Software Reliability Class

- Software Engineering
  - SENG 691 D
- Computer Science
  - CS 791 X
- Time: Tuesdays 6PM – 8:30PM
  - SENG section is on-line, CS section in ESB 801
- Instructor: Bojan Cukic
  - Office phone: 304-293-0405 ext. 2526
  - Email: bojan.cukic@mail.wvu.edu

Rules of operation

- Email communication strongly preferred.
- Chats possible (gmail or Skype), need to arrange time.
- Office visits encouraged
  - ESB 731, Evansdale campus, Morgantown.
- Textbook:
  - The book is out of print but the 2nd edition can be obtained as a Print On Demand (POD) from AuthorHouse publishers with a considerable discount. Access the following Web site: http://members.aol.com/JohnDMusa/book.htm and follow appropriate links.

Rules of Operation

- Tests: Midterm and finals
- Presentations and research papers
  - Each student will choose a topic in agreement with the instructor.
  - Presentations during regular classes (on-line students use phone).
  - Presentations will grow into papers by the end of the semester.
  - SENG students can report on reliability best practices or the application of reliability engineering to projects they are involved with in their organizations.
  - CS student’s topics will require research content (project).
West Virginia University

Rules of Operation

- See the syllabus for details on tests, presentations and papers.
- Grading:
  - Midterm + Finals = 40%
  - Class presentation = 15%
  - Term papers / project reports = 35%
  - Class participation = 10%
- 90+ = A, 80+ = B, 70+ = C …
- Must obtain a passing grade from both tests and papers/presentations.

Software Reliability Engineering: A Short Overview

Bojan Cukic
Lane Department of Computer Science and Electrical Engineering
West Virginia University

Introduction

- Hardware for safety-critical systems is very reliable and its reliability is being improved
- Software is not as reliable as hardware, however, its role in safety-critical systems increases
- “Today, the majority of engineers understand very little about the science of programming or the mathematics that one needs to analyze a program. On the other hand, the scientists who study programming know very little about what it means to be an engineer…” [Parnas 1997]
Introduction

How good is software?

- Close to 75% of software projects never achieve completion or are never used
- 25% - 35% of UNIX utilities crash or hang the system when exposed to unusual inputs [Miller]
- OS X the most unreliable, according to these studies.
- 12 commercial programs for seismic data processing:
  - Numerical disagreement between results grows 1% per 4000 lines of source code [Hatton 94]

Software needs to be “sufficiently good” for its application

- Increased use of computerized control systems in safety critical applications
  - flight control, nuclear plant monitoring, robotic surgery, military applications, etc.
- Can we expect “perfect software” in practice?

lim_{resources \to \infty} “good software” = “perfect software”?

Introduction: Essential Difficulties

The goal of producing “perfect software” remains elusive [Brooks 86] due to:

- complexity
  - functional complexity, structural complexity, code complexity
- changing requirements
- invisibility
- Software faults introduced in all phases of the life-cycle: specification, design, implementation, testing, maintenance
Introduction: Ariane flight 501 failure

- Ariane 4 SRI (Inertial Reference Systems) software was reused on Ariane 5
- Ariane 4 accelerated much slower, used different trajectory
- In SRI-1 and SRI-2 Operand Error exception appeared due to an overflow in converting 64 bit floating point to 16 bit unsigned integer
- SRIs declared failure in two successive data cycles (72 ms)
- On Board Computer interpreted SRI-2 diagnostic pattern as flight data and commanded nozzle deflection
- 39s after launch, the launcher disintegrated because of high aerodynamic loads due to an angle of attack of more than 20 degrees

Infamous Software Failures

- July 28, 1962, Mariner I space probe.
  - A formula written by pencil on paper improperly coded.
  - Trajectory miscalculated, rocket diverted from the path at launch, destroyed over the Atlantic.
- 1982, Trans-Siberian gas pipeline explosion.
  - Fault planted into Canadian pipeline control software, covertly acquired by the Russians. Not much known about the nature of the fault.
  - Radiation therapy device delivers lethal doses at several facilities. Software safety interlock replaced electromechanical and failed. An operating system race condition was at fault.
- 1988, Buffer overflow at Berkeley Unix finger daemon.
  - Allowed the spread of the first internet worm. `gets()` function did not control the length of string, warm code was able to take control of the machines.
  - Generator not properly “seeded”. For 8 years, it was possible to break into the most “secure” authentication system using trivial mathematics. Not known whether fault ever exploited.
Infamous Software Failures

- Jan 15, 1990, AT&T network outage.
  - A fault in a new software release causes long distance switches to crash when they receive a crash recovery message from the neighboring machine. 114 switches kept crashing and rebooting every 6 seconds for 9 hours. Old software release loaded back to fix the problem.

- 1993, Intel Pentium floating point division.
  - Error of 0.006% in division causes public relations nightmare. 3 to 5 million chips in circulation, replacement for anyone who complains. $475M in damages.

- 1995/96 The Ping of death.
  - Malformed “ping” packets not checked and cause the computers to display the “blue screen of death”. Lack of error handling sanity checks. Windows, Mac and UNIX systems affected.

- November 2000, National cancer institute, Panama City.
  - Therapy planning software miscalculates the proper dosage of radiation. Doctors “trick” the software by placing additional shielding blocks not planned in software. Dosage calculation depends on peculiar user interaction. 8 patients die, at least 20 receive overdoses. Doctors, who were supposed to double-check computer’s calculations by hand, are indicted for murder.

Software Reliability

- Software Reliability: $P(A|B)$
  - $A$: Software does not fail when operated for $t$ time units under specified conditions.
  - $B$: Software has not failed at time 0.

- Ultra-high reliability requirements for safety-critical systems (Draft Int’l Standard IEC61508 for Safety Integrity Level 4):
  - Continuous control systems: $<10^4$ failures per hour
    - Airbus 320/330/340 and Boeing 777: $<10^4$ failures/h
    - This translates to 113,155 years of operation without encountering a failure
  - Protection systems (emergency shutdown): $<10^4$ failures/h
    - UK Seizewell B nuclear reactor (emerg.): $<10^3$ failures/h
Introduction

Software faults introduced in all phases of the life-cycle: specification, design, implementation, testing, maintenance.

Reliable operation of programmable electronics requires assurance in all the phases of the life-cycle.

Formal Verification

Software Reliability Assessment

Formal Verification

PRO:

- Proves program correctness, i.e., that the program meets its specifications
- Reliability 1 is established by proving the absence of implementation errors
- Independent of operational profile (system usage)

CONS:

- Cannot cope with specification errors, OS, compilers and hardware faults
- Proofs can be erroneous, unless performed automatically
- Its applicability limited to small & medium size programs

Formal methods in SE

Used for requirements specifications and verification

Based on mathematical logic, state machines or process algebra

Most popular forms of verification

Model checking

- Finite state transition model represents the system
- Constraints expressed in temporal logic
- Significant limitations wrt. system size!

Formal verification: Proving properties from the set of axioms
Time Domain Approach

- **Software Reliability Assessment**
  - **Formal Verification**
  - **Testing**
  - **Time Domain**
  - **Input Domain**

  - Observed failure data from testing fitted to various statistical models
  - Time-Between-Failure models, and Period Failure Count models
  - Used for:
    - Assessing current reliability
    - Predicting future reliability
    - Controlling software testing
  - **CONS:**
    - Perfect fault removal assumed
    - Cannot be used to predict ultra-high reliability levels

Time domain models

- **Reliability Growth models**
  - Jelenski-Moranda model (JM)
    - The number of initial faults unknown but fixed
    - Fault detection is perfect (no new faults introduced)
    - Times between failure occurrences are independent exponentially distributed random quantities
    - All remaining faults contribute equally to failure intensity
  - **General problems (more assumptions)**
    - All faults detectable
    - Statistical independence of inter-failure arrival

Related Work: Statistical testing

- **Software Reliability Assessment**
  - **Formal Verification**
  - **Testing**
  - **Time Domain**
  - **Input Domain**

  - **PROS**
    - System level assessment
    - Theoretically sound
  - **CONS**
    - Large number of test cases, an oracle needed
    - Depends on the operational profile
Urn Model of Software Testing

- Random software testing is modeled as sampling with replacement.
- If repeated sampling reveals no black balls, all we gain is a confidence that there are none.
- Only one type of testing can prove that there are no existing faults: exhaustive testing.

Program with 20 variables, 10 values/var, 1 ms per test case

\[ \geq 3,000 \text{ years of testing!} \]

Prob. Operational distribution

Input values (balls)

Introduction: Dependability

- Safety-critical systems require both
  - best practices for software development with dependability being the major concern
  - rigorous validation procedures

A Reality Check

- Collection of operational software data is difficult.
- Problem occurrence rates for essential aircraft flight functions [Shooman 96]:
  - \( 2 \times 10^8 \) to \( 10^6 \) occurrences per hour of operation
  - The reported failure occurrence rates are higher than required.
- Error, Fault and Failure (EFF) data collection initiatives
  - Come and go
  - We still miss data!!!
Software Reliability

- Measurement?
- Practical techniques?
- Right or wrong?
  - (Un)reliability of released products
  - Missed schedules
  - Cost overruns
  - Market share/reaction?

What is SRE

- The set of best practices that empower testers and developers to
  - Ensure product reliability meets users needs
  - Speed the product to market faster
  - Reduce product cost
  - Improve customer satisfaction (fewer angry users)
  - Increase their productivity
- Applicable to all software based systems
- Two fundamental ideas
  - Focus resources on the most used/critical functions
  - Make testing realistically represent field conditions

SRE Process

- Widely used and accepted, especially by the large corporations (Microsoft included!!!)
- Increase in project cost: less than 1%
- Predominant SRE workflow:
  - Define Necessary Reliability
  - Develop Operational Profiles
  - Prepare for Test
  - Execute Tests
  & Apply Failure Data to Guide Decisions
  - Test & Validation
SRE Process

- Tasks frequently iterate
- Post-delivery and maintenance phase (not shown)
- Testers must be involved throughout the process
- Allows better understanding of user’s perspective
- Improvement of system requirements, planning
- Selection of appropriate mix of
- fault prevention
- fault removal
- fault tolerance

SRE

- Types of tests applicable to SRE (based on objectives, rather than phases in the life-cycle)
  - Reliability growth tests (find and remove faults)
    - need a minimum of 10-20 detected faults to achieve statistically meaningful results
  - Feature (minimize impact of the environment), load (maximize environmental impacts), regression tests (following a major change)
  - Certification tests
    - no debugging, accept or reject software under test
    - no. observed failures not important

Defining the “system”

- System is an independently tested unit
- SRE should be applied to subsystems (acquired COTS, OS, for example), systems and supersystems
- Different configuration represents different system
  - Interface stubs may not be correct
- But, more “systems” implies higher cost
  - aggregation welcome
  - Product lines help reducing the cost
SRE and SW design & test process

- Use knowledge of operational profile to guide and focus design efforts
- Established failure intensity drives the quality assurance efforts
- Failure intensity goal determines when to stop testing
- Measurement throughout the life-cycle helps identify better methodologies

Is Reliability Important?

- It should be, since it is measurable property
  - Unlike “software quality”
  - Useful, since the software is tested under the conditions of perceived usage.
  - The number of resident faults, for example, is a developer oriented measure. Reliability is a user oriented measure.
  - The number of faults found has NO correlation to reliability. Neither has program complexity.
  - Accurate measurements of reliability are feasible.

Why to Measure Reliability?

- Isn’t the “best software development process” sufficient?
  - What is “best’”?
  - It is important to measure the results of the process.
- Early consideration of target reliability is beneficial, since it impacts cost and schedule.
- CMM levels 4 and 5 (and 3, indirectly), recommend reliability measurement.
Common Misconceptions

- Software reliability is primarily concerned with software reliability models.
- It copies hardware reliability theory.
- Not, because reliability of software is more likely to change over time (modifications, upgrades).
- It deals with faults or “bugs”.
- It does not concern itself with requirements based testing.
- Testing “ultrareliable” software is hopeless.

Reliability Measurement

- Observe failure occurrences in terms of execution time.

<table>
<thead>
<tr>
<th>Failure No</th>
<th>Failure time (s)</th>
<th>Failure interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>43</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>88</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>103</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>125</td>
<td>22</td>
</tr>
</tbody>
</table>

Measurements

- Typical variation of failure intensity and reliability over testing
- Each expression has its advantages
- Curves not necessarily so smooth
- Alternatives
  - MTTF (larger better), but may be undefined.
  - MTBF=MTTF+MTTR (comes from HW reliability)
Summary

- Definition of software reliability.
- Software reliability engineering is the process that leads to high reliability software.
- Based on statistical evaluation of quality factors throughout the development lifecycle.
- Reliability can be assessed using different approaches.
- Simple activities can significantly reduce software failure rates.