9
Embedded Software Testing

18-649 Distributed Embedded Systems

Philip Koopman

September 22, 2014

Carnegie Mellon

Overview

- **Testing is an attempt to find bugs**
  - The reasons for finding bugs vary
  - Finding all bugs is impossible

- **Various types of testing for various situations**
  - Exploratory testing – guided by experience
  - White Box testing – guided by software structure
  - Black Box testing – guided by functional specifications

- **Various types of testing throughout development cycle**
  - Unit test
  - Subsystem test
  - System integration test
  - Regression test
  - Acceptance test
  - Beta test

Definition Of Testing

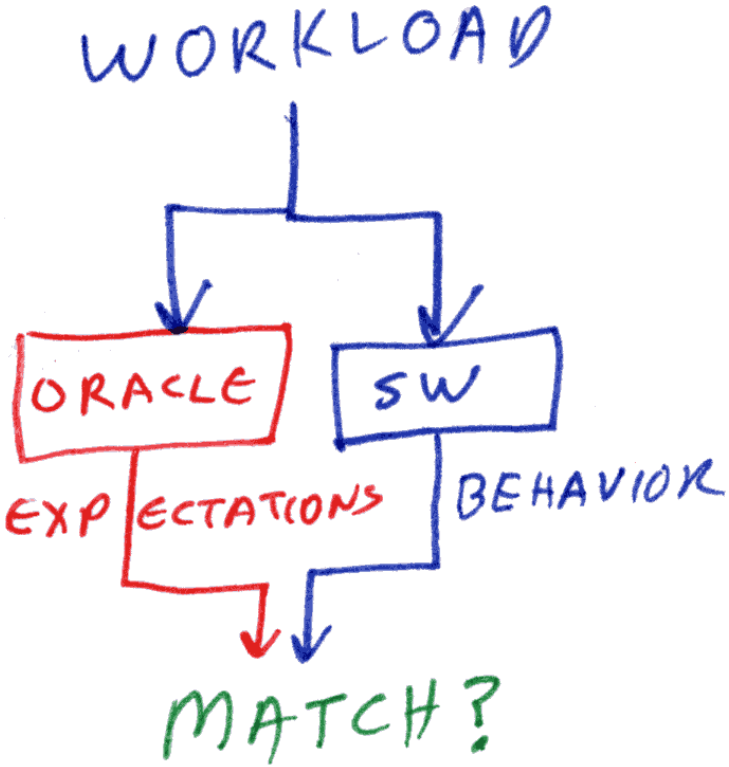
- **Testing is performing all of the following:**
  - Providing software with inputs (a “workload”)
  - Executing a piece of software
  - Monitoring software state and/or outputs for expected properties, such as:
    - Conformance to requirements
    - Preservation of invariants (e.g., never applies brakes and throttle together)
    - Match to expected output values
    - Lack of “surprises” such as system crashes or unspecified behaviors

- **General idea is attempting to find “bugs” by executing a program**

- **The following are potentially useful techniques, but are not *testing*:**
  - Model checking
  - Static analysis (lint; compiler error checking)
  - Design reviews of code
  - Traceability analysis

Testing Terminology

- **Workload:**
  - "Inputs applied to software under test"
  - Each test is performed with a specific workload
- **Behavior:**
  - "Observed outputs of software under test"
  - Sometimes special outputs are added to improve observability of software (e.g., "test points" added to see internal states)
- **Oracle:**
  - "A model that perfectly predicts correct behavior"
  - Matching behavior with oracle output tells if test passed or failed
  - Oracles come in many forms:
    - Human observer
    - Different version of same program
    - Hand-created script of predicted values based on workload
    - List of invariants that should hold true over behavior



What's A Bug?

- **Simplistic answer:**
  - A "bug" is a software defect = incorrect software
  - A software defect is an instance in which the software violates the specification

- **More realistic answer – a "bug" can be one or more of the following:**
  - Failure to provide required behavior
  - Providing an incorrect behavior
  - Providing an undocumented behavior or behavior that is not required
  - Failure to conform to a design constraint (e.g., timing, safety invariant)
  - Omission or defect in requirements/specification
  - Instance in which software performs as designed, but it's the "wrong" outcome
  - Any "reasonable" complaint from a customer
  - … other variations on this theme …

- **The goal of most testing is to attempt to find bugs in this expanded sense**

Types Of Testing – Outline For Following Slides

- **Testing styles:**
  - Smoke testing
  - Exploratory testing
  - Black box testing
  - White box testing

- **Testing situations:**
  - Unit test
  - Subsystem test
  - System integration test
  - Regression test
  - Acceptance test
  - Beta test

- **Most testing styles have some role to play in each testing situation**

Smoke Testing

- **Quick test to see if software is operational**
  - Idea comes from hardware realm – turn power on and see if smoke pours out
  - Generally simple and easy to administer
  - Makes no attempt or claim of completeness

  - Smoke test for car: turn on ignition and check:
    - Engine idles without stalling
    - Can put into forward gear and move 5 feet, then brake to a stop
    - Wheels turn left and right while stopped

- **Good for catching catastrophic errors**
  - Especially after a new build or major change
  - Exercises any built-in internal diagnosis mechanisms

- **But, not usually a thorough test**
  - More a check that many software components are “alive”

Exploratory Testing

- **A person exercises the system, looking for unexpected results**
  - Might or might not be using documented system behavior as a guide
  - Is especially looking for “strange” behaviors that are not specifically required nor prohibited by the requirements

- **Advantages**
  - An experienced, thoughtful tester can find many defects this way
  - Often, the defects found are ones that would have been missed by more rigid testing methods

- **Disadvantages**
  - Usually no documented measurement of coverage
  - Can leave big holes in coverage due to tester bias/blind spots
  - An inexperienced, non-thoughtful tester probably won’t find the important bugs

Black Box Testing

- **Tests designed with knowledge of behavior**
  - But without knowledge of implementation
  - Often called “functional” testing
- **Idea is to test what software does, but not how function is implemented**
  - Example: cruise control black box test
    - Test operation at various speeds
    - Test operation at various underspeed/overspeed amounts
    - BUT, no knowledge of whether lookup table or control equation is used
- **Advantages:**
  - Tests the final behavior of the software
  - Can be written independent of software design
    - Less likely to overlook same problems as design
  - Can be used to test different implementations with minimal changes
- **Disadvantages:**
  - Doesn’t necessarily know the boundary cases
    - For example, won’t know to exercise every lookup table entry
  - Can be difficult to cover all portions of software implementation

Examples of Black Box Testing

Assume you want to test a floating point square root function: sqrt(x)

- sqrt(0) = 0 (boundary condition)
- sqrt(1) = 1 (behavior changes between <1 and >1
- sqrt(9) = 3 (test some number greater than 1)
- sqrt(.25) = .5 (test some number less than 1)
- sqrt(-1) => error (test an out of range input)
- sqrt(FLT_MAX) => … (test maximum numeric range)
- sqrt(FLT_EPSILON) => … (test smallest positive number)
- sqrt(NaN) => NaN (test for Not a Number input values)
- Pick random positive numbers and confirm that: sqrt(x) * sqrt(x) = x

- Other types of possible results to monitor:
  - Monitor numerical accuracy/stability for floating point math
  - Check to see if software crashes on some inputs

White Box Testing

- **Tests designed with knowledge of software design**
  - Often called “structural” testing
- **Idea is to exercise software, knowing how it is designed**
  - Example: cruise control white box test
    - Test operation at every point in control loop lookup table
    - Tests that exercise both paths of every conditional branch statement
- **Advantages:**
  - Usually helps getting good coverage (tests are specifically designed for coverage)
  - Good for ensuring boundary cases and special cases get tested
- **Disadvantages:**
  - 100% coverage tests might not be good at assessing functionality for “surprise” behaviors and other testing goals
  - Tests based on design might miss bigger picture system problems
  - Tests need to be changed if implementation/algorithm changes
  - Hard to test code that isn’t there (missing functionality) with white box testing

Examples of White Box Testing

Assume you want to test a floating point square root function: sqrt(x)

- Uses lookup table spaced at every 0.1 between zero and 10
- Uses iterative algorithm at and above value of 10

- Test sqrt(x) for negative numbers (expect error)
- Test sqrt(x) for every value of x in middle of lookup table: 0.05, 0.15, … 9.95
- Test sqrt(x) exactly at every lookup table entry: 0, 0.1, 0.2, … 10.0
- Test sqrt(x) at 10.0 + FLT_EPSILON
- Test sqrt(x) for some numbers that exercise interpolation algorithm

◆ **Main differences from Black Box Testing:**

- Tests exploit knowledge of software design & coding details
- Usually strives for 100% coverage of known properties
  - e.g., lookup table entries
- Digs deepest at algorithmic discontinuities & branches
  - e.g., lookup table boundaries and center values

Testing Coverage

- **"Coverage" is a notion how completely testing has been done**
  - Usually a percentage (e.g., "97% branch coverage")
- **White box testing coverage (this is the usual use of the word "coverage"):**
  - Percent of conditional branches where both sides of branch have been tested
  - Percent of lookup table entries used in computations
- **Black box testing coverage:**
  - Percent of requirements tested
  - Percent of documented exceptions exercised by tests
  - But, must relate to externally visible behavior or environment, not code structure
- **Important note: 100% coverage is not "100% tested"**
  - Each coverage aspect is narrow; good coverage is necessary, but not sufficient to achieve good testing

What Can You Test For? (from [kaner])

- Conformance to specification
- Correctness
- Usability
- Boundary conditions
- Performance
- State transitions
- Mainstream usage (against scenarios)
- Load testing (events, memory usage, error rates)
- Error recovery
- Security
- Compatibility/configuration (equipment variations; old versions)
- Installability/serviceability

What Errors Do People Make? (from [kaner])

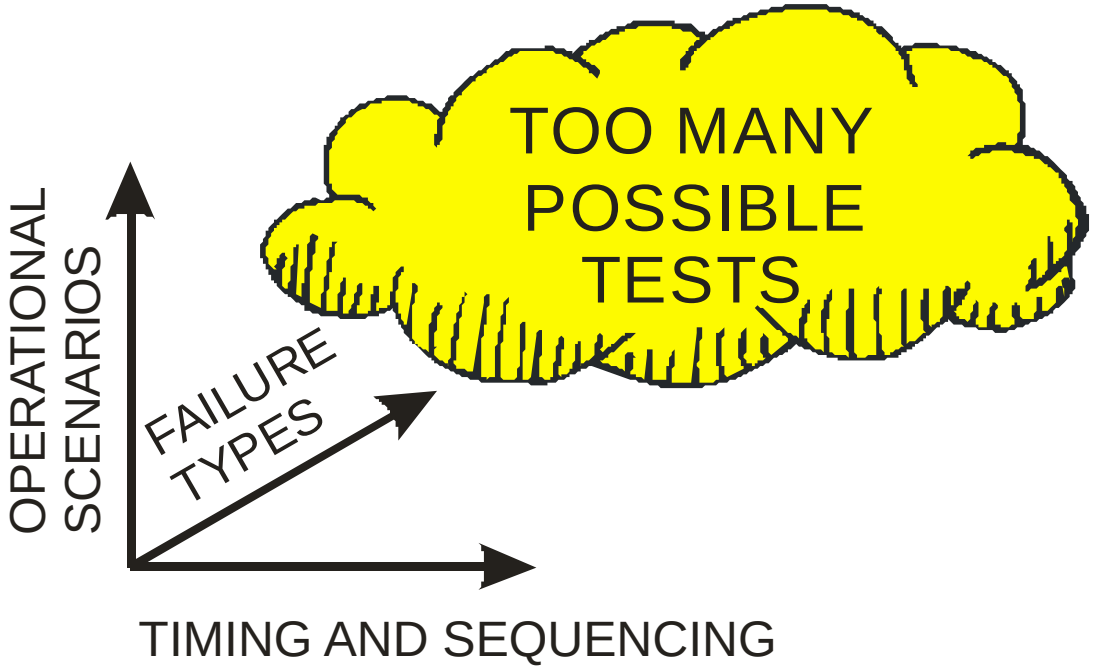
Sometimes it is a good idea to look for common programming errors

- **User interface**
- **Error handling**
  - Incorrectly handled errors
  - Missed error checks
- **Boundary (e.g., overflow/underflow)**
- **Calculation errors (wrong algorithm)**
- **Control flow**
- **Race conditions**
- **Load conditions**
- **Hardware error handling**
- **Version control errors**
- **Documentation**

- *Remember that test programs are programs too, and can have errors*

System-Level Testing Can't Be Complete

- **System level testing is useful and important**
  - Can find unexpected component interactions
- **But, it is impracticable to test everything at the system level**
  - Too many possible operating conditions, timing sequences, system states
  - Too many possible faults, which might be intermittent
    - Combinations of component failures + memory corruption patterns
    - Multiple software defects activated by a sequence of operations



Even Unit Testing Is Typically Incomplete

- **How many test cases to completely test the following :**

```
myfunction(int a, int b, int c)
```

- **Assume 32-bit integers:**
  - $2^{32} * 2^{32} * 2^{32} = 2^{96} = 7.92 * 10^{28}$ => at 1 billion tests/second
  - Testing all combinations takes 2,510,588,971,096 years
  - Takes longer for complete tests if there are any state variables inside function
  - Takes longer if there are environmental dependencies to test as well
- **This is a fundamental problem with testing – you can't test everything**
  - Therefore, need some notion of how much is "enough" testing
- **Even if you could test "everything," there is more to "everything" than just all possible input values**
  - Kaner has published a list of more than 100 types of coverage *http://www.kaner.com/coverage.htm*

excerpt from: *http://www.kaner.com/coverage.htm*

81. Recovery from every potential type of equipment failure. Full coverage includes each type of equipment, each driver, and each error state. For example, test the program's ability to recover from full disk errors on writable disks. Include floppies, hard drives, cartridge drives, optical drives, etc. Include the various connections to the drive, such as IDE, SCSI, MFM, parallel port, and serial connections, because these will probably involve different drivers.
82. Function equivalence. For each mathematical function, check the output against a known good implementation of the function in a different program. Complete coverage involves equivalence testing of all testable functions across all possible input values.
83. Zero handling. For each mathematical function, test when every input value, intermediate variable, or output variable is zero or near-zero. Look for severe rounding errors or divide-by-zero errors.
84. Accuracy of every graph, across the full range of graphable values. Include values that force shifts in the scale.
85. Accuracy of every report. Look at the correctness of every value, the formatting of every page, and the correctness of the selection of records used in each report.
86. Accuracy of every message.
87. Accuracy of every screen.
88. Accuracy of every word and illustration in the manual.
89. Accuracy of every fact or statement in every data file provided with the product.
90. Accuracy of every word and illustration in the on-line help.
91. Every jump, search term, or other means of navigation through the on-line help.
92. Check for every type of virus / worm that could ship with the program.
93. Every possible kind of security violation of the program, or of the system while using the program.
94. Check for copyright permissions for every statement, picture, sound clip, or other creation provided …
95. Verification of the program against every program requirement and published specification.
96. Verification of the program against user scenarios. Use the program to do real tasks that are challenging and well-specified. For example, create key reports, pictures, page layouts, or other documents events to match ones that have been featured by competitive programs as interesting output or applications.
97. Verification against every regulation (IRS, SEC, FDA, etc.) that applies to the data or procedures of the program.

"White Box" Testing – Based on Structure

- **Look at program structure & attempt to cover various aspects with tests**
  - Traceability is to design & implementation

- **Types of coverage:**
  - All paths & states in a statechart
  - All statements
  - All branches
  - All data dependencies
  - All inputs
  - All exceptions/error conditions
  - …

- **Coverage doesn't mean you necessarily did the "best" tests**
  - But, it gives some confidence that you didn't completely miss an area of code in the testing

"Black Box" Testing – Based On Functionality

- **Look at specification & attempt coverage**
  - Traceability is to specification

- **Types of coverage:**
  - All numbered requirements
  - All scenarios
  - Standard bag of tricks:
    - Boundary testing
    - Special value testing (e.g., string terminator characters)
  - All implicit requirements
    - Doesn't crash
    - Reports reasonable error codes
  - All implementation techniques for approaches
    - E.g., a trig. Function could be:
      - » Table lookup + interpolation
      - » Polynomial approximation
      - » Computed by hardware coprocessor

Which Is Better – Black Box or White Box?

- **Both types of test have their place (of course!)**

- **White box best for ensuring details are correct in implementation**
  - Tests are designed to ensure that code conforms to design
  - Exercises different nooks and crannies of code in a way black box usually can't
  - Best way to ensure good coverage metrics

- **Black box best for requirements-based testing**
  - Emphasis on meeting requirements and, in general, overall behavior
  - Tests are designed to ensure that the software actually works!

- **Smoke tests – good for quick checks, but not for rigorous evidence of correctness**

- **Exploratory tests – can be helpful in finding bugs before users find them, but does not assure coverage**

Why Do We Test?

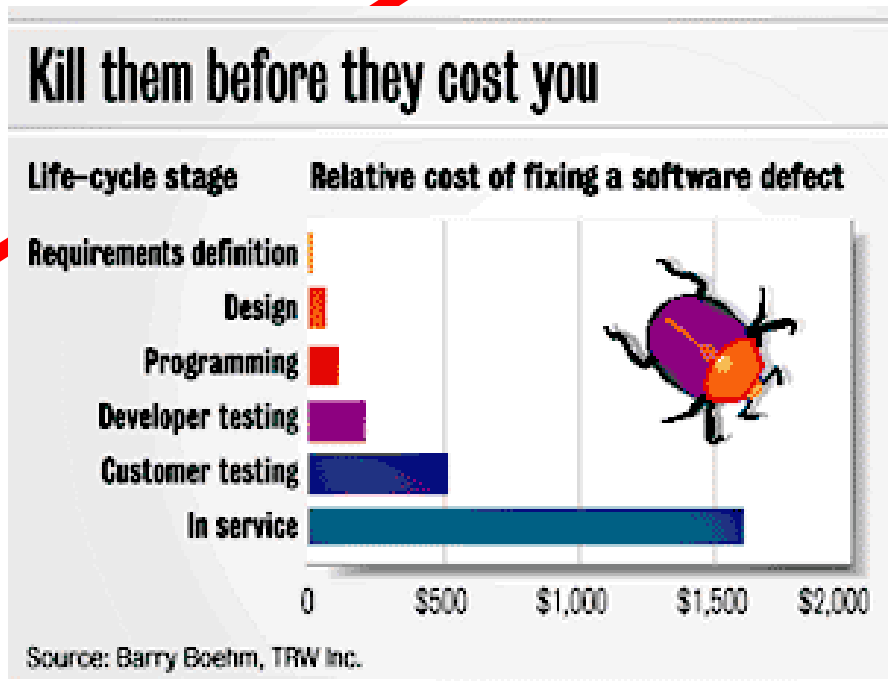
- **Because someone makes us test**
  - This is an unsatisfactory reason
  - Unlikely to be productive because it is a bureaucratic imposition
  - BUT, can happen for certification (e.g., “must have 100% branch coverage”)

- **Want to find bugs in program so they can be removed**
  - Tests are designed to ensure that important operations work properly
  - Usually, approach is to test until we stop finding bugs
  - When “important” bugs are fixed, product is shipped
  - Most often, this is how desktop software testing works

- **Want to test the hypothesis that there are no (important) bugs**
  - Tests are never supposed to find an (important) defect
  - Keep testing until it seems unlikely any bugs are really there
  - If a bug is found this indicates a software development process failure!
  - In general, this is the goal of testing for safety critical systems

What's The Cost Of A Finding & Fixing A Defect?

log(cost)



Product Lifecycle Phase

Notes:

- Cost includes both money and lost time to market
- Higher costs late in lifecycle are when profits are low, making it worse

When Do We Test?

- **Different phases of development cycle**
  - Unit test – development
  - Subsystem test – software module integration
  - System test – system integration
  - Acceptance test – product shipment
  - Beta test – selected customer use of system

- **Different people play roles of tester:**
  - Programmer often does own testing for unit test
  - Independent testers are often involved in subsystem, system & acceptance tests
  - Customers are often involved in acceptance & beta tests

Unit Test

- **A "unit" is a few lines of code to a small program**
  - Usually created by a single developer
  - Usually tested by the programmer who created it

- **Purpose of unit test:**
  - Try to find all the "obvious" defects
  - Can be done before and/or after code review

- **Approaches** (mostly exploratory & white box)
  - Exploratory testing makes a lot of sense
    - Helps programmer build intuition and understand code
  - White box testing to ensure all portions of code exercised
    - Often useful to ensure 100% arc and state coverage for statecharts
  - Some black box testing as a "sanity check"

Subsystem Test

- **A "subsystem" is a relatively complete software component (e.g., engine controller software)**
  - Usually created by a team of developers
  - Usually tested by a combination of programmers and independent testers

- **Purpose of subsystem test:**
  - Try to find all the "obvious" defects
  - Can be done before and/or after code review

- **Approaches** (mostly white box; some black box)
  - White box testing is key to ensuring good coverage
  - Black box testing should at a minimum check interface behaviors against specification to avoid system integration surprises

  - Exploratory testing can be helpful, but shouldn't find a lot of problems
  - Smoke test is helpful in making sure a change in one part of subsystem doesn't completely break the entire subsystem

System Integration Test

- **A "system" is a complete multi-component system (e.g., a car)**
  - Often created by multiple teams organized in different groups
  - Usually tested by independent test organizations

- **Purpose of system integration test:**
  - Assume that components are mostly correct; ensure system behaves correctly
  - Find problems/holes/gaps in interface specifications that cause system problems
  - Find unexpected behavior in system

- **Approaches** (mostly black box)
  - Tends to be mostly black box testing to ensure system meets requirements
  - Want white box techniques to ensure all aspects of interfaces are tested
  - Exploratory testing can help look for strange component interactions
  - Smoke tests are often used to find version mismatches and other problems in independent components

Regression Test

(A) Regression tests ensure that a fixed bug stays fixed

- Often based on test that was used to reproduce the bug before the fix

And/or

(B) Regression tests ensure functionality not broken after a change

- Often a subset of unit and integration tests (e.g., nightly build & test cycle)

- **Purpose of regression testing:**
  - We made a change to the software; did we break anything?
  - In the case of iterated development, ensure there is a working system at the end of every periodic change/build/test cycle
  - It is a cheaper-to-run test than an acceptance test

- **Approaches**
  - Usually a combination of all types of tests that are sized to get decent coverage and reasonably fast execution speed.
  - Often concentrates on areas in which bugs have previously been found

Acceptance Test

- **Acceptance tests ensure system provides all advertised functions**
  - Testing performed by a customer or surrogate
  - Might also involve a certification authority or independent test observer

- **Purpose of acceptance test:**
  - Does the system meet all requirements to be used by a customer?
  - Usually the last checkpoint before shipping a system
  - Might be performed on all systems to check for hardware defects/manufacturing defects, not just software design problems
  - In a mature software process, it is a quality check on the entire process
    - OUGHT to find few or no significant bugs if software has high quality
    - If bugs are found, it means you have a *quality problem*

- **Approaches**
  - Usually black box testing of system vs. 100% of high level product requirements

Beta Test

- **A "beta version" is complete software that is "close" to done**
  - Theoretically all defects have been corrected or are at least known to developers
  - Idea is to give it to sample users and see if a huge problem emerges
- **Purpose of beta test:**
  - See if software is good enough for a friendly, small user community (limit risk)
  - Find out if "representative" users are likely to stimulate failure modes missed by testing
- **Approaches**
  - This is almost all exploratory testing
    - Assumption is that different users have different usage profiles
    - Hope is that if small user community doesn't find problems, there won't be many important problems that slip through into full production
- **Defect in beta test means you have a significant hole somewhere**
  - If you have a good process, it is likely a requirements issue
  - If it is "just a bug" then why did you miss it in reviews & testing???

How Many Testers For Each Developer?

- **Tester to Developer ratio varies depending on situation**

Web development:	**1 tester**	**per**	**5-10 developers**
Microsoft:	**1 tester**	**per**	**1 developer**
Safety critical software:	**up to 4-5 testers**	**per**	**1 developer**

  - In my interpretation, this is "test hours" to "developer hours."
    That means time spent on peer review & unit test counts as "test" effort even if it is done by developers

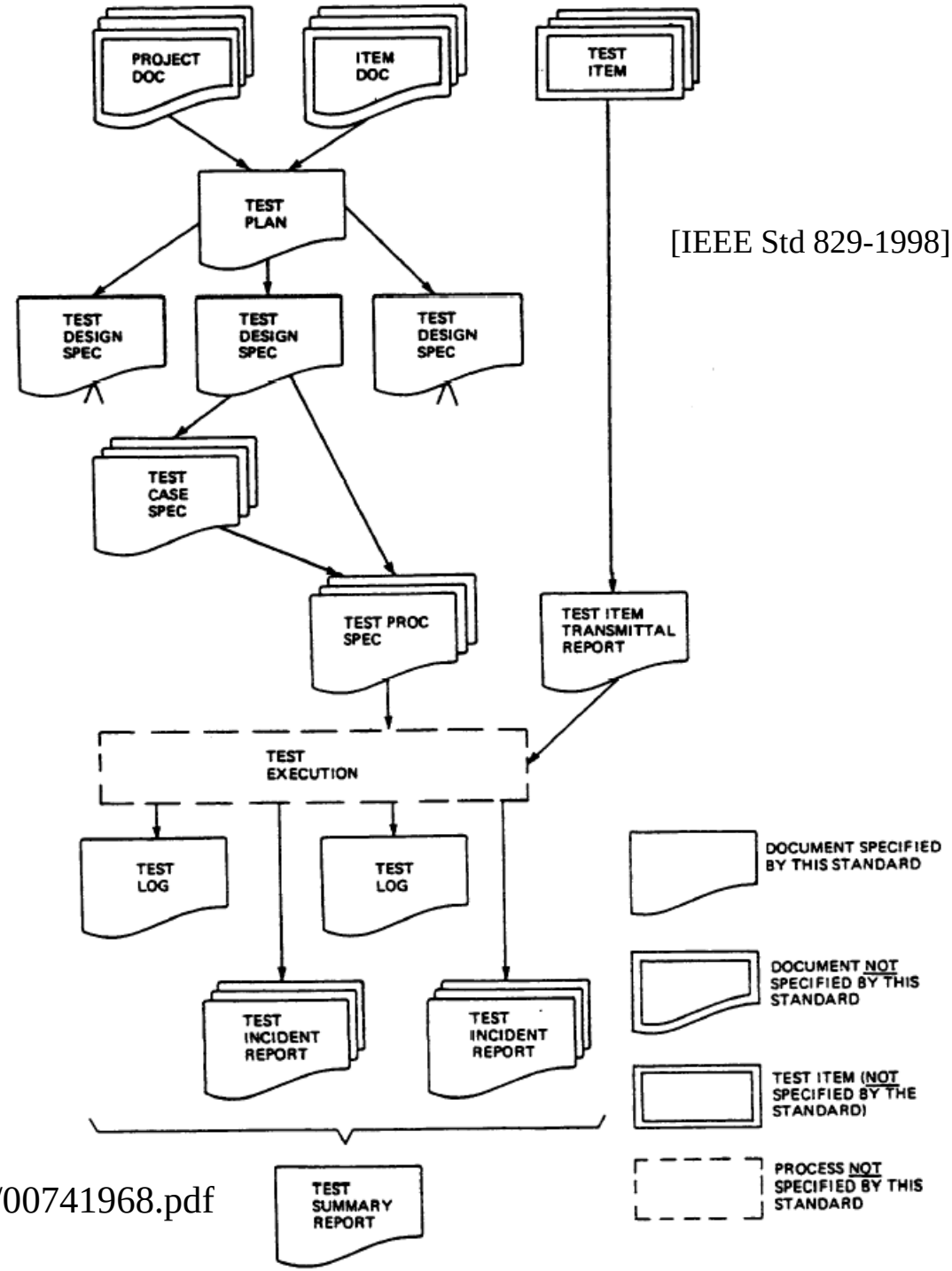
- **This is in addition to:**
  - External acceptance testing (customer performs test)
  - Beta test

- **How much does testing cost?**
  - Total validation & verification (including testing) is often 50% of software cost
  - For "good" embedded systems, total test cost is about 60% of project
    - This includes time that developers spend on unit test & peer reviews
  - In critical systems, it can be 80% of total software cost!
    - (Take a look at the ratios above; this checks with them.)

IEEE Std 829-1998

- **IEEE Std 829-1998**
  IEEE standard for software test documentation

- **Standard for all the pieces of a test plan**
  - Not embedded specific; but the best available standard
  - Typical of IEEE processes in requiring lots of paper
  - BUT, if there no paper, then the test can't be used in a safety case!



[IEEE Std 829-1998]

http://ieeexplore.ieee.org/iel4/5976/16010/00741968.pdf

Software Defect Issues

- **Pesticide paradox**
  - Tests will get rid of the bugs you test for, but not other bugs
  - Thus, a program that passes 100% of automated tests is NOT bug-free!!!

- **Fixing a bug causes new bugs**
  - Fixing 1 bug will cause X other bugs
    - (X is “fault reinjection rate”)
  - Fixing bugs is more expensive than writing new code, so
    - If X is medium to high (say, 0.2 to 1.0), then may be cheaper to write new code
    - If X is greater than 1, it is probably too risky to fix any more bugs

- **Bugs tend to congregate**
  - The more bugs you find in a piece of software, the more bugs are left
  - It is common that some pieces of code are “bug farms” due to:
    - Overly complex requirement for one piece of software
    - Poor quality implementation

Why Do We Test – Revisited

1. Testing to find bugs

2. Testing to demonstrate correctness

- **Implications of testing to find bugs in practice**
  - This is really the "fly-fix-fly" philosophy used historically in aviation
    - Fly around until a plane crashes
    - Fixes whatever caused a crash so the next plane won't break that way
    - Fly around some more
    - Eventually improves quality, but never achieves perfection
  - We hope that only a small percentage of bugs slip past each stage
  - Important to base tests on "things that matter" to find "bugs that matter"
  - Finding & fixing "all" bugs in practice never ends
    - Eventually must decide to ship with some known & unknown defects
    - "Software reliability" theory comes into play ("how much do we test" slide later)
  - Important approach is to ensure all specified functions work
    - Doesn't prove they work in all cases, but at least gives some confidence in result

Why Do We Test – Revisited

3. Testing to demonstrate software quality

- **Hypothesis – "there are no defects in this software"**
  - Test approach: attempt to find defects
  - Can disprove hypothesis by finding defects
  - Can never prove hypothesis, because complete testing is impossible

- **This is more of a manufacturing QA philosophy**
  - Detect errors in the "manufacturing" process (creating software)
    - Detection done through defects in created items
    - ***BUT***, the issue isn't the item being out of specification; it is that the process is broken
  - Fix process errors

How Much Do We Test?

- **If you believe in statistical independence of bugs, can use sampling theory**
  - But, bugs aren't statistically independent!
  - For example, a piece of software with many discovered bugs usually has many undiscovered bugs still left (bugs tend to cluster)

- **Usual answers are:**
  - Pretend statistical independence, apply some math, decide when to stop
  - Keep testing until you stop finding bugs
    - Sometimes you don't stop finding them – then just throw software away as "bad"
  - Keep testing until it is time to ship the product
  - Keep testing until testing budget is expended
  - Keep testing until all essential functions work in most common scenarios
  - Skip testing, since it's hopeless anyway
    - (We didn't say these are good reasons! Just common reasons)

Problems With Testing-Only Approach

- **Unfortunate reality:**
  - In general, it is impossible to prove a system is safe by testing alone!
    - It is impossible to get 100% testing coverage on every possible testing metric
  - That is why we have other lectures on V&V and ultra-dependability

- **So, how long does it take to find everything via testing?**
  - One third of all software faults take more than 5000 execution-years to manifest
    - 5000 years = 43,830,000 hours = $4.38 * 10^7$ hours
    - *Adams, N.E., "Optimizing preventive service of software product," IBM Journal of Research and Development, 28(1), p. 2-14, 1984. (Table 2, pg. 9, 60 kmonth column)*
    - This is field experience data based on IBM software services run on IBM mainframes, not testing data.
  - So if you test for 5000 years, you'll only find about two-thirds of the defects
    - Do you plan to test that long?
  - What this really means is you won't find these faults until you deploy

- **This is why reviews and inspections are so important!**

Testing Resources

- **Best starting point for more info:**

  http://www.testingfaqs.org/

- **Most useful books:**

  - Beizer, Boris. *Black Box Testing*. New York: John Wiley & Sons, 1995, ISBN 0-471-120904-4.
  - Kaner, Cem, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*, Second Edition. Boston: International Thomson Computer Press, 1993. ISBN 1-85032-847-1. John Wiley & Sons, Inc., 1999. ISBN 0-471-35846-0.

Issue Tracking

- **An "issue" is a bug, defect, feature request, complaint, etc.**
  - Anything that is worth recording and tracking can be an issue
  - This includes process issues (e.g., defect in peer review recording sheet design)

- **Why do we need Bugzilla? (or similar tool)**
  - Easy to loose track of all the outstanding issues, especially if scattered around
  - Takes effort to prioritize issues; don't want to re-do all that work continually
  - Lets us do data mining for patterns and trends
    - Bug rates increasing/decreasing
    - Look for bug farms (places where bugs congregate)

- **Most essential aspect of defect tracking is priority**
  - Not "severity"
    - If a bug causes a system crash but nobody cares, it may be low priority
    - A very minor bug that annoys all users all the time may have high priority
  - Priority is "how important is this to fix"
  - Look at the "PHA" (Preliminary Hazard Analysis) technique for safety critical systems for a good strategy to determine priority based on severity+frequency

How Does This Map To The Project?

- **Unit test/Subsystem test**
  - Test to ensure statecharts are implemented correctly
  - We have very simple subsystems, so these two types of testing are combined
  - We emphasize white box testing
    - Best approach is to create tests looking at sequence diagrams
    - Then determine state and arc coverage to attain 100% coverage

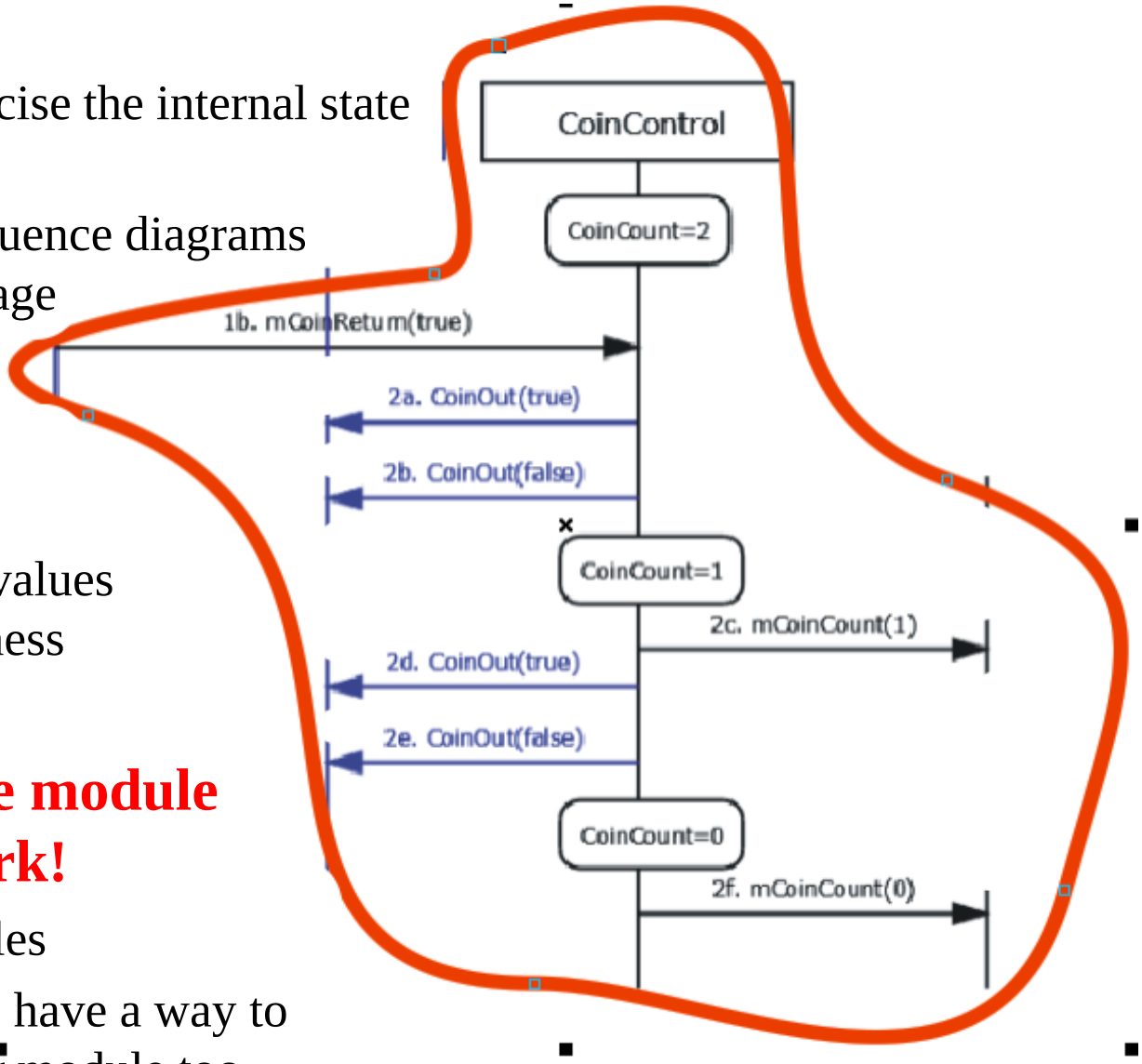
- **Integration test**
  - Test to ensure that sequence diagrams are followed properly
  - We emphasize white box testing at the arc level
    - Best approach is to create one or more test cases for each SD
    - Determine state and arc coverage against those SDs
  - Use instrumentation to check against behavioral requirements

- **Acceptance test**
  - Test to see if passengers get delivered
  - Throw different profiles and randomized passenger models at the system
  - Use instrumentation to check against high level elevator requirements

Idea Behind Unit Test

- **Isolate single module and feed it direct inputs**
  - Feed in inputs that exercise the internal state machine
  - Base tests on single sequence diagrams and on statechart coverage

Test Input

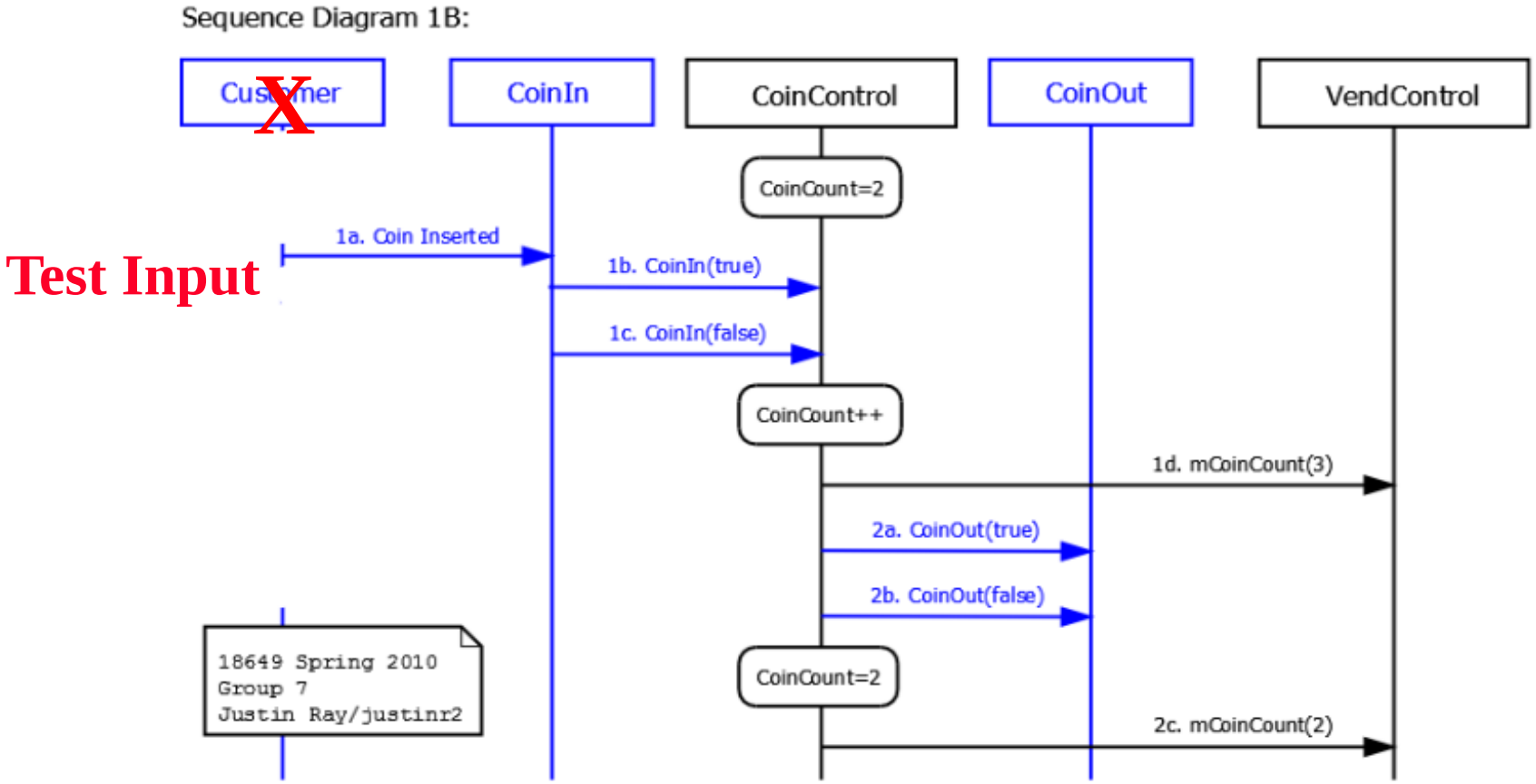


  - Monitor state machine values and outputs for correctness

- **ONLY LOAD a single module into the test framework!**
  - Plus test/monitor modules
  - Might be a good idea to have a way to set internal state in your module too

Idea Behind Integration Test

- **Run all modules in a Sequence Diagram except selected inputs**
  - Artificially set up state information to meet preconditions
  - Feed primary inputs from test harness; let rest of arcs run on their own
  - Make sure other arcs perform as expected



Acceptance Test

- **Ensure system as a whole actually meets requirements**
  - In simple systems, testing all scenarios suffices
  - In real systems, need to test sequences of Use Cases
- **In our system we have simulated passengers**
  - So make sure passenger workload covers all reasonable usages
- **If you don't have a simulated workload:**
  - Go through the high level system requirements and get coverage
  - Go through all the use cases and get coverage
  - Go through all the high level scenarios and get coverage
- **Traditionally this is called a "Factory Acceptance Test"**
  - The factory (manufacturer) accepts it as a viable product to sell
  - Traceability is to functional requirements for entire system

Review

- **Testing is supposed to find bugs**
  - The reasons for finding bugs vary
  - Finding all bugs is impossible

- **Various types of testing for various situations**
  - Exploratory testing – guided by experience
  - White Box testing – guided by software structure
  - Black Box testing – guided by functional specifications

- **Various types of testing throughout development cycle**
  - Unit test
  - Subsystem test
  - System integration test
  - Regression test
  - Acceptance test
  - Beta test