SE 2A04 Fall 2002

04 Verification and Analysis

Instructor: W. M. Farmer

Revised: 18 October 2002

The Problem

- What behavior does the software product exhibit?
  Is the behavior correct?
  Is the behavior acceptable?

- Measures of software quality:
  - **Correctness**: To what extend does the product satisfy its requirements specification?
  - **Reliability**: How probable is correct behavior?
  - **Trustworthiness**: How probable is critical failure?

- Forms of verification and analysis:
  - Inspection
  - Testing
  - Mathematical verification

Correctness

- Full correctness is very difficult to achieve and even more difficult to demonstrate
- Some lack of correctness must usually be accepted
  - It can be possible to achieve and prove full correctness for some simple software products
  - For most software products, full correctness is an unaffordable dream
- Full correctness is an important goal but rarely necessary
- Inspection, testing, and mathematical verification can show incorrectness, but mathematical verification is needed to show correctness

Reliability

- Reliability is a useful measure when:
  - All errors are considered equally important
  - There are no critical failures
  - The operating conditions are predictable
  - We want to compare risks
- Testing is most useful for measuring reliability

Trustworthiness

- Some systems have critical requirements that must be fully satisfied by the software product
  - It can be useful to rank the requirements by how critical they are
- Critical requirements may concern such things as:
  - Safety to users and the environment
  - Information security
  - High cost of failure
- Inspection and mathematical verification are useful for measuring trustworthiness, but testing is not
- Unreliable products are often accepted, but untrustworthy products with critical requirements should never be accepted

Product Inspection

- The full product, both documentation and code, should be inspected

- The inspection should be **systematic**
  - Guided by checklists and questionnaires

- The inspection should be an **active** process
  - Inspectors use the product documents
  - They document their analysis and provide specifics
  - They produce their own product descriptions from the code which they compare with the product specifications

- The inspection should be performed by a small team that includes people with different kinds of expertise

Software Testing

- Testing can show instances of incorrectness, but it is usually not practical for demonstrating correctness and trustworthiness
  - There are often an unbounded number of possible inputs and environmental configurations
  - Only what is executable (code but usually not specifications) can be tested
- Positive testing results are not, by themselves, an indication of software quality
- Testing can be used to assess reliability
- The smallest components and the lowest levels of the uses hierarchy should be tested first
  - Integration should be done only after the components have been fully tested

Kinds of Code Testing

1. **Blackbox testing**
   - Based on the specification alone
   - Test cases chosen without looking at the code
   - Can be reused with a new implementation
   - Can be done independently of the designer

2. **Clearbox testing**
   - Based on the code
   - Test cases chosen by looking at code
   - Tests the implementation mechanism

3. **Greybox testing**
   - Intended for modules with internal data structures
   - Test cases chosen with respect to the internal data structures
   - Gives better coverage than blackbox testing

Kinds of Test Case Selection

1. **Planned**: Test cases selected to cover the behavior of the code
   - Based on specification (blackbox)
   - Based on code (clearbox)
   - Based on internal data structures (graybox)

2. **Wild random**: Test cases selected using a uniform random distribution
   - Can find cases nobody thought of
   - Can violate assumptions yielding spurious results

3. **Statistical random**: Test cases selected using an operational profile
   - Provides meaningful reliability figures
   - Only as good as the operational profile

General Recommendations (Parnas)

1. Test all possible paths through the program
   - So every possible statement is tested at least once
2. Test all data states
3. Test all degenerate data states
4. Test extreme cases
   - Try very large numbers
   - Try very small numbers
5. Test erroneous cases
6. Think of cases that nobody thinks of

Mathematical Verification

- Main idea: Use the mathematics process to analyze the behavior of a software product
  - Most effective for high-level design
  - Requires significant human expertise
  - Requires effective machine support
  - Can be very expensive

- The mathematics process consists of three activities:
  1. **Model creation**: Create mathematical models that represent mathematical aspects of the world
  2. **Model exploration**: Explore the models by stating and proving conjectures and by performing calculations
  3. **Model connection.** Connect the models to one another so that results obtained in one model can be used in other models

Two Approaches

1. **Informal but rigorous**: Models are expressed using a natural language and are explored by informal conjecture proving and computation
    - All the work is done by humans
    - Usually not feasible for problems with many details

2. **Formal and mechanized**: Models are expressed and explored using a **mechanized mathematics system** like a theorem proving system or computer algebra system
    - A major portion of the work is done by machine

In most applications, the mathematical verification will be a mixture of these two approaches

Application to Software

- Problem: Does an implementation I satisfy a specification S?

- First solution:
  - Choose an appropriate axiomatic theory T in an appropriate background logic L
  - Formalize I as a term $\tilde{I}$ in T
  - Formalize S as a unary predicate $\tilde{S}$
  - Prove in L that $\tilde{S}(\tilde{I})$ is a theorem of T

- Second solution:
  - Choose an appropriate background logic L
  - Formalize I as a theory T_I in L
  - Formalize S as a theory T_S in L
  - Show that there is an interpretation of T_S in T_I

Final Comments

- Verification and analysis should be done at all stages in the development of a software product—the earlier the better

- Inspection, testing, and mathematical verification complement each other
  - Inspection is good for finding things that are missing in the software product and in its documentation
  - Testing is good for finding low-level errors, especially coding errors
  - Mathematical verification is good for finding high-level errors, especially design errors

- The same documentation should be used for inspection, testing, and mathematical verification

References

1. D. Parnas and D. Weiss, ‘‘Active design reviews: principles and practices’’, in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.

2. D. Parnas, “Inspection of safety-critical software using program-function tables’’, in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.