Black Box Software Testing

2004 Academic Edition

by

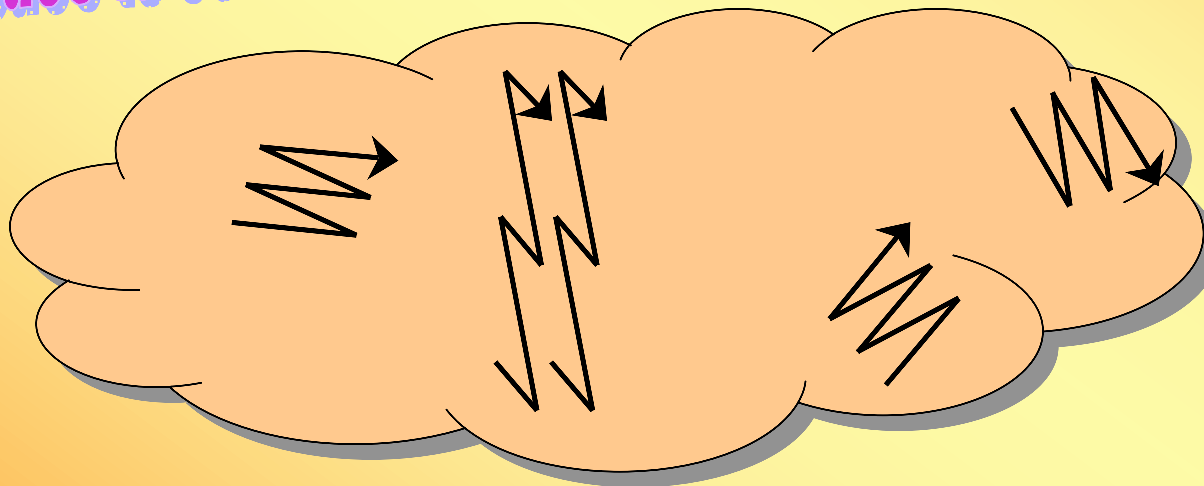
Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology
and
James Bach
Principal, Satisfice Inc.

These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Kaner & Bach grant permission to make digital or hard copies of this work for personal or classroom use, including use in commercial courses, provided that (a) Copies are not made or distributed outside of classroom use for profit or commercial advantage, (b) Copies bear this notice and full citation on the front 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 "Black Box Software Testing (Course Notes, Academic Version, 2004) www.testingeducation.org", (c) Each page that you use from this work must bear the notice "Copyright (c) Cem Kaner and James Bach, kaner@kaner.com", or if you modify the page, "Modified slide, originally from Cem Kaner and James Bach", and (d) If a substantial portion of a course that you teach is derived from these notes, advertisements of that course should include the statement, "Partially based on materials provided by Cem Kaner and James Bach." 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 Cem Kaner, kaner@kaner.com.

Black Box Software Testing

Part 2
Complete Testing Is Impossible

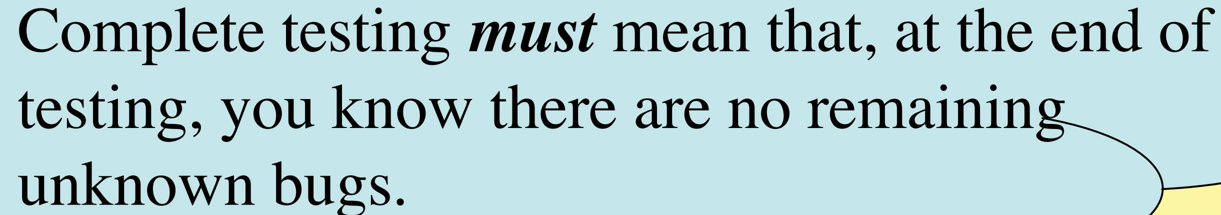
the space to search is vast...



...and your resources are meager.

Complete testing?

- What ***do*** we mean by "complete testing"?
  - Complete "coverage": Tested every line / branch / basis path?
  - Testers not finding new bugs?
  - Test plan complete?



  - After all, if there are more bugs, you can find them if you do more testing. So testing couldn't yet be "complete."

Complete coverage?

- Some people (attempt to) simplify away the problem of *complete testing* by advocating "*complete coverage*."
- What is coverage?
  - Extent of testing of certain attributes or pieces of the program, such as statement coverage or branch coverage or condition coverage.
  - Extent of testing completed, compared to a population of possible tests.
- Typical definitions are oversimplified. They miss, for example,
  - Interrupts and other parallel operations
  - Interesting data values and data combinations
  - Missing code
- The number of variables we might measure is stunning. I (Kaner) listed 101 examples in *Software Negligence & Testing Coverage.*

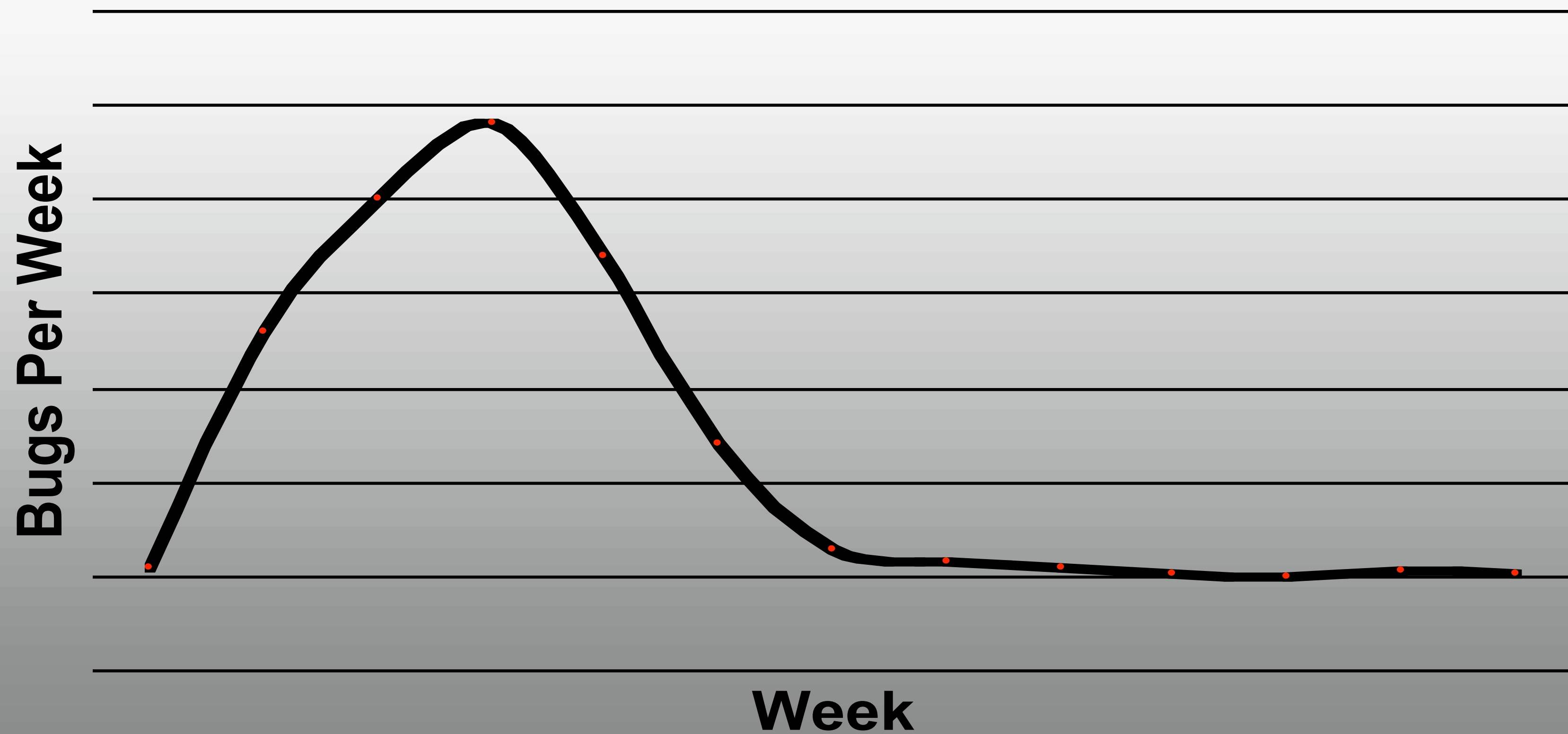
Measuring and achieving high coverage

- Coverage measurement is a useful and interesting way to tell that you are far away from complete testing.
- But testing in order to achieve "high" coverage is likely to result in development of a mass of low-power tests.
  - People optimize what we measure them against, at the expense of what we don't measure.
    - For more on measurement distortion and dysfunction, read Bob Austin's book, *Measurement and Management of Performance in Organizations.*
  - Brian Marick, raises this and several other issues in his papers at www.testing.com (e.g. How to Misuse Code Coverage). Brian has been involved in development of several of the commercial coverage tools.

What about low bug find rates?

- Another way people measure completeness, or extent, of testing is by plotting bug curves, such as:
  - New bugs found per week
  - Bugs still open (each week)
  - Ratio of bugs found to bugs fixed (per week)
- These curves can be useful progress indicators.
- But some people tell us to fit the data to a theoretical curve, often a probability distribution, and read our position from the curve. At some point, it is "clear" from the curve that we're done.

The bug curve



A common model (Weibull) and its assumptions

- Testing occurs in a way that is similar to the way the software will be operated.
- All defects are equally likely to be encountered.
- All defects are independent.
- There is a fixed, finite number of defects in the software at the start of testing.
- The time to arrival of a defect follows the Weibull distribution.
- The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals.

The Weibull distribution

- I think it is absurd to rely on a distributional model (or any model) when every assumption it makes about testing is obviously false.
- One of the advocates of this approach points out that

  "Luckily, the Weibull is robust to most violations."

  – This illustrates the use of surrogate measures—we don't have an attribute description or model for the attribute we really want to measure, so we use something else, that is allegedly "robust", in its place. This can be very dangerous

  – The Weibull distribution has a shape parameter that allows it to take a very wide range of shapes. If you have a curve that generally rises then falls (one mode), you can approximate it with a Weibull.

BUT WHAT DOES THAT TELL US? HOW SHOULD WE INTERPRET IT?

Side effects of bug curves

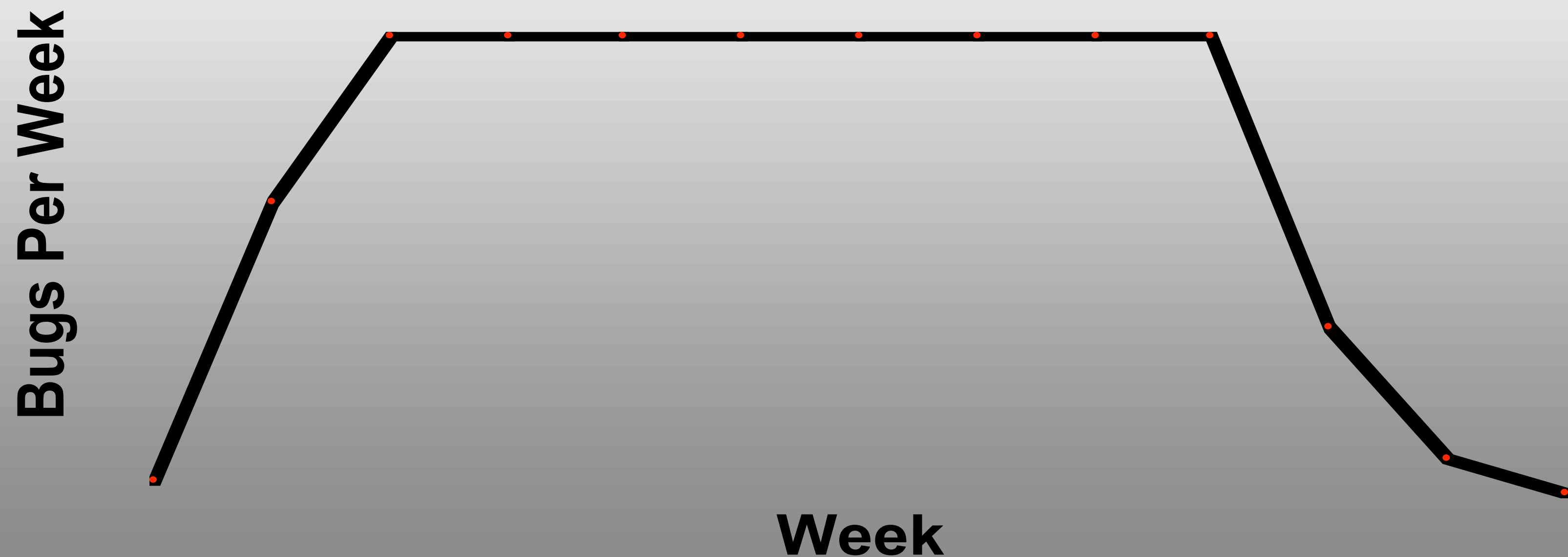
- Earlier in testing, the pressure is to increase bug counts. In response, testers will:
  - Run tests of features known to be broken or incomplete.
  - Run multiple related tests to find multiple related bugs.
  - Look for easy bugs in high quantities rather than hard bugs.
  - Less emphasis on infrastructure, automation architecture, tools and more emphasis of bug finding. (Short term payoff but long term inefficiency.)
- For more on measurement dysfunction, read Bob Austin's book, *Measurement and Management of Performance in Organizations*.
- For more observations of problems like these in reputable software companies, see Doug Hoffman's article, *The Dark Side of Software Metrics.*

Side effects of bug curves

- Later in testing, there is pressure to decrease the new bug rate:
  - Run lots of already-run regression tests.
  - Don't look as hard for new bugs.
  - Shift focus to appraisal, status reporting.
  - Classify unrelated bugs as duplicates.
  - Class related bugs as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
  - Postpone bug reporting until after the measurement checkpoint (milestone). (Some bugs are lost.)
  - Report bugs informally, keeping them out of the tracking system.
  - Testers get sent to the movies before measurement checkpoints.
  - Programmers ignore bugs they find until testers report them.
  - Bugs are taken personally.
  - More bugs are rejected.

Copyright © 2003 Cem Kaner & James Bach

Bad models are counterproductive



Testers live and breathe tradeoffs

- When you get past the simplistic answers, you realize:

  The time needed for test-related tasks is infinitely larger than the time available.

- Example: Time you spend on

  - analyzing, troubleshooting, and effectively describing a failure

  is time no longer available for

  - Designing tests
  - Executing tests
  - Reviews, inspections
  - Retooling
  - Documenting tests
  - Automating tests
  - Supporting tech support
  - Training other staff

Let's consider the nature of the infinite set of tests

- There are enormous numbers of possible tests. To test everything, you would have to:
  - Test every possible input to every variable (including output variables and intermediate results variables).
  - Test every possible combination of inputs to every combination of variables.
  - Test every possible sequence through the program.
  - Test every hardware / software configuration, including configurations of servers not under your control.
  - Test every way in which any user might try to use the program.

» Read Testing Computer Software, p. 17 - 22

 Copyright © 2003 Cem Kaner & James Bach

Inputs to single variables

Valid inputs

- –Doug Hoffman worked for MASPAR (the Massively Parallel computer, 64K parallel processors). This machine is used for mission-critical and life-critical applications.
- –To test the 32-bit integer square root function, Hoffman checked all values (all 4,294,967,296 of them). This took the computer about 6 minutes to run the tests and compare the results to an oracle.
- –There were 2 (two) errors, neither of them near any boundary. (The underlying error was that a bit was sometimes mis-set, but in most error cases, there was no effect on the final calculated result.) Without an exhaustive test, these errors probably wouldn't have shown up.
- –***What about the 64-bit integer square root?*** How could we find the time to run all of these? If we don't run them all, don't we risk missing some bugs?

Inputs to single variables

- Easter Eggs
  - Bizarre inputs, by design
- Edited inputs
  - These can be quite complex. How much editing is enough?
- Variations on input timing
  - Try entering the data very quickly, or very slowly. Enter them before, after, and during the processing of some other event, or just as the time-out interval for this data item is about to expire.
- Now, what about all the error handling that you can trigger with "invalid" inputs?
  - Think about Whittaker & Jorgensen's constraint-focused attacks (Whittaker, *How Software Fails)*
  - Think about Jorgensen's hostile data stream attacks

 Copyright © 2003 Cem Kaner & James Bach

When people challenge extreme value tests…

“No user would do that.”

really means…

“No user I can *think of*, who I *like*, would do that *on purpose*.”

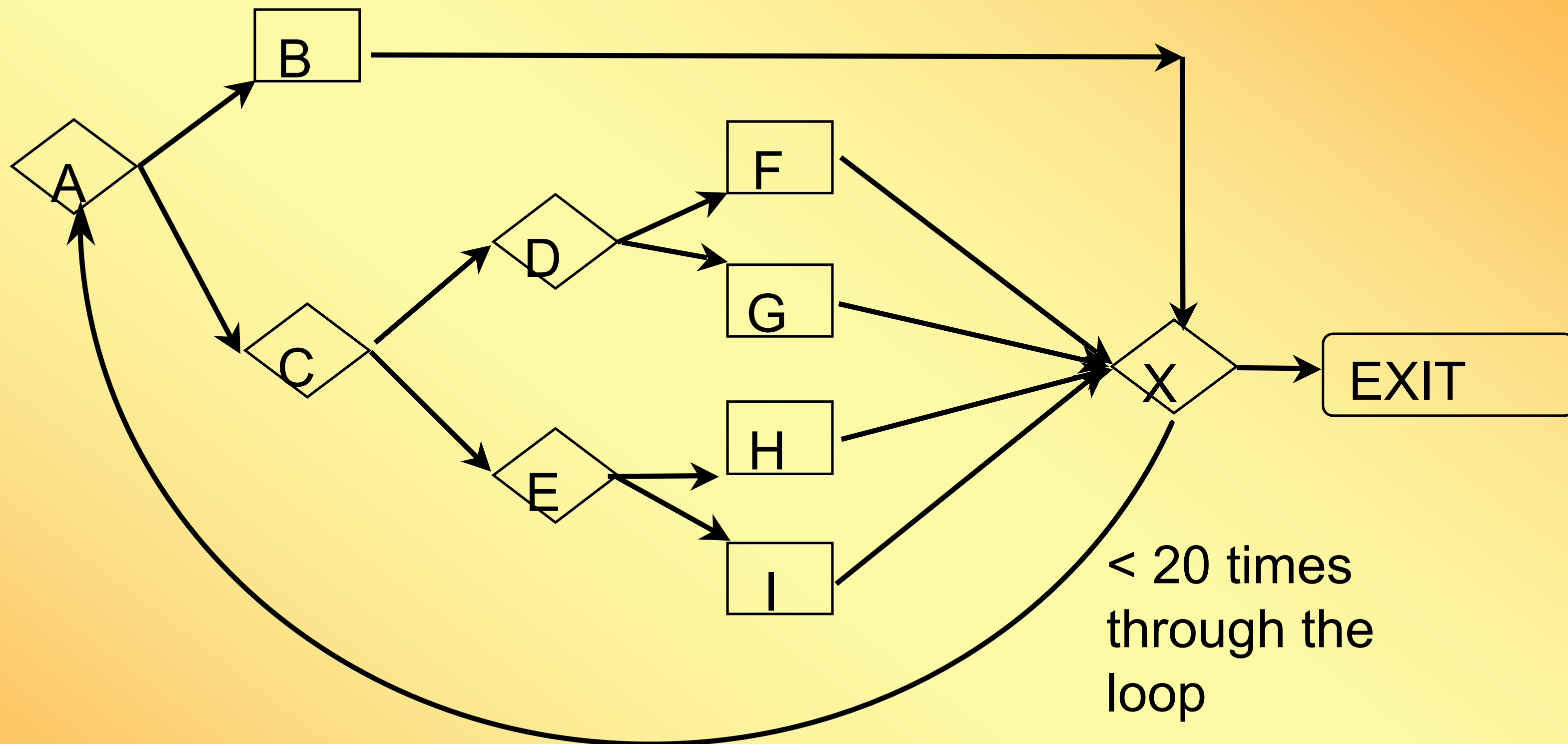
Who aren’t you thinking of?
Who don’t you like who might really use this product?
What might good users do by accident?

Copyright © 2003 Cem Kaner & James Bach

Combination testing

- Variables interact.
  - Example 1: a program crashed when attempting to print preview a high resolution (back then, 600x600 dpi) output on a high resolution screen. The option selections for printer resolution and screen resolution were interacting.
  - Example 2: a program fails when the sum of variables is too large.
- Suppose there are N variables.
  - Suppose the number of choices for the variables are V1, V2, through VN.
  - The total number of possible combinations is V1 x V2 x . . . x VN. This is huge.
  - For example, a field that accepts only {1, 2, 3} and another that accepts only {A, B, C} yields 9 cases, 1A, 1B, 1C, 2A, 2B, 2C, 3A, 3B, and 3C.
  - Combine two fields that accept one digit (0 to 9) each, yields 10 x 10 = 100 possible combinations.
  - There are 318,979,564,000 possible combinations of the first four moves in chess.

Sequences



Here's an example that shows that there are too many paths to test in even a fairly simple program. This is from Myers, *The Art of Software Testing.*

 Copyright © 2003 Cem Kaner & James Bach

Sequences

The program starts at A.

- From A it can go to B or C
  - From B it goes to X
  - From C it can go to D or E
    - From D it can go to F or G
      - From F or from G it goes to X
    - From E it can go to H or I
      - From H or from I it goes to X
        - From X the program can go to EXIT or back to A. It can go back to A no more than 19 times.

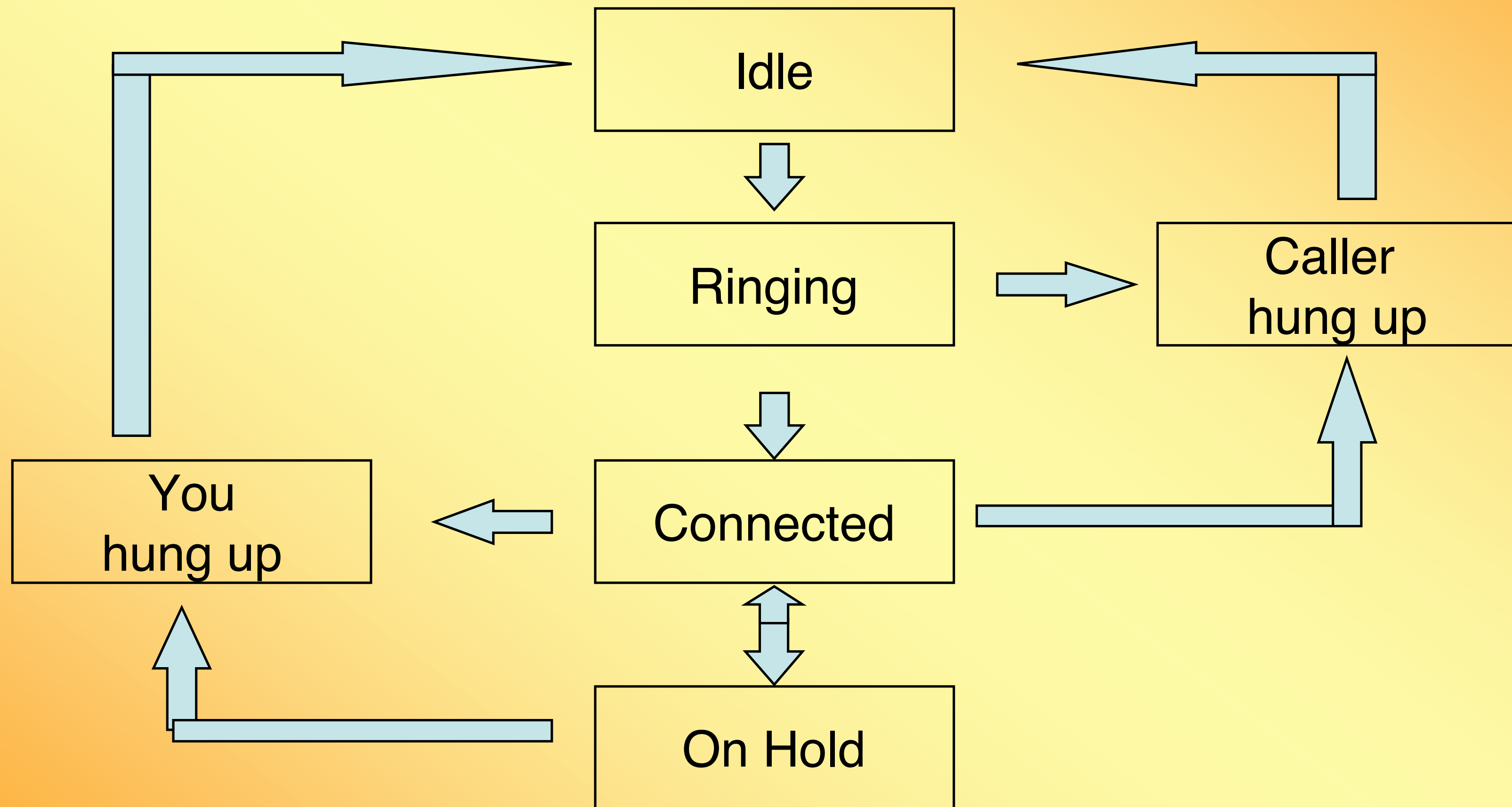
One path is ABX-Exit**. There are 5 ways to get to X and then to the EXIT in one pass.**

Another path is ABXACDFX-Exit**. There are 5 ways to get to X the first time, 5 more to get back to X the second time, so there are 5 x 5 = 25 cases like this**.

Sequences

- There are $5^1 + 5^2 + ... + 5^{19} + 5^{20} = 10^{14}$ = 100 trillion paths through the program.
- Testing would take approximately one billion years to try every path (if one could write, execute and verify a test case every five minutes).
  –

Sequences

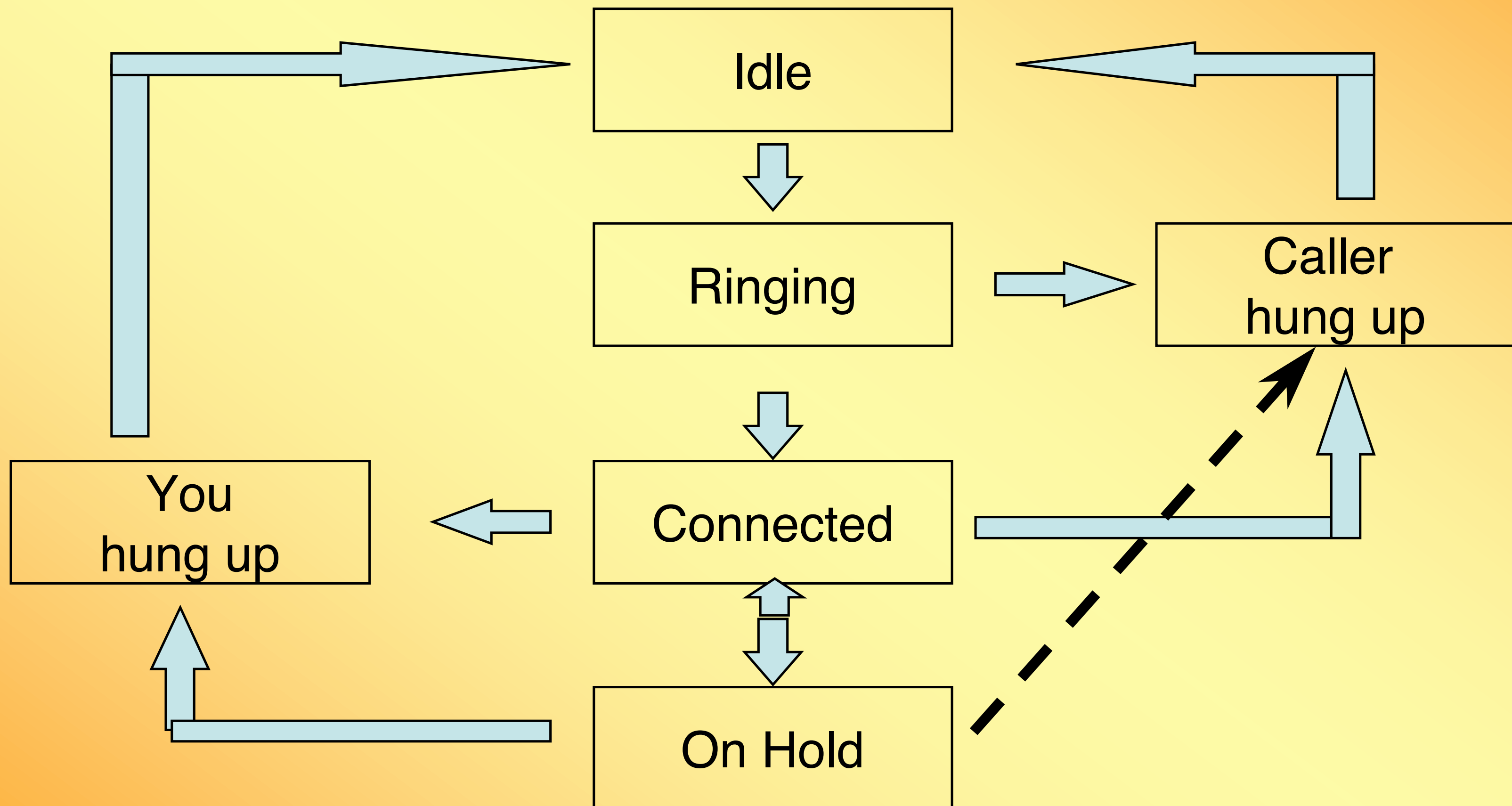


Sequences--the telephone example

- Why are we spending so much time on this crazy example?
- Because it illustrates several important points:
  - Simplistic approaches to path testing can miss critical defects.
  - Critical defects can arise under circumstances that appear (in a test lab) so specialized that you would never intentionally test for them.
  - When (in some future course or book) you hear a new methodology for combination testing or path testing, I want you to test it against this defect. If you have no suspicion that there is a stack corruption problem in this program, will the new method lead you to find this bug?

- This example also lays a foundation for our introduction to random/statistical testing.

Sequences



Conclusion

- Complete testing is impossible
  - *There is no simple answer for this.*
  - *There is no simple, easily automated, comprehensive oracle to deal with it.*
  - *Therefore testers live and breathe tradeoffs.*