When to Stop Testing:

Dependability & Reliability

CSCE 740 - Lecture 25 - 11/30/2017

When is software ready for release?

We Will Cover

- How do we know when we are done?
- Stopping Criteria
  - Requirements
  - Coverage
  - Budget
  - Plan
  - Dependability

When All Functional Tests Pass?

- Write tests based on your requirements, then stop when they pass.
  - AKA: Stop when you can make an argument for verification.
- Are there problems with this?

When We Have Achieved Coverage?

- Set your sights on some structural coverage metric and test until that is achieved.
    - branch coverage, condition coverage, etc.
- Problems?

When We Run Out of Time?

- The “Budget Coverage Criterion”
  - The usual answer to when testing is done.
  - When we run out of time.
  - When the money dries up.
- Problems?

What if We Make a Plan?

- Plan a series of tests carefully, then test according to that plan.
  - Consider forms of functional and structural testing.
  - Factor in the budget and the cost of test case creation in choosing how we test.
- When those tests are done, you are done.
- Problems?

$%$*, I give up. When are we done?

- Can we can argue that we’ve done enough?
- Provide evidence that the system is *dependable*.
- The goal is to establish four things about the system:
  - That it is **correct**.
  - That it is **reliable**.
  - That it is **safe**.
  - That is is **robust**.

Correctness

- A program is **correct** if it is consistent with its specifications.
    - A program cannot be 30% correct. It is either correct or not correct.
    - A program can easily be shown to be correct with respect to a bad specification. However, it is often impossible to prove correctness with a good, detailed specification.
    - Correctness is a goal to aim for, but is rarely provably achieved.

Reliability

- A statistical approximation of correctness.
- Reliability is a measure of the likelihood of correct behavior from some period of observed behavior.
  - Time period, number of system executions
  - Measured relative to a specification and a usage profile (expected pattern of interaction).
    - Reliability is dependent on how the system is interacted with by a user.

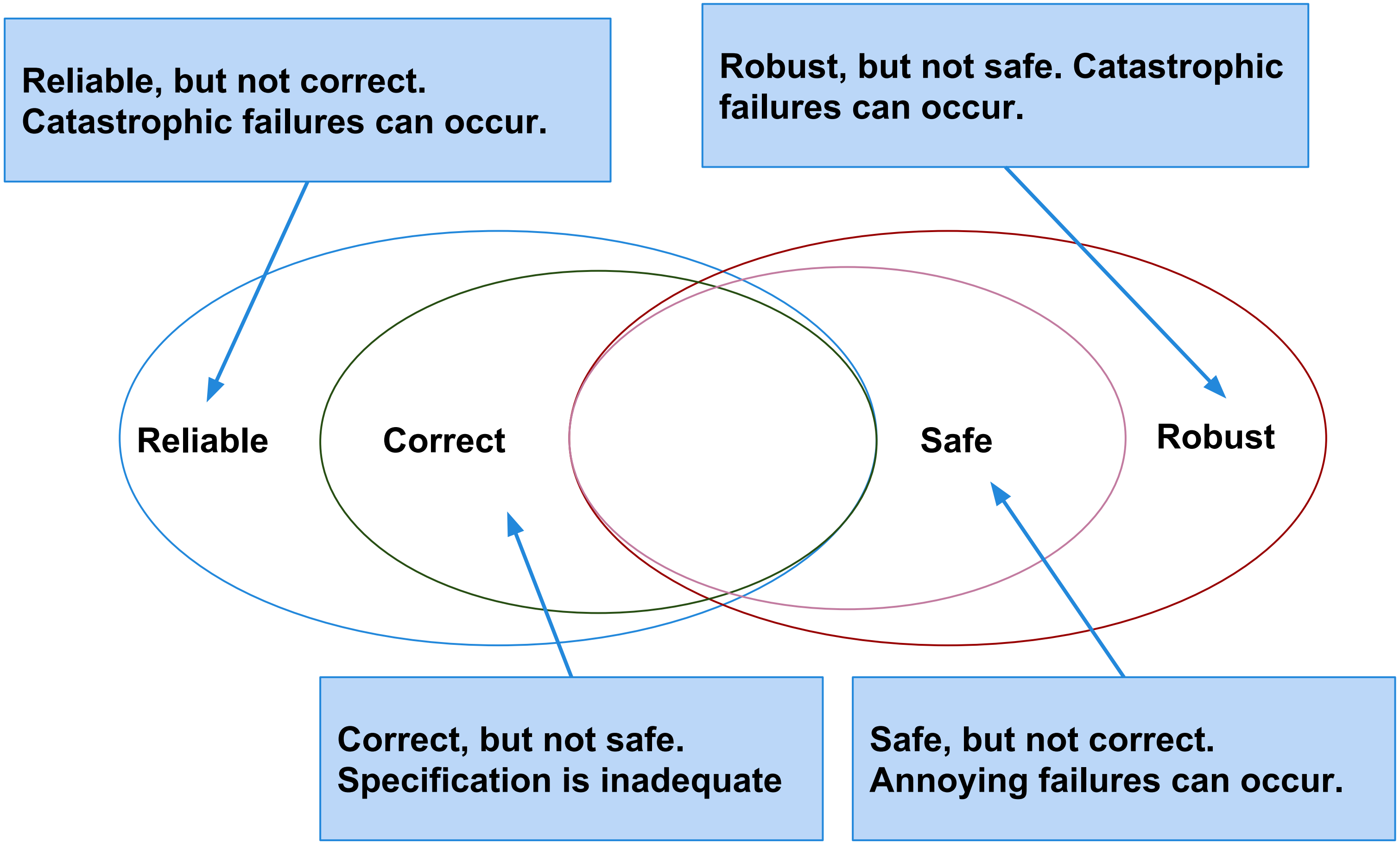
Safety

- Two flaws with correctness/reliability:
    - Success is relative to the strength of the specification.
    - Severity of a failure is not considered. Some failures are worse than others.
- **Safety** is the ability of the software to avoid *hazards*.
    - Hazard = any undesirable situation.
    - Relies on a specification of hazards.
        - But is only concerned with avoiding hazards, not other aspects of correctness.

Robustness

- Correctness and reliability are contingent on normal operating conditions.
- Software that is “correct” may still fail when the assumptions of its design are violated. *How* it fails matters.
- Software that “gracefully” fails is **robust**.
  - Consider events that could cause system failure.
  - Decide on an appropriate counter-measure to ensure graceful degradation of services.

Dependability Property Relations



Reliable, but not correct. Catastrophic failures can occur.

Robust, but not safe. Catastrophic failures can occur.

Reliable

Correct

Safe

Robust

Correct, but not safe. Specification is inadequate

Safe, but not correct. Annoying failures can occur.

Measuring Dependability

- Finding all faults is nearly impossible, and always expensive.
- We can *always* test more.
- Must establish criteria for when the system is dependable *enough* to release.
  - Correctness hard to prove conclusively.
  - Robustness/Safety important, but not enough.
  - Reliability is the basis for arguing dependability.

Analyzing Software Reliability

What is Reliability?

- Reliability is the probability of failure-free operation for a specified time in a specified environment for a given purpose.
- This means different things depending on the system and the users of that system.
- Informally, reliability is a measure of how well users think the system provides the services they require.

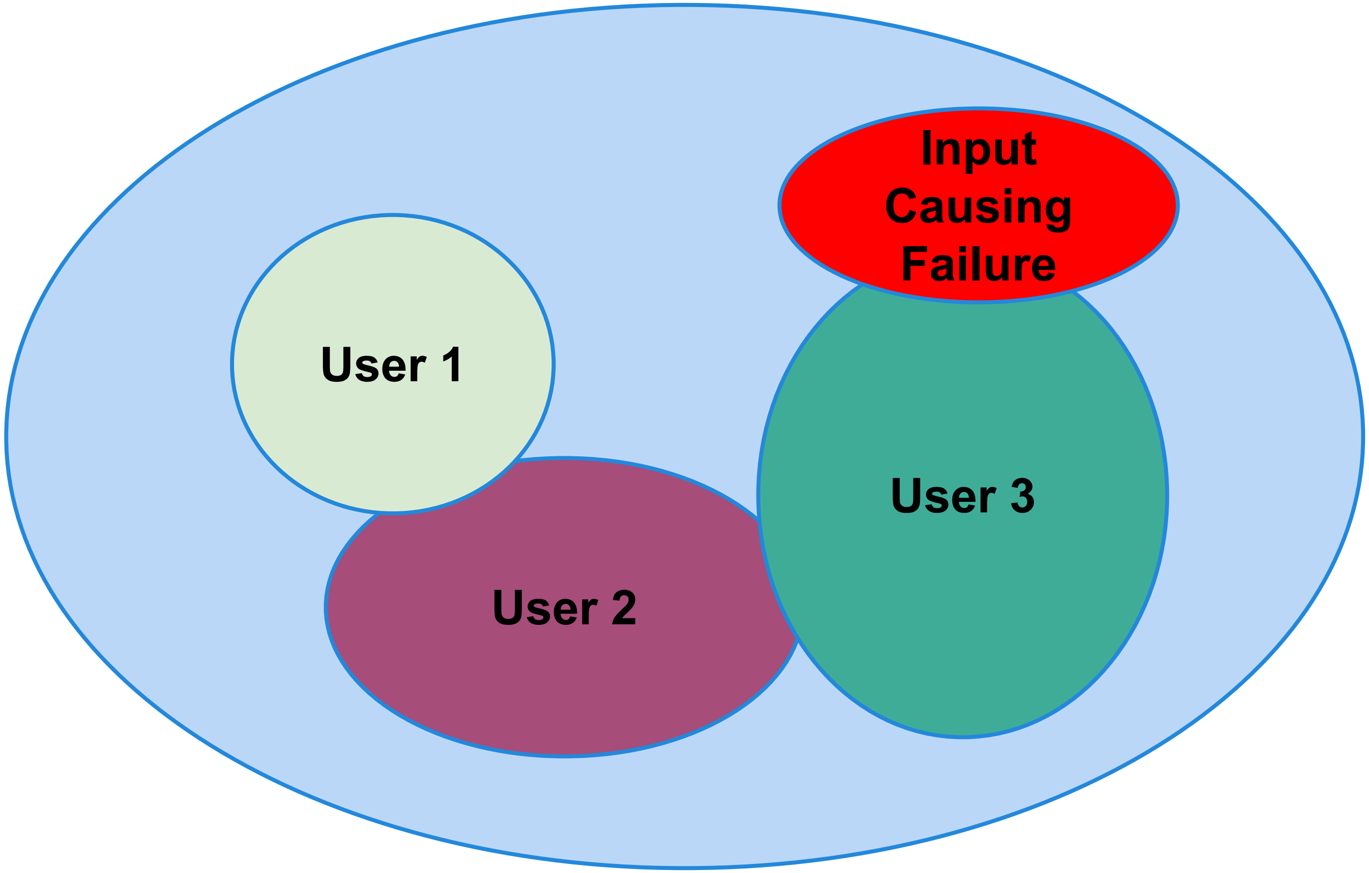
Reliability is Measurable

- Reliability can be defined and measured.
- Reliability requirements can be specified:
  - Non-functional requirements can define the number of failures that are acceptable during normal use of the system, or the time in which the system is allowed to be unavailable for use.
  - Functional requirements can define how the software avoids, detects, and tolerates faults to ensure they don’t lead to failures.

Improving Reliability

- Reliability is improved when software faults that occur in the most frequently-used parts of the software are removed.
    - Removing X% of the faults will not necessarily lead to an X% improvement in reliability.
        - In a study, removing 60% of the faults actually led to a 3% reliability improvement.
- Removing faults with serious consequences is the top priority.

Reliability Perception



Software Reliability

- Reliability cannot be defined objectively for all situations.
    - Reliability measurements quoted out of context are meaningless.
- Requires operational profile for its definition.
    - A profile of the expected pattern of software usage.
- Must consider fault consequences.
    - Not all faults are equally serious.
    - System is perceived as unreliable if there are more serious faults.

How to Measure Reliability

- Measuring reliability is normal when building hardware, but hardware metrics often aren’t suitable for software.
    - Based on component failures and the need to repair or replace a component once it has failed.
    - In hardware, the design is assumed to be correct.
- Software failures are always design failures.
    - Often, the system is available even though a failure has occurred.

Availability

- The availability of a system reflects its ability to deliver services when available (uptime/total time).
  - Takes repair and restart time into account.
  - Does not tend to take incorrect computations (partial failures) into account.
- Availability of 0.9999 means the system is available 99.99% of the time.
  - 0.9 = down for 144 minutes a day, 0.99 = down for 14.4 minutes, 0.999 = down for 84 seconds, 0.9999 = down for 8.4 seconds.

Probability of Failure on Demand (POFOD)

- The likelihood that a service request will result in a system failure (failures/requests over a period).
- POFOD = 0.001 means that 1 out of 1000 service requests result in a failure.
- Should be used as a reliability metric in situations where a failure on request is serious.
  - Independent of the frequency of requests.
  - 1/1000 failure rate sounds risky, but if one failure per lifetime, it is good.

Rate of Occurrence of Fault (ROCOF)

- Frequency of the occurrence of unexpected behavior.
  - Probable number of failures over a period of time or number of system executions.
- ROCOF of 0.02 means that 2 failures are likely per 100 time units.
- Most appropriate metric when requests are made on a regular basis (such as a shop).

Mean Time Between Failures (MTBF)

- Measures the average length of time between observed failures.
  - Requires the timestamp of each failure and the timestamp of when the system resumed service.
- MTBF of 500 means that the time between failures is, on average, 500 time units (or requests).
- For systems with long user sessions, you want to require a long MTBF.

Data Needed for Measurements

To assess reliability, data must be captured from users’ sessions with the system:

- Measure the number of failures per a given number of requests (used for POFOD).
- Measure the number of failures, plus total elapsed time or request number (used for ROCOF).
- Requires the timestamp of each failure and the timestamp of when service is resumed (used for MTBF).
- Measure the time to restart after a failure (for availability).

Reliability Examples

- Provide software with 10000 requests.
  - Wrong result on 35 requests, crash on 5 requests.
  - What is the POFOD?
- 40 / 10000 = 0.0004
- Run the software for 144 hours
  - (6 million requests). Software failed on 6 requests.
  - What is the ROCOF? The POFOD?
- ROCOF = 6/144 = 1/24 = 0.04
- POFOD = 6/6000000 = (10^{-6})

Reliability Examples

- You advertise a piece of software with a ROCOF of 0.001 failures per hour.
  - However, it takes 3 hours (on average) to get the system up again after a failure.
  - What is the availability per year?
- Failures per year:
  - approximately 8760 hours per year (24*365)
  - 0.001 * 8760 = 8.76 failures per year
- Availability
  - 8.76 * 3 = 26.28 hours of downtime per year.
  - Availability = 0.997 ((8760 - 26.28)/8760)

Activity - Availability

- Your customers want an availability of at least 99%, a POFOD of less than 0.1, and ROCOF of less than 2 failures per 8 hour work period.
- After testing your code for 7 full days, 972 requests were made. The product failed 64 times (37 system crashes, 17 bad calculations) and it took an average of 2 minutes to restart after each failure.
    - What is the availability, POFOD, and ROCOF?
    - Can we calculate MTBF?
    - Is the product ready to ship?
    - If not, why not?

Activity Solution

- What is the rate of fault occurrence?
  - 64/168 hours = 0.38/hour = 3.04/8 hour work day
- What is the POFOD?
  - 64/972 = 0.066
- What is the availability?
  - Was down for (37*2) = 74 minutes out of 168 hours = 74/10089 minutes = 0.7% of the time.
  - Availability is 0.993.

Activity Solution

- Can we calculate MTBF?
  - No - need timestamps. We know how long they were down (on average), but not when each crash occurred.
- Is the product ready to ship?
  - No. Availability/POFOD are good, but ROCOF is too low.
  - Suggestions for improvement?

Reliability Economics

- Raising reliability is expensive. It may be cheaper to accept unreliability and pay for failure costs.
- The balancing point depends on social and political factors and the system type.
  - A reputation for unreliable products may hurt more than the cost of improving reliability.
  - Cost of failure depends on risks of failure. For business systems, modest reliability may be fine.

Statistical Testing

- Rather than using tests to trigger faults, we can use tests to measure reliability.
    - Test inputs should match the predicted usage profile of a user.
    - By recording errors and other measurements, we can calculate ROCOF, POFOD, etc.
    - An acceptable level of reliability should be specified and the software tested until that level is reached.

Operational Profiles

- Reflects how the software is used.
- Consists of classes of input and the probability of their occurrence.
- Can be specified in advance if other systems exist that perform similar actions.
- For new systems, it is harder to specify.
  - Conduct beta testing to gather initial usage data.
  - Remember that usage changes over time.

Statistical Testing Procedure

- Study existing systems and form an operational profile.
- Construct test input that reflects the profile.
- Apply inputs and count the frequency and type of failures that occur, along with the time between failures.
- After observing a statistically significant number of failures, compute the reliability.

Statistical Testing Challenges

- Operation profile uncertainty
  - A profile based on other systems may not be valid for your system.
- High cost of test input generation
  - Large volume of inputs needed. Can be expensive.
- Statistical uncertainty
  - Need to generate enough failures to estimate reliability. This is hard when the system is already reliable.
  - Hard to estimate confidence in operational profile.

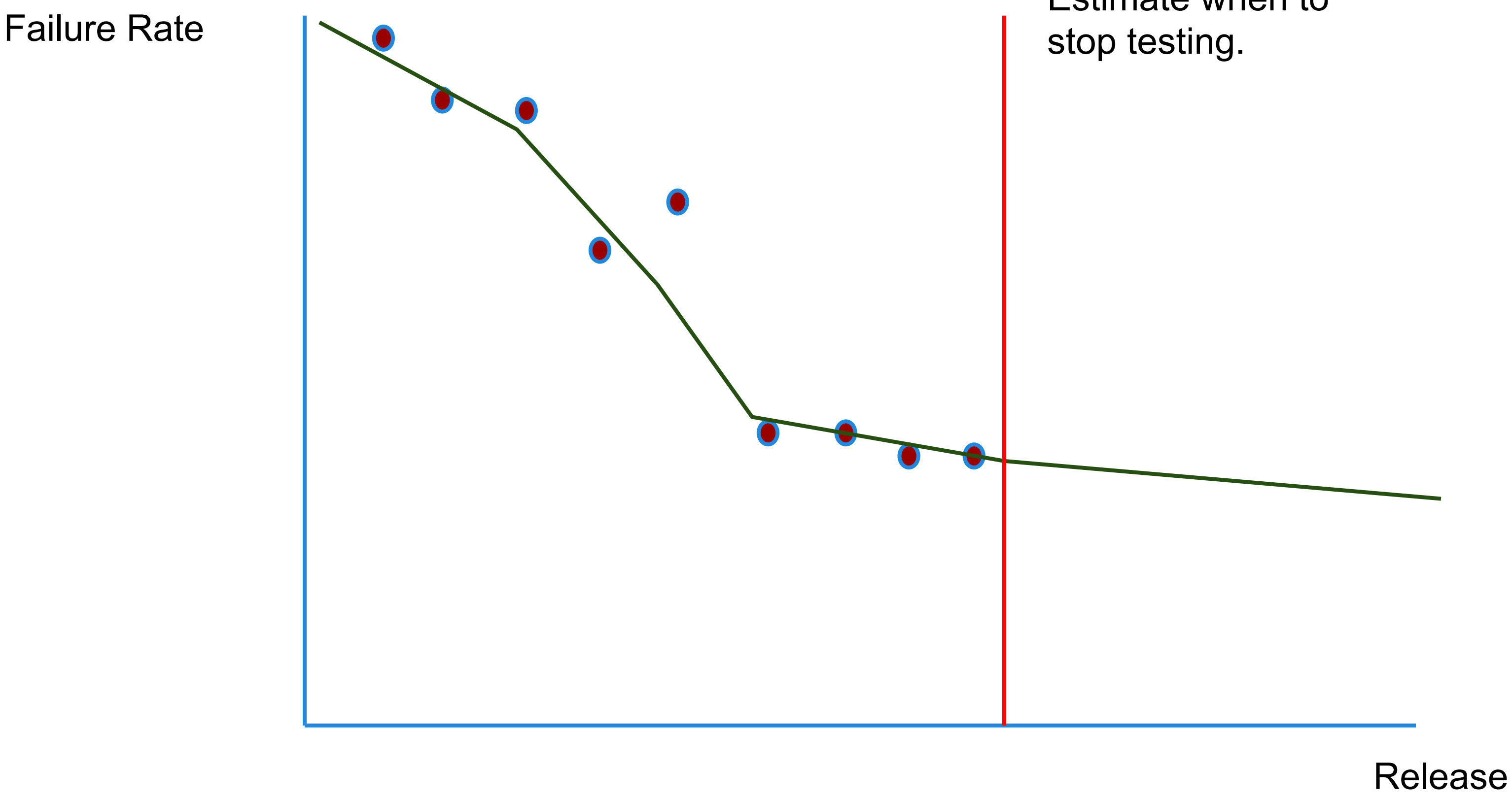
Getting the Most Out of Statistical Testing

- Statistical testing often reveals errors that do not emerge from other V&V activities.
  - As these emerge, fix the system and re-test.
  - As you gather more data, reliability growth can be modeled and used to plan testing.
- Mutation often used to plant “known faults” for reliability testing.
  - Can make an argument that you already rooted out most of the real faults.

Reliability Growth Modeling

- Can build mathematical model of the change in system reliability as changes are made to the code base.
- Used as a means of predicting additional reliability from more testing and changes.
- Use statistical testing to measure reliability of each system version, and make a call on when to stop trying to raise reliability.

Reliability Prediction



Estimate when to stop testing.

Key Points

- Reliability is one of the most important software characteristics.
- We should aim to produce reliable software.
- Reliability depends on the pattern of usage of the software. Different users will interact differently.
    - Faulty software can be reliable for some users.

Key Points

- Reliability can be measured quantitatively.
    - ROCOF, POFOD, Availability, MTBF
- Statistical testing is used to estimate reliability without actual users.
- Reliability growth models may be used to predict when a required level of reliability may be achieved.

When Do We Stop Testing?

- Come up with a plan that reflects your budget.
- Aim for correctness, robustness, and safety.
- You are done when you can present evidence that you have built a dependable system.
    - That is, a system that achieves a set dependability threshold.

Next Time

- Next time - Software Evolution and (Re)engineering
  - Reading: Sommerville, ch. 9
- Practice Final up soon.
  - Like midterm - no answers yet.
  - We will go over during the last class (12/7).
  - Final is 12/14