Verification and Validation

Steven Zeil

February 13, 2013

Contents

V & V

Verification & Validation: assuring that a software system meets the users' needs
Principal objectives:

- The discovery of defects in a system
- The assessment of whether or not the system is usable in an operational situation.

. . . . . .

1 The Process

Verification & Validation

- Verification:
  - "Are we building the product right"
  - The software should conform to its (most recent) specification
- Validation:
  - "Are we building the right product"
  - The software should do what the user really requires

. . . . . .

Testing

- *Testing* is the act of executing a program with selected data to uncover bugs.
  - As opposed to *debugging*, which is the process of finding the faulty code responsible for failed tests.
- Testing is the most common, but not the only form of V&V

. . . . . .

What Can We Find?

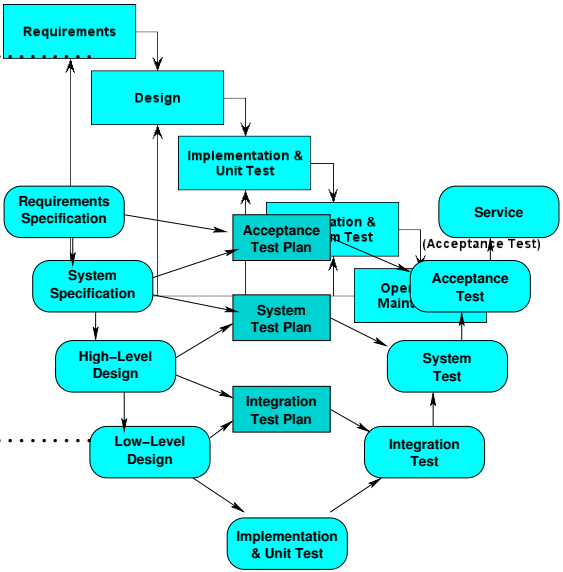
- *Fault*: A defect in the source code.
- *Failure*: An incorrect behavior or result.
- *Error*: A mistake by the programmer, designer, etc., that led to the the fault.

. . . . . .

Industry figures of 1-3 faults per 100 statements are quite common.

In Context

• V& V is often portrayed as final phase of waterfall



In Fuller Context

But is more properly a process-wide activity

• Requirements must be validated

• Designs may be validated & verified

• Implementation is tested

• final system is tested

• maintenence changes are tested

2 Non-Testing V&V

Static Verification

Verifying the conformance of a software system and its specification without executing the code

- Involves analyses of source text by humans or software
- Can be carried out on ANY documents produced as part of the software process
- Discovers errors early in the software process
- Usually more cost-effective than testing for defect detection at the unit and module level
- Allows defect detection to be combined with other quality checks

Static verification effectiveness

It has been claimed that

- More than 60% of program errors can be detected by informal program inspections
- More than 90% of program errors may be detectable using more rigorous mathematical program verification

. . . . . .

2.1 Code Review

Code Review

Inspecting the code in an effort to detect errors

- Desk Checking
- Inspection

. . . . . .

Desk Checking

- An exercise conducted by the individual programmer.
- "Playing computer" with the aid of a listing.
  - Values of variables are tracked using pencil and paper as the programmer moves step-by-step through the code.
- Can be done with pseudocode, diagrams, etc. even before code has been written

- A good way to find fundamental flaws in algorithms, especially before actually writing the code.
- Useful in checking the results of an intended change during debugging.

Inspection

- Formalized approach to document reviews
- Intended explicitly for defect detection (not correction)
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable) or non-compliance with standards

Inspection pre-conditions

- A precise specification must be available
- Team members must be familiar with the organization standards
- Syntactically correct code must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process
- Management must not use inspections for staff appraisal

Inspection procedure

- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- After inspection meeting,
  - Modifications are made to repair discovered errors
  - Re-inspection may or may not be required

. . . . . .

Inspection checklists

- Checklist of common errors should be used to drive the inspection
- Error checklist is programming language dependent
- The “weaker” the type checking, the larger the checklist
- Examples: Initialization, Constant naming, loop termination, array bounds, etc.

. . . . . .

Inspection checks

- Data Faults
- Control Faults
- I/O Faults
- Interface faults
- Storage Mgmt Faults
- Stylistic/standards Faults

. . . . . .

Inspection checks

- Data Faults
  - Are all variables initialized before use?
  - Have all constants been named?
  - Should array lower bounds be 0, 1, or something else?
  - Should array upper bounds be size of the array or size−1?
  - If character strings are used, is a delimited explicitly.
  - Are all data members initialized in every constructor?
  - C++'s "Rule of the Big 3" assigned?

. . . . . .

Inspection checks

- Control Faults
  - For each conditional statement, is the condition correct?
  - Is each loop certain to terminate?
  - Are compound statements correctly bracketed?
  - In case statements, are all possible cases accounted for?

. . . . . .

Inspection checks

- I/O Faults
  - Are all input variables used?
  - Are all output variables assigned before being output?

. . . . . .

Inspection checks

- Interface faults
  - Do all function/procedure calls have the correct number of parameters?
  - Do the formal and actual parameter types match?
  - Are the parameters in the right order?
  - If components access shared memory, do they have the same model of the shared memory structure?

. . . . . .

Inspection checks

- Storage Mgmt Faults
  - If a linked structure is modified, have all links been correctly assigned?
  - If dynamic storage is used, has space been allocated correctly?
  - Is space explicitly deallocated after it is no longer required?
  - Are all pointer data members deallocated in the destructor?

.....

Inspection checks

- Exception Mgmt Faults
  - Have all possible error conditions been taken into account?

.....

Inspection checks

- Stylistic/standards Faults
  - Are names understandable?
  - Does code conform to standards for commenting?
  - Does code provide capturable outputs for testing?
  - Does code take advantage of possible re-use?

.....

2.2 Mathematically-based verification

Mathematically-based verification

- Verification is based on mathematical arguments which demonstrate that a program is consistent with its specification
- Programming language semantics must be formally defined
- The program must be formally specified

. . . . . .

Program proving

- Rigorous mathematical proofs that a program meets its specification are long and difficult to produce
- Some programs cannot be proved because they use constructs such as interrupts. These may be necessary for real-time performance
- The cost of developing a program proof is so high that it is not practical to use this technique in the vast majority of software projects

. . . . . .

Program verification arguments

- Less formal, mathematical arguments can increase confidence in a program's conformance to its specification
- Must demonstrate that a program conforms to its specification

- Must demonstrate that a program will terminate

. . . . . .

Model Checking

- Simplified models on which porperties can be proved
  - FSA
  - Markov Chains
- Focus on properties short of correctness
  - e.g., avoiding race conditions
- Machine-Assisted

. . . . . .

2.3 Static analysis tools

Static analysis tools

- Software tools for source text processing
- Try to discover potentially erroneous conditions in a program and bring these to the attention of the V & V team
- Very effective as an aid to inspections. A supplement to but not a replacement for inspections

. . . . . .

Static analysis checks

- Data faults
- Control faults
- Interface faults
- Storage mgmt faults

. . . . . .

Static analysis checks

- Data faults
  - Variables used before initialization
  - Variables declared but never used
  - Variables assigned twice but never used between assignments
  - Possible array bounds violations
  - Undeclared variables

. . . . . .

Static analysis checks

- Control faults
  - Unreachable code
  - Unconditional branches into loops

. . . . . .

Static analysis checks

- Interface faults
  - Parameter type mismatches
  - Parameter number mismatches
  - Function return values unused
  - Uncalled functions and procedures

. . . . . .

Static analysis checks

- Storage mgmt faults
  - Unassigned pointers
  - Pointer arithmetic

. . . . . .

Stages of static analysis

- Control flow analysis.
  - Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- Data use analysis.

  - Detects uninitialized variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.

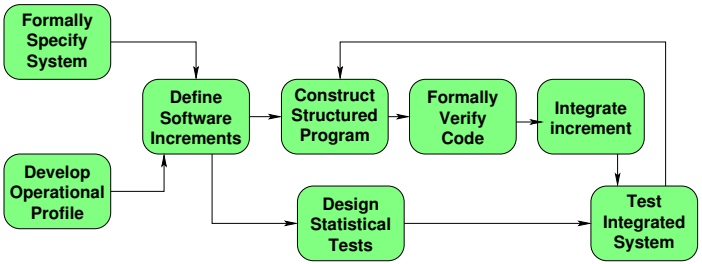
- Interface analysis.
  - Checks the consistency of routine and procedure declarations and their use
- Information flow analysis.
  - Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- Path analysis.
  - Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. Must be used with care.

. . . . . .

2.4 Cleanroom software development

Cleanroom software development

The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal



- Software development process based on:
  - Incremental development
  - Formal specification.
  - Static verification using correctness arguments
  - Statistical testing to determine program reliability.

. . . . . .

Cleanroom process teams

Specification team. Responsible for developing and maintaining the system specification

Development team. Responsible for developing and verifying the software. The software is *NOT* executed during this process

Certification team. Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable

. . . . . .

- Results in IBM have been very impressive with few discovered faults in delivered systems
- Some independent assessment shows that the process is no more expensive than other approaches
  - But no controlled studies
  - And their assessment of other approaches is questionable
- Fewer errors than in a 'traditional' development process
- Not clear how this approach can be transferred to an environment with less skilled or less highly motivated engineers

3 Testing

Testing stages

- *Unit Test*: Tests of individual subroutines and modules,
  - usually conducted by the programmer.
- *Integration Test*: Tests of "subtrees" of the total project hierarchy chart (groups of subroutines calling each other).
  - generally a team responsibility.
- *System Test*: Test of the entire system,
  - supervised by team leaders or by V&V specialists.
  - Many companies have independent teams for this purpose.

- *Regression Test*: Unit/Integration/System tests that are repeated after a change has been made to the code.
- *Acceptance Test*: A test conducted by the customers or their representatives to decide whether to purchase/accept a developed system.

. . . . . .

Testing goals

- Unit Test: does it work?
- Integration Test: does it work?
- System Test: does it work?
- Regression Test: has it changed?
- Acceptance Test: should we pay for it?

. . . . . .

3.1 Unit Testing

Unit Testing

By testing modules in isolation from the rest of the system

- Easier to design and run extensive tests
- Much easier to debug any failures
- Errors caught much earlier

Main challenge is *how* to test in isolation

. . . . . .

Scaffolding

To do Unit tests, we have to provide replacements for parts of the program that we will omit from the test.

- *Scaffolding* is any code that we write, not as part of the application, but simply to support the process of Unit and Integration testing.
- Scaffolding comes in two forms
  - Drivers
  - Stubs

. . . . . .

Drivers

A *driver* is test scaffolding that calls the module being tested.

- Often just a simple main program that reads values, uses them to construct ADT values, apply ADT operations and print the results

. . . . . .

3.2 Integration Testing

Integration Testing

Integration testing is testing that combines several modules, but still falls short of exercising the entire program all at once.

- Integration testing usually combines a small number of modules that call upon one another.
- Integration testing can be conducted

  - bottom-up

    (start by unit-testing the modules that dont'call anything else, then add the modules that call those starting modules and thest the combination, then add the modules that call those, and so on until you are ready to test `main()`.)

    * relieves the need for stubs

  - or *top-down* (start by unit-testing `main()` with stubs for everything it calls, then replace those stubs by the real code, but leaving in stubs for anything called from the replacement code, then replacng those stubs, and so on, until you have assembled and tested the entire program).

. . . . . .

Types of testing

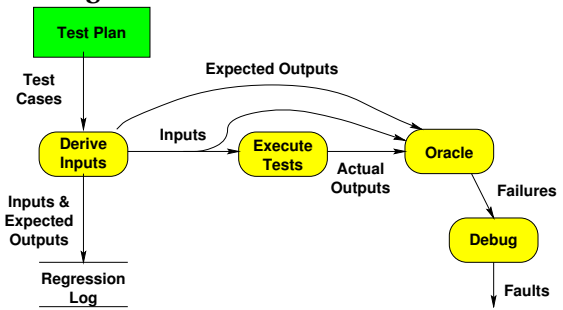
- Statistical testing
  - tests designed to reflect the frequence of user inputs.

    Used for reliability estimation.
- Defect testing
  - Tests designed to discover system defects.
  - A successful defect test is one which reveals the presence of defects in a system.

. . . . . .

4 The Testing Process

The Testing Process



Testing Process Components

- *Test case*: a general description of a required test
  - There may be many possible inputs that would serve
- *Regression Log*: Database of tests and expected outputs
  - Used during regression testing to quickly rerun old tests
- *Oracle*: The process, person, and/or program that determines if test output is correct
- *Failure*: An execution on which incorrect behavior occurs
- *Fault*: A defect in the code that (may) cause a failure

- *Error*: A human mistake that results in a fault

. . . . . .