Static verification

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

Objectives

- To discuss the cost-effectiveness of static verification
- To describe the program inspection process
- To illustrate a mathematical approach to program verification
- To show how static analysis tools may be used
- To describe the Cleanroom software development process

Topics covered

- Program inspection
- Mathematically-based verification
- Static analysis tools
- Cleanroom software development

Static verification

- 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

- 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
- The error detection process is not confused by the existence of previous errors

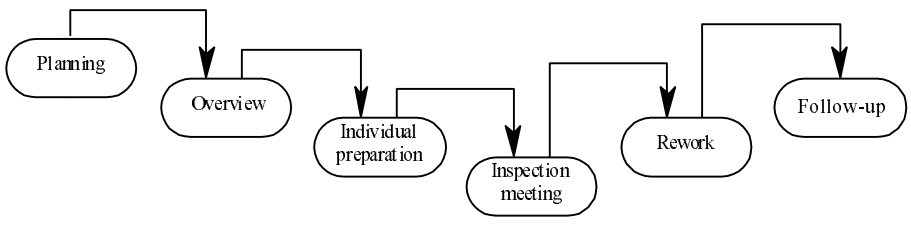
Program inspections

- 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



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
- Reader who reads the code to the team
- Inspector who finds errors, omissions and inconsistencies
- Moderator who chairs the meeting and notes discovered errors
- Other roles are Scribe and Chief moderator

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

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

- Replace with portrait slide

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

Axiomatic approach

- Define pre and post conditions for the program or routine
- Demonstrate by logical argument that the application of the code logically leads from the pre to the post-condition
- Demonstrate that the program will always terminate

Binary search specification

procedure Binary_search (Key : ELEM ; T: ELEM_ARRAY;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
T'LAST - T'FIRST >= 0 **and**
for_all i, T'FIRST <= i <= T'LAST-1, T (i) <= T (i + 1)

Post-condition
(Found **and** T (L) = Key) **or**
(**not** Found **and not** (**exists** i, T'FIRST <= i <= T'LAST, T (i) = Key))

Binary search procedure 1

```
procedure Binary_search (Key: ELEM ; T: ELEM_ARRAY ;
                         Found: in out BOOLEAN ; L: in out ELEM_INDEX ) is
        --  Pre: T'LAST - T'FIRST > 0 and
        --  for_all i, T'FIRST <= i <= T'LAST-1, T (i) <= T (i + 1)
        Bott : ELEM_INDEX := T'FIRST;
        Top : ELEM_INDEX := T'LAST ;
        Mid : ELEM_INDEX;
begin
        L := ( T'FIRST + T'LAST ) mod 2;
        Found := T( L ) = Key;
        -- loop invariant
        -- 1. Found and T(L) = Key or
        --   not Found and not Key in T(T'FIRST..Bott-1, Top+1..T'LAST)
```


Binary search procedure 2

```
while Bott <= Top and not Found loop
        Mid := (Top + Bott) / 2;
        if T( Mid ) = Key then
            Found := true;
            L := Mid;
        --  2. Key = T(Mid) and Found
        elsif T( Mid ) < Key then
            --  3. not Key in T(T'FIRST..Mid)
            Bott := Mid + 1;
        --  4. not Key in T(T'FIRST..Bott-1)
        else
            --  5. not Key in T( Mid..T'LAST )
            Top := Mid - 1;
        --  6. not Key in T(Top+1..T'LAST)
        end if;
  end loop;
--  Post: Found and T (L) = Key or
--        ( not Found and not (exists i, T'FIRST <= i <= T'LAST, T (i) = Key ))
end Binary_search;
```

Termination argument

- While loop terminates if Found or Bott > Top
- If an element = key exists, Found is set true
- In a loop execution either Found is set true, Bott is increased or Top is decreased
- As Top > Bott initially, the effect of loop execution (if Found is false) will always mean that eventually Top-Bott will become negative so Bott > Top and the loop will terminate

Correctness argument

- Loop invariant states that Key does not lie in the portion of the array which has been examined or the value at the mid-point of the array matches Key. True on entry to the loop as none of the array has been examined.
- Assertion 2 follows because of the successful test Key = Mid
- Assertion 3 follows because the array is ordered. If T (Mid) < Key all values up to T (Mid) must also be less than the key

Correctness argument

- Assertion 4 follows by substituting Bott-1 for Mid
- Assertions 5 and 6. Similar argument to 3 and 4
- After loop execution, either the key has been found or there is no value in the array which has been searched which matches the key. However, Bott > Top so all the array has been searched
- Therefore, the binary search routine code conforms to its specification

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

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

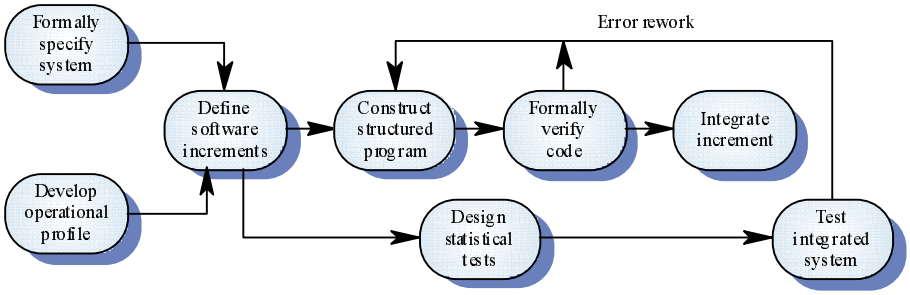
LINT static analysis

- Replace with portrait slide

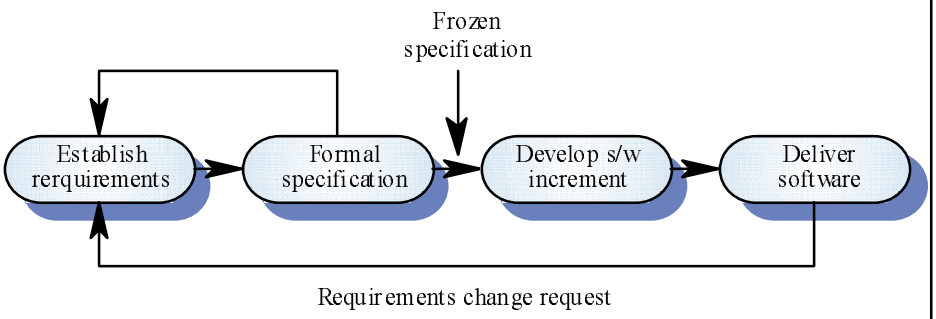
Cleanroom software development

- 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



Incremental development



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 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

Cleanroom process evaluation

- 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

- Static verification is based on source code analysis. Complements rather than replaces program testing
- Program inspections are very effective in discovering errors
- Mathematical verification involves making logical arguments about program correctness
- The axiomatic approach to verification argues correctness from a pre to a post condition

Key points

- 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