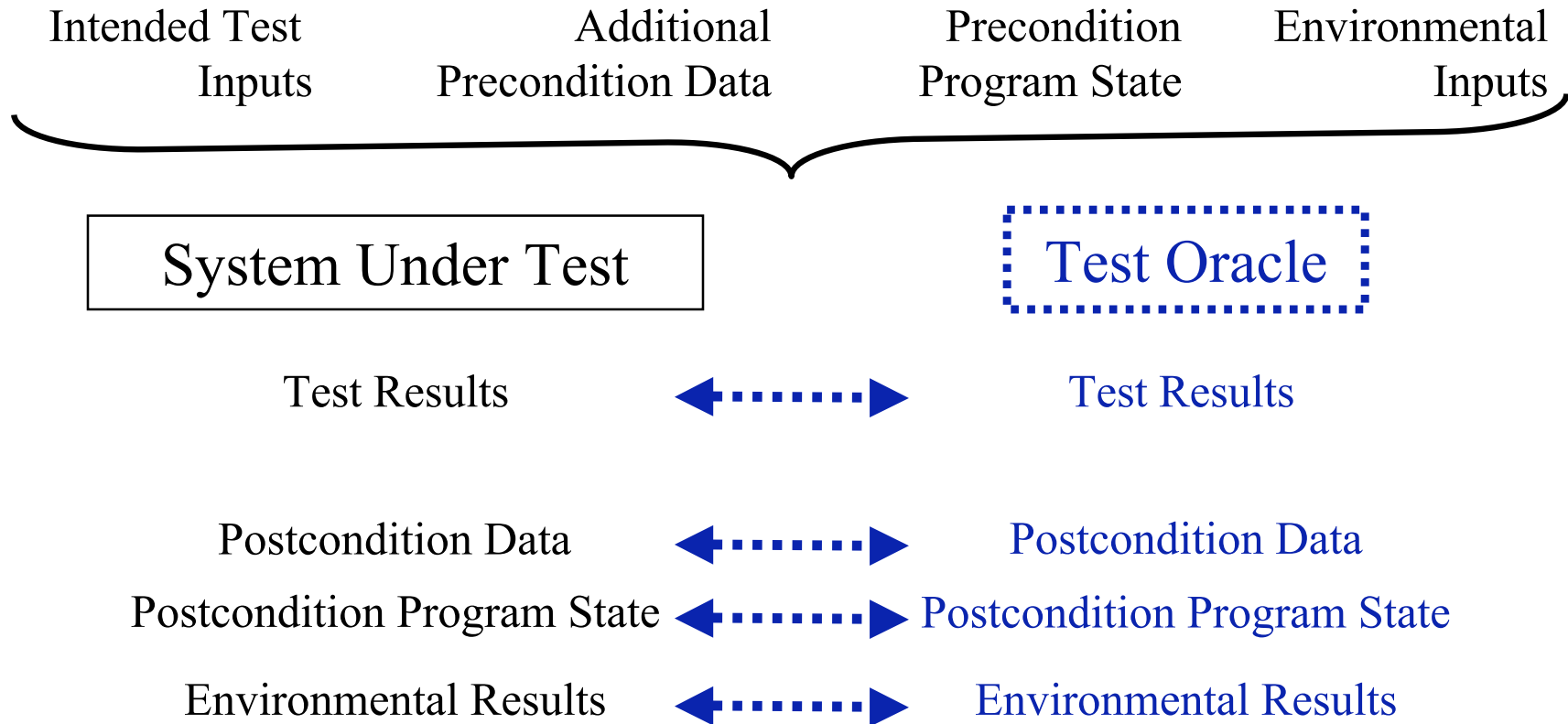
- Example from the Testing 2 final exam last spring:
  - Test Open Office spreadsheet by comparing it with Excel
  - (We used COM interface for Excel and an equivalent interface for OO, drove the API-level tests using a simple scripting language, Ruby)
  - Pick a function in OO (and Excel)
  - Generate random input to the function
  - Compare OO evaluation and Excels
  - Continue until you find errors or are satisfied of the equivalence of the two functions.
  - Now test expressions that combine several of the tested functions



# Function Equivalence Testing

- INPUTS:
  - Random
- OUTPUTS
  - We compare output with the output from a reference function. In practice, we also independently check a small sample of calculations for plausibility
- EVALUATION STRATEGY
  - Output fails to match, or fails to match within delta, or testing stops from crash or other obvious misbehavior.
- EXPLICIT MODEL?
  - The reference function is, in relevant respects, equivalent to the software under test.
  - If we combine functions (testing expressions rather than single functions), we need a grammar or other basis for describing combinations.
- WHAT ARE WE MISSING?
  - Anything that the reference function can't generate
- SEQUENCE OF TESTS
  - Tests are typically independent
- THEORY OF ERROR
  - Incorrect data processing / storage / calculation
- TROUBLESHOOTING SUPPORT
  - Inputs saved
- BASIS FOR IMPROVING TESTS?

# Oracle comparisons are heuristic: We compare only a few result attributes



Modified from notes by Doug Hoffman

What does *this one*  
have to do with  
reliability models?



What is this technique  
useful for?



# Summary So Far

- Traditional test techniques tie us to a small number of tests.
- Extended Random Regression exposes bugs the traditional techniques probably won't find.
- The results of Extended Random Regression provide another illustration of the weakness of current models of software reliability.
- ERR is just one example of a class of high volume tests
- High volume tests are useful for:
  - exposing delayed-effect bugs
  - automating tedious comparisons, for any testing task that can be turned into tedious comparisons
- Test oracles are incomplete.
  - If we rely on them too heavily, we'll miss bugs

# Phone System: Simulator with Probes



Telenova Station Set 1. Integrated voice and data.  
108 voice features, 110 data features. 1985.

# Simulator with Probes

```
July 4, 1985      12:01 PM   Ext: 257  
Directory Admin  Messages  Voice Data
```

```
1-(212)662-7777 Connected   Ext: 567  
Transfer Record  Confernce Park  Acct
```

```
Please enter selection  
LVMsa   GetMsa   Greeting  Code
```

```
Ted K. waiting      Wt:1 Hd:0  
I'llCall  CallLater PlsWait  Answ
```

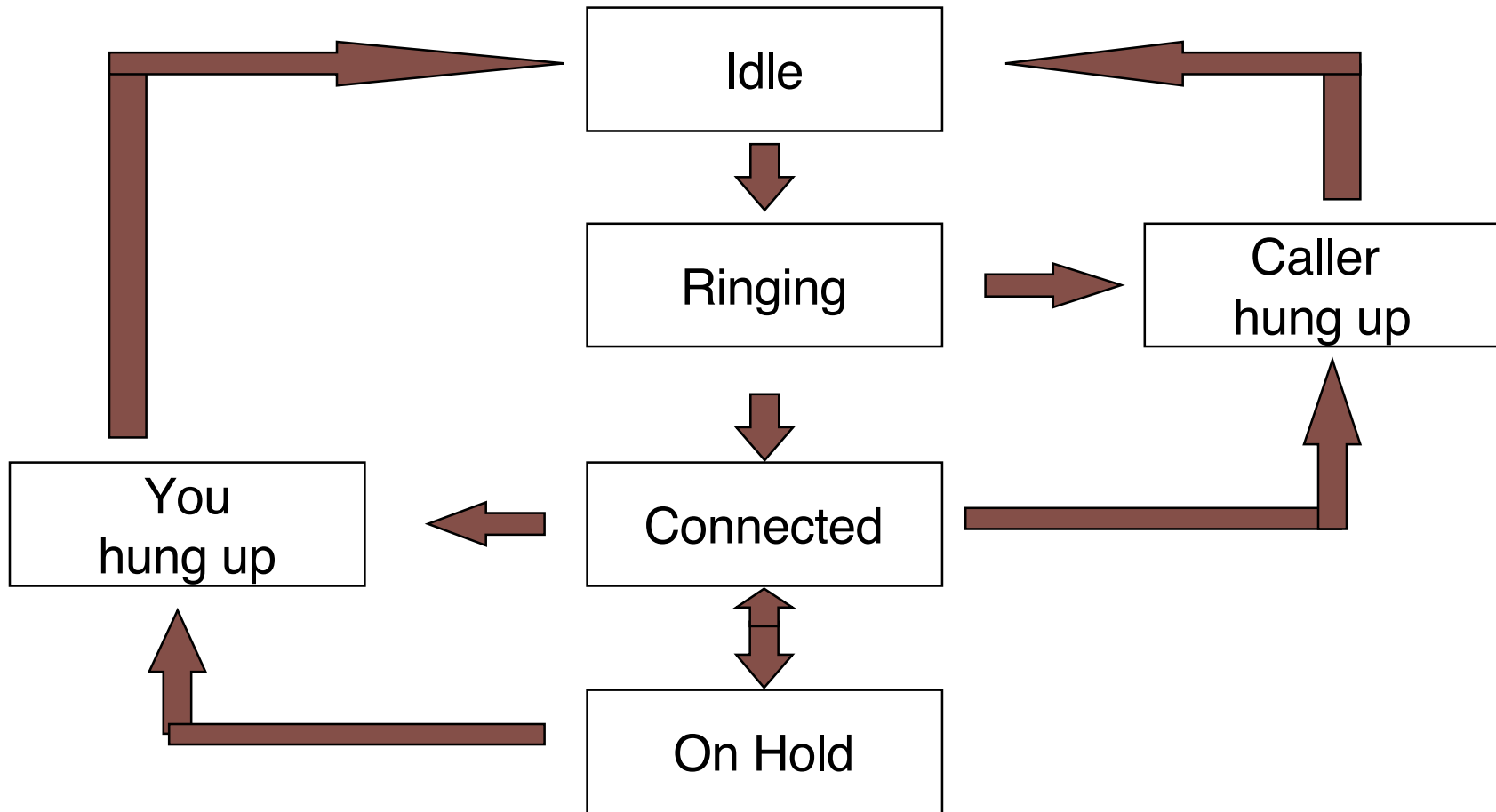
```
Select a call & lift handset  Wt:5 Hd:5  
Ted K.   Peter T.  Trunk 6  Trk 2Trk 7
```

```
Kenix 3 Connected for Data  
Transfer  Baud      EndCall  Park Acct
```



Context-sensitive  
display  
10-deep hold queue  
10-deep wait queue

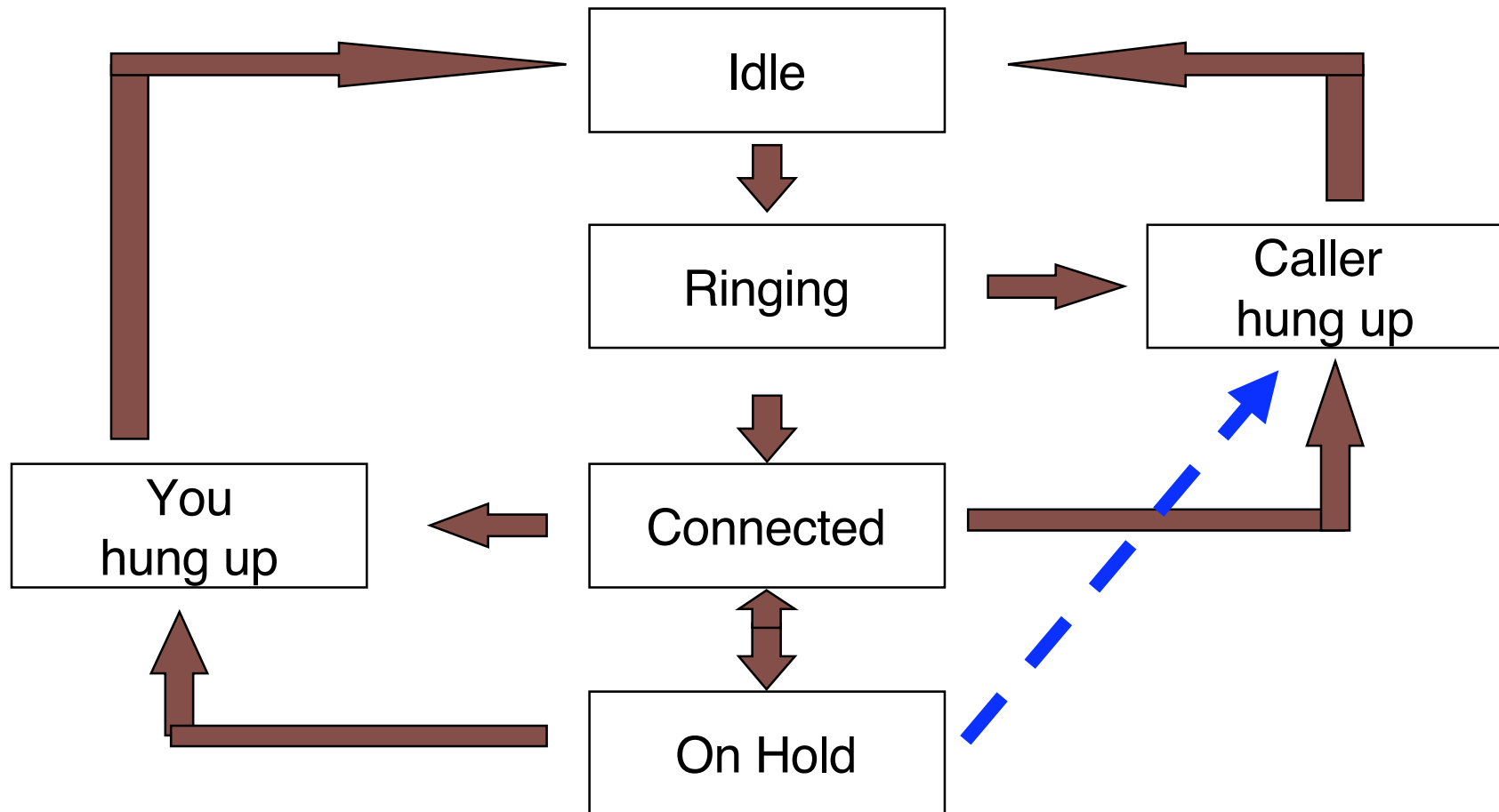
# Simulator with Probes



Simplified state diagram



# Simulator with Probes



Cleaned up everything *but* the stack. Failure was invisible until crash. From there, held calls were hold-forwarded to other phones, causing a rotating outage.

# Simulator with Probes

The bug that triggered the simulation looked like this:

- Beta customer (a stock broker) reported random failures
  - Could be frequent at peak times
  - An individual phone would crash and reboot, with other phones crashing while the first was rebooting
  - On a particularly busy day, service was disrupted all (East Coast) afternoon
- We were mystified:
  - All individual functions worked
  - We had tested all lines and branches.
- Ultimately, we found the bug in the hold queue
  - Up to 10 calls on hold, each adds record to the stack
  - Initially, checked stack whenever call was added or removed, but this took too much system time
  - Stack has room for 20 calls (just in case)
  - Stack reset (forced to zero) when we knew it *should* be empty
  - The error handling made it almost impossible for us to detect the problem in the lab. Because we couldn't put more than 10 calls on the stack (unless we knew the magic error), we couldn't get to 21 calls to cause the stack overflow.

# Simulator with Probes

**Having found and fixed  
the hold-stack bug,  
should we assume  
that we've taken care of the problem  
or that if there is one long-sequence bug,  
there will be more?**

**Hmmm...  
If you kill a cockroach in your kitchen,  
do you assume  
you've killed the last bug?  
Or do you call the exterminator?**



# Simulator with Probes

- Telenova (\*) created a simulator
  - generated long chains of random events, emulating input to the system's 100 phones
  - could be biased, to generate more holds, more forwards, more conferences, etc.
- Programmers added probes (non-crashing asserts that sent alerts to a printed log) selectively
  - can't probe everything b/c of timing impact
- After each run, programmers and testers tried to replicate failures, fix anything that triggered a message. After several runs, the logs ran almost clean.
- At that point, shift focus to next group of features.
- Exposed lots of bugs

(\*) By the time this was implemented, I had joined Electronic Arts.

# Simulator with Probes

- INPUTS:
  - Random, but with biasable transition probabilities.
- OUTPUTS
  - Log messages generated by the probes. These contained some troubleshooting information (whatever the programmer chose to include).
- EVALUATION STRATEGY
  - Read the log, treat any event leading to a log message as an error.
- EXPLICIT MODEL?
  - At any given state, the simulator knows what the SUT's options are, but it doesn't verify the predicted state against actual state.
- WHAT ARE WE MISSING?
  - Any behavior other than log
- SEQUENCE OF TESTS
  - Ongoing sequence, never reset.
- THEORY OF ERROR
  - Long-sequence errors (stack overflow, memory corruption, memory leak, race conditions, resource deadlocks)
- TROUBLESHOOTING SUPPORT
  - Log messages
- BASIS FOR IMPROVING TESTS?
  - Clean up logs after each run by eliminating false alarms and fixing bugs. Add more tests and log details for hard-to-repro errors

# Summary

- Traditional test techniques tie us to a small number of tests.
- Extended random regression and long simulations exposes bugs the traditional techniques probably won't find.
- Extended random regression and simulations using probes provide another illustration of the weakness of current models of software reliability.
- ERR is just one example of a class of high volume tests
- High volume tests are useful for:
  - exposing delayed-effect bugs
    - embedded software
    - life-critical software
    - military applications
    - operating systems
    - anything that isn't routinely rebooted
  - automating tedious comparisons, for any testing task that can be turned into tedious comparisons
- Test oracles are incomplete.
  - If we rely on them too heavily, we'll miss bugs

# Where We're Headed

## ***1. Enable the adoption and practice of this technique***

- Find and describe compelling applications (motivate adoption)
- Build an understanding of these as a class, with differing characteristics
  - vary the characteristics to apply to a new situation
  - further our understanding of relationship between context and the test technique characteristics
- Create usable examples:
  - free software, readable, sample code
  - applied well to an open source program

## ***2. Critique and/or fix the reliability models***

# A Few More Examples

- We aren't discussing these in the talk
- These just provide a few more illustrations that you might work through in your spare time.



# State Transition Testing

- State transition testing is *stochastic*. It helps to distinguish between independent random tests and stochastic tests.
- Random Testing
  - Random (or statistical or stochastic) testing involves generating test cases using a random number generator. Individual test cases are not optimized against any particular risk. The power of the method comes from running large samples of test cases.
- Independent Random Testing
  - Our interest is in each test individually, the test before and the test after don't matter.
- Stochastic Testing
  - A stochastic process involves a series of random events over time
    - Stock market is an example
    - Program may pass individual tests when run in isolation: The goal is to see whether it can pass a large series of the individual tests.

# State Transition Tests Without a State Model: Dumb Monkeys

- Phrase coined by Noel Nyman. Many prior uses (UNIX kernel, Lisa, etc.)
- Generate a long sequence of random inputs driving the program from state to state, but without a state model that allows you to check whether the program has hit the correct next state.
  - **Executive Monkey:** (dumbest of dumb monkeys) Press buttons randomly until the program crashes.
  - **Clever Monkey:** No state model, but knows other attributes of the software or system under test and tests against those:
    - Continues until crash *or* a diagnostic event occurs. The diagnostic is based on knowledge of the system, not on internals of the code. (Example: button push doesn't push—this is system-level, not application level.)
    - Simulator-with-probes is a clever monkey
- Nyman, N. (1998), “Application Testing with Dumb Monkeys”, *STAR Conference West*.

# Dumb Monkey

- INPUTS:
  - Random generation.
  - Some commands or parts of system may be blocked (e.g. format disk)
- OUTPUTS
  - May ignore all output (executive monkey) or all but the predicted output.
- EVALUATION STRATEGY
  - Crash, other blocking failure, or mismatch to a specific prediction or reference function.
- EXPLICIT MODEL?
  - None
- WHAT ARE WE MISSING?
  - Most output. In practice, dumb monkeys often lose power quickly (i.e. the program can pass it even though it is still full of bugs).
- SEQUENCE OF TESTS
  - Ongoing sequence, never reset
- THEORY OF ERROR
  - Long-sequence bugs
  - Specific predictions if some aspects of SUT are explicitly predicted
- TROUBLESHOOTING SUPPORT
  - Random number generator's seed, for reproduction.
- BASIS FOR IMPROVING TESTS?

# State Transitions: State Models (Smart Monkeys)

- For any state, you can list the actions the user can take, and the results of each action (what new state, and what can indicate that we transitioned to the correct new state).
- Randomly run the tests and check expected against actual transitions.
- See [www.geocities.com/model\\_based\\_testing/online\\_papers.htm](http://www.geocities.com/model_based_testing/online_papers.htm)
- The most common state model approach seems to drive to a level of coverage, use Chinese Postman or other algorithm to achieve all sequences of length N. (A lot of work along these lines at Florida Tech)
  - **High volume approach runs sequences until failure appears or the tester is satisfied that no failure will be exposed.**
- Coverage-oriented testing fails to account for the problems associated with multiple runs of a given feature or combination.
- Robinson, H. (1999a), “Finite State Model-Based Testing on a Shoestring”, *STAR Conference West*. Available at [www.geocities.com/model\\_based\\_testing/shoestring.htm](http://www.geocities.com/model_based_testing/shoestring.htm).
- Robinson, H. (1999b), “Graph Theory Techniques in Model-Based Testing”, *International Conference on Testing Computer Software*. Available at [www.geocities.com/model\\_based\\_testing/model-based.htm](http://www.geocities.com/model_based_testing/model-based.htm).
- Whittaker, J. (1997), “Stochastic Software Testing”, *Annals of Software Engineering*, 4, 115-131.

# State-Model Based Testing

- INPUTS:
  - Random, may be guided or constrained by a model
- OUTPUTS
  - The state model predicts values for one or more reference variables that tell us whether we reached the expected state.
- EVALUATION STRATEGY
  - Crash or other obvious failure. Compare to prediction from state model.
- EXPLICIT MODEL?
  - Detailed state model or simplified model: *operational modes*.
- WHAT ARE WE MISSING?
  - The test highlights some relationships and hides others.
- SEQUENCE OF TESTS
  - Does any aspect of test N+1 depend on test N?
- THEORY OF ERROR
  - What types of errors are we hoping to find with these tests?
- TROUBLESHOOTING SUPPORT
  - What data are stored? How else is troubleshooting made easier?
- BASIS FOR IMPROVING TESTS?

# Hostile Data Stream Testing

- Pioneered by Alan Jorgenson (FIT, recently retired)
- Take a “good” file in a standard format (e.g. PDF)
  - corrupt it by substituting one string (such as a really, really huge string) for a much shorter one in the file
  - feed it to the application under test
  - Can we overflow a buffer?
- Corrupt the “good” file in thousands of different ways, trying to distress the application under test each time.
- Jorgenson and his students showed serious security problems in some products, primarily using brute force techniques.
- Method seems appropriate for application of genetic algorithms or other AI to optimize search.

# Hostile Data Stream and HVAC

- INPUTS:
  - A series of random mutations of the base file
- OUTPUTS
  - Simple version--not of much interest
- EVALUATION STRATEGY
  - Run until crash, then investigate
- EXPLICIT MODEL?
  - None
- WHAT ARE WE MISSING?
  - Data corruption, display corruption, anything that doesn't stop us from further testing
- SEQUENCE OF TESTS
  - Independent selection (without repetition). No serial dependence.
- THEORY OF ERROR
  - What types of errors are we hoping to find with these tests?
- TROUBLESHOOTING SUPPORT
  - What data are stored? How else is troubleshooting made easier?
- BASIS FOR IMPROVING TESTS?
  - Simple version: hand-tuned
  - Seemingly obvious candidate for GA's and other AI