Verification and Validation


Tutorial Questions

  - Question 19.1
  - Question 19.2
  - Question 19.4
  - Question 20.1
  - Question 20.2
  - Question 20.8

Verification and Validation

- Assuring that a software system meets a user’s needs

Objectives

- To introduce software verification and validation and to discuss the distinction between them
- To describe the program inspection process and its role in V & V
- To explain static analysis as a verification technique
- To describe the Cleanroom software development process

Topics covered

- Verification and validation planning
- Software inspections
- Automated static analysis
- Cleanroom software development
Verification vs validation

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

The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.

Static and dynamic verification

- **Software inspections** Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis
- **Software testing** Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

Static and dynamic V&V

Program testing

- Can reveal the presence of errors NOT their absence
- A successful test is a test which discovers one or more errors
- The only validation technique for non-functional requirements
- Should be used in conjunction with static verification to provide full V&V coverage

Types of testing

- Defect testing
  - Tests designed to discover system defects.
  - A successful defect test is one which reveals the presence of defects in a system.
  - Covered in Chapter 20
- Statistical testing
  - Tests designed to reflect the frequency of user inputs. Used for reliability estimation.
  - Covered in Chapter 21
V& V goals

- Verification and validation should establish confidence that the software is fit for purpose
- This does NOT mean completely free of defects
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed

V & V confidence

- Depends on system’s purpose, user expectations and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organisation
  - User expectations
    - Users may have low expectations of certain kinds of software
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program

Testing and debugging

- Defect testing and debugging are distinct processes
- Verification and validation is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

Testing and debugging

- Defect testing and debugging are distinct processes
- Verification and validation is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

The debugging process

- Test results
- Specification
- Design
- Error repair
- Repair error
- Re-test program

The V-model of development

- Careful planning is required to get the most out of testing and inspection processes
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning is about defining standards for the testing process rather than describing product tests

The V-model of development

- Careful planning is required to get the most out of testing and inspection processes
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning is about defining standards for the testing process rather than describing product tests
The structure of a software test plan
- The testing process
- Requirements traceability
- Tested items
- Testing schedule
- Test recording procedures
- Hardware and software requirements
- Constraints

Software inspections
- Involve people examining the source representation with the aim of discovering anomalies and defects
- Do not require execution of a system so may be used before implementation
- May be applied to any representation of the system (requirements, design, test data, etc.)
- Very effective technique for discovering errors

Inspection success
- Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise

Inspections and testing
- Inspections and testing are complementary and not opposing verification techniques
- Both should be used during the V & V process
- Inspections can check conformance with a specification but not conformance with the customer’s real requirements
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Program inspections
- Formalised 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 uninitialised variable) or non-compliance with standards

Inspection pre-conditions
- A precise specification must be available
- Team members must be familiar with the organisation 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
The inspection process

- Planning
- Overview
- Individual preparation
- Inspection meetings
- Follow-up
- Re-inspection

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
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required

Inspection teams

- Made up of at least 4 members
- Author of the code being inspected
- Inspector who finds errors, omissions and inconsistencies
- Reader who reads the code to the team
- Moderator who chairs the meeting and notes discovered errors
- Other roles are Scribe and Chief moderator

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: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection checks

- Fault class Inspection check
  - **Data faults**
    - Are all program variables initialised before their values are used?
    - Have all constants been named?
    - Should the lower bound of arrays be 0, 1, or something else?
    - Should the upper bound of arrays be equal to the size of the array or Size -1?
    - If characters strings are used, is a delimiter explicitly assigned?
  - **Control faults**
    - For each conditional statement, is the condition correct?
    - Is each loop certain to terminate?
    - Are compound statements correctly bracketed?
  - **Input/output faults**
    - Are all input variables used?
    - Are all output variables assigned a value before they are output?
  - **Interface faults**
    - Do all function and procedure calls have the correct number of parameters?
    - Do 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?
  - **Storage management faults**
    - Are all output variables assigned a value before they are output?
    - Have all links been correctly reassigned?
    - If dynamic storage is used, has space been allocated correctly?
    - Is space explicitly deallocated after it is no longer required?
  - **Exception management faults**
    - Have all possible error conditions been taken into account?

Inspection rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 500 lines costs about 40 man/hours effort = £2800
Automated static analysis

- Static analysers are software tools for source text processing
- They parse the program text and try to discover potentially erroneous conditions 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

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Static analysis check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data faults</td>
<td>Variables used before initialisation</td>
</tr>
<tr>
<td></td>
<td>Variables declared but never used</td>
</tr>
<tr>
<td></td>
<td>Variables assigned twice but never used between assignments</td>
</tr>
<tr>
<td></td>
<td>Possible array bound violations</td>
</tr>
<tr>
<td></td>
<td>Undeclared variables</td>
</tr>
<tr>
<td>Control faults</td>
<td>Unreachable code</td>
</tr>
<tr>
<td></td>
<td>Unconditional branches into loops</td>
</tr>
<tr>
<td>Input/output faults</td>
<td>Variables output twice with no intervening assignment</td>
</tr>
<tr>
<td>Interface faults</td>
<td>Parameter type mismatches</td>
</tr>
<tr>
<td></td>
<td>Parameter number mismatches</td>
</tr>
<tr>
<td></td>
<td>Non-usage of the results of functions</td>
</tr>
<tr>
<td></td>
<td>Uncalled functions and procedures</td>
</tr>
<tr>
<td>Storage management faults</td>
<td>Unassigned pointers</td>
</tr>
<tr>
<td></td>
<td>Pointer arithmetic</td>
</tr>
</tbody>
</table>

Stages of static analysis

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

Stages of static analysis

- **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.

Use of static analysis

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler
- Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation

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.
The Cleanroom process

- Define software increments
- Structured programming: limited control and abstraction constructs are used
- Incremental development
- Formally specify and verify code
- Formally specify system
- Static verification: rigorous inspections
- Statistical testing of the system (covered in Ch. 21)

Cleanroom process characteristics

- Formal specification using a state transition model
- Incremental development
- Structured programming: limited control and abstraction constructs are used
- Static verification using rigorous inspections
- Statistical testing of the system (covered in Ch. 21)

Incremental development

- Establish requirements
- Formal specification
- Develop software
- Deliver software
- Requirements change request

Formal specification and inspections

- The state based model is a system specification and the inspection process checks the program against this model
- Programming approach is defined so that the correspondence between the model and the system is clear
- Mathematical arguments (not proofs) are used to increase confidence in the inspection process

LINT static analysis

138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
int Anarray;
{
    printf("%d",Anarray);
}
main()
{
    int Anarray[5]; int i; char c;
    printarray (Anarray, i, c);
    printarray (Anarray);
}

139% cc lint_ex.c
140% lint lint_ex.c
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
• Specification team. Responsible for developing and maintaining the system specification.
• Development team. Responsible for developing and verifying the software. The software is NOT executed or even compiled 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. Independent assessment shows that the process is no more expensive than other approaches. 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.

Key points
• Verification and validation are not the same thing. Verification shows conformance with specification; validation shows that the program meets the customer's needs.
• Test plans should be drawn up to guide the testing process.
• Static verification techniques involve examination and analysis of the program for error detection.

Program inspections are very effective in discovering errors. Program code in inspections is checked by a small team to locate software faults. Static analysis tools can discover program anomalies which may be an indication of faults in the code. The Cleanroom development process depends on incremental development, static verification and statistical testing.

Defect testing
• Testing programs to establish the presence of system defects.

Software Testing
Objectives

• To understand testing techniques that are geared to discover program faults
• To introduce guidelines for interface testing
• To understand specific approaches to object-oriented testing
• To understand the principles of CASE tool support for testing

Topics covered

• Defect testing
• Integration testing
• Object-oriented testing
• Testing workbenches

The testing process

• Component testing
  – Testing of individual program components
  – Usually the responsibility of the component developer (except sometimes for critical systems)
  – Tests are derived from the developer’s experience
• Integration testing
  – Testing of groups of components integrated to create a system or sub-system
  – The responsibility of an independent testing team
  – Tests are based on a system specification

Testing phases

Component testing   Integration testing
Software developer   Independent testing team

Defect testing

• The goal of defect testing is to discover defects in programs
• A successful defect test is a test which causes a program to behave in an anomalous way
• Tests show the presence not the absence of defects

Testing priorities

• Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible
• Tests should exercise a system's capabilities rather than its components
• Testing old capabilities is more important than testing new capabilities
• Testing typical situations is more important than boundary value cases
Test data and test cases

- **Test data** Inputs which have been devised to test the system
- **Test cases** Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

The defect testing process

1. Design test cases
2. Prepare test data
3. Run program with test data
4. Compare results to test cases

Black-box testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process

Equivalence partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition
**Equivalence partitioning**

- Partition system inputs and outputs into 'equivalence sets'
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are
    - <10,000, 10,000-99,999 and > 10,000
  - Choose test cases at the boundary of these sets
    - 00000, 09999, 10000, 99999, 10001

---

**Search routine specification**

```plaintext
procedure Search (Key : ELEM; T: ELEM_ARRAY; Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
Pre-condition
-- the array has at least one element
T'FIRST <= T'LAST
Post-condition
-- the element is found and is referenced by L
( Found and T (L) = Key)
or
-- the element is not in the array
( not Found and not (exists. T'FIRST >= i <= T'LAST, T (i) = Key ))
```

---

**Testing guidelines (sequences)**

- Test software with sequences which have only a single value
- Use sequences of different sizes in different tests
- Derive tests so that the first, middle and last elements of the sequence are accessed
- Test with sequences of zero length

---

**Search routine - input partitions**

- Inputs which conform to the pre-conditions
- Inputs where a pre-condition does not hold
- Inputs where the key element is a member of the array
- Inputs where the key element is not a member of the array
Structural testing

- Sometime called white-box testing
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Objective is to exercise all program statements (not all path combinations)

White-box testing

![Diagram of white-box testing process]

- Test data
- Derives
- Component code
- Test outputs

Binary search (Java)

```java
class BinSearch {
    // This is an encapsulation of a binary search function that takes an array of ordered objects and a key and returns an object with 2 attributes namely index - the value of the array index
    // found - a boolean indicating whether or not the key is in the array
    // An object is returned because it is not possible in Java to pass basic types by reference to a function and so return two values
    // the key is -1 if the element is not found
    public static void search ( int key, int[] elemArray, Result r ) {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top ) {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key) {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} // BinSearch
```

Binary search - equiv. partitions

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array
- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values
- Input array has an odd number of values

Binary search - test cases

<table>
<thead>
<tr>
<th>Input array (T)</th>
<th>Key (Key)</th>
<th>Output (Found, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>17, 21, 23, 29</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>9, 16, 18, 30, 31, 41, 45</td>
<td>45</td>
<td>true, 7</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 38, 41</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 33, 38</td>
<td>21</td>
<td>true, 3</td>
</tr>
<tr>
<td>12, 18, 21, 23, 32</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>21, 23, 29, 33, 38</td>
<td>25</td>
<td>false, ??</td>
</tr>
</tbody>
</table>
Path testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.

Program flow graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node.
- Used as a basis for computing the cyclomatic complexity.
- Cyclomatic complexity = Number of edges - Number of nodes + 2

Cyclomatic complexity

- The number of tests to test all control statements equals the cyclomatic complexity.
- Cyclomatic complexity equals number of conditions in a program.
- Useful if used with care. Does not imply adequacy of testing.
- Although all paths are executed, all combinations of paths are not executed.

Independent paths

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- Test cases should be derived so that all of these paths are executed.
- A dynamic program analyser may be used to check that paths have been executed.

Integration testing

- Tests complete systems or subsystems composed of integrated components.
- Integration testing should be black-box testing with tests derived from the specification.
- Main difficulty is localising errors.
- Incremental integration testing reduces this problem.
Incremental integration testing

Test sequence 1  Test sequence 2  Test sequence 3

Approaches to integration testing

- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Top-down testing

Level 1  Level 2  Level 3

Le vel 2 stubs

Level 2 stubs

Level 2

Testing sequence

Bottom-up testing

Level N

Level N

Level N

Level N

Level N

Testing sequence

Testing approaches

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration testing
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

Interface testing

- Takes place when modules or sub-systems are integrated to create larger systems
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces
- Particularly important for object-oriented development as objects are defined by their interfaces
Interface testing

Test cases

A

B

C

Interfaces types

• Parameter interfaces
  – Data passed from one procedure to another
• Shared memory interfaces
  – Block of memory is shared between procedures
• Procedural interfaces
  – Sub-system encapsulates a set of procedures to be called by other sub-systems
• Message passing interfaces
  – Sub-systems request services from other sub-systems

Interface errors

• Interface misuse
  – A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
• Interface misunderstanding
  – A calling component embeds assumptions about the behaviour of the called component which are incorrect
• Timing errors
  – The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

• Design tests so that parameters to a called procedure are at the extreme ends of their ranges
• Always test pointer parameters with null pointers
• Design tests which cause the component to fail
• Use stress testing in message passing systems
• In shared memory systems, vary the order in which components are activated

Stress testing

• Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light
• Stressing the system test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data
• Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded

Object-oriented testing

• The components to be tested are object classes that are instantiated as objects
• Larger grain than individual functions so approaches to white-box testing have to be extended
• No obvious ‘top’ to the system for top-down integration and testing
Testing levels

- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

Object class testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised

Weather station object interface

WeatherStation

- Test cases are needed for all operations
- Use a state model to identify state transitions for testing
- Examples of testing sequences
  - Shutdown $\rightarrow$ Waiting $\rightarrow$ Shutdown
  - Waiting $\rightarrow$ Calibrating $\rightarrow$ Testing $\rightarrow$ Transmitting $\rightarrow$ Waiting
  - Waiting $\rightarrow$ Collecting $\rightarrow$ Waiting $\rightarrow$ Summarising $\rightarrow$ Transmitting $\rightarrow$ Waiting

Object integration

- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

Approaches to cluster testing

- Use-case or scenario testing
  - Testing is based on a user interactions with the system
  - Has the advantage that it tests system features as experienced by users
- Thread testing
  - Tests the systems response to events as processing threads through the system
- Object interaction testing
  - Tests sequences of object interactions that stop when an object operation does not call on services from another object

Scenario-based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the interaction between the objects involved
- Consider the scenario in the weather station system where a report is generated
Collect weather data

Weather station testing

- Thread of methods executed
  - CommsController:request → WeatherStation:report → WeatherData:summarise
- Inputs and outputs
  - Input of report request with associated acknowledge and a final output of a report
  - Can be tested by creating raw data and ensuring that it is summarised properly
  - Use the same raw data to test the WeatherData object

Testing workbenches

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs
- Most testing workbenches are open systems because testing needs are organisation-specific
- Difficult to integrate with closed design and analysis workbenches

Testing workbench adaptation

- Scripts may be developed for user interface simulators and patterns for test data generators
- Test outputs may have to be prepared manually for comparison
- Special-purpose file comparators may be developed

Key points

- Test parts of a system which are commonly used rather than those which are rarely executed
- Equivalence partitions are sets of test cases where the program should behave in an equivalent way
- Black-box testing is based on the system specification
- Structural testing identifies test cases which cause all paths through the program to be executed
Key points

- Test coverage measures ensure that all statements have been executed at least once.
- Interface defects arise because of specification misreading, misunderstanding, errors or invalid timing assumptions.
- To test object classes, test all operations, attributes and states.
- Integrate object-oriented systems around clusters of objects.