Verification and Validation

Dr. James A. Bednar

jbednar@inf.ed.ac.uk

http://homepages.inf.ed.ac.uk/jbednar

Dr. David Robertson

dr@inf.ed.ac.uk

http://www.inf.ed.ac.uk/ssp/members/dave.htm

Verification & Validation

Verification is **getting the system right**:

Does the program you built do what you designed it to do?

Validation is **getting the right system**:

Does the program you designed meet the requirements of the user?

Both tasks are hard — hard to do, and hard to demonstrate to the end user.

V & V Objectives

Correctness : Is the system fault free?

Consistency : Does everything work in harmony?

Sufficiency : Is all of the necessary functionality present?

Performance : Does it do the job well enough?

Necessity : Are there things in it which aren't essential?

V & V Raw Material

Requirements: Typically an informal description of users' needs

Specifications: Formal and/or informal description of properties of the system

Design: Describes how the specifications will be satisfied

Implementation: Realization of the design, e.g. as source code

Changes: History of modifications to correct errors or add functionality

V & V Approaches

Proof of correctness: Formally demonstrate match between program, specifications, and requirements

Testing: Verify a finite list of test cases

Technical reviews: Structured group review meetings

Simulation and prototyping: Testing the design

Requirements tracing: Relating software/design structures (e.g. modules, use cases) back to requirements

Gold Standard: Formal Proofs

The idea of formally proving correctness is appealing:

- Represent problem domain and user requirements in logic, e.g. first-order predicate calculus
- Represent program specifications in logic, and prove that any program meeting these specifications will satisfy the requirements
- Define formal semantics for all the primitives in your programming language
- Prove that for all inputs your program will meet the specifications

Problems With Formal Proofs

Unfortunately, complete formal proofs of large systems are rare and extremely difficult because:

- Converting informal user requirements to formal logic can be as error prone as converting to a program
- Specifications for realistic programs quickly become too complicated for humans or computers to construct proofs
- Developing a logic-based semantics for all languages and components involved in a large system is daunting
- It is difficult to get end users to validate and approve logic-based representations

Practical Formal Methods

"Lightweight" applications of formal methods that do not depend on proving complete program correctness can still be useful (Robertson & Agusti 1999).

E.g. one can formalize some of the trickier parts of the domain and requirements, to clarify what is required before coding.

Also, it can be easier to prove that two programs are equivalent than to prove that either matches a specification. This can help show that a heavily optimized function matches a simpler but slower equivalent.

Cleanroom Software Methodology

Example: Cleanroom is a mostly formal SW development methodology for avoiding defects by developing in an "ultra-clean" atmosphere (Linger 1994). It is based on:

- Formal specification
- Incremental development (perhaps partitioned by modules)
- Structured programming and stepwise refinement (so both structural elements and design choices are constrained)
- Static verification (e.g. using proof of correctness)
- Statistical testing of integrated system, but no unit testing

Testing

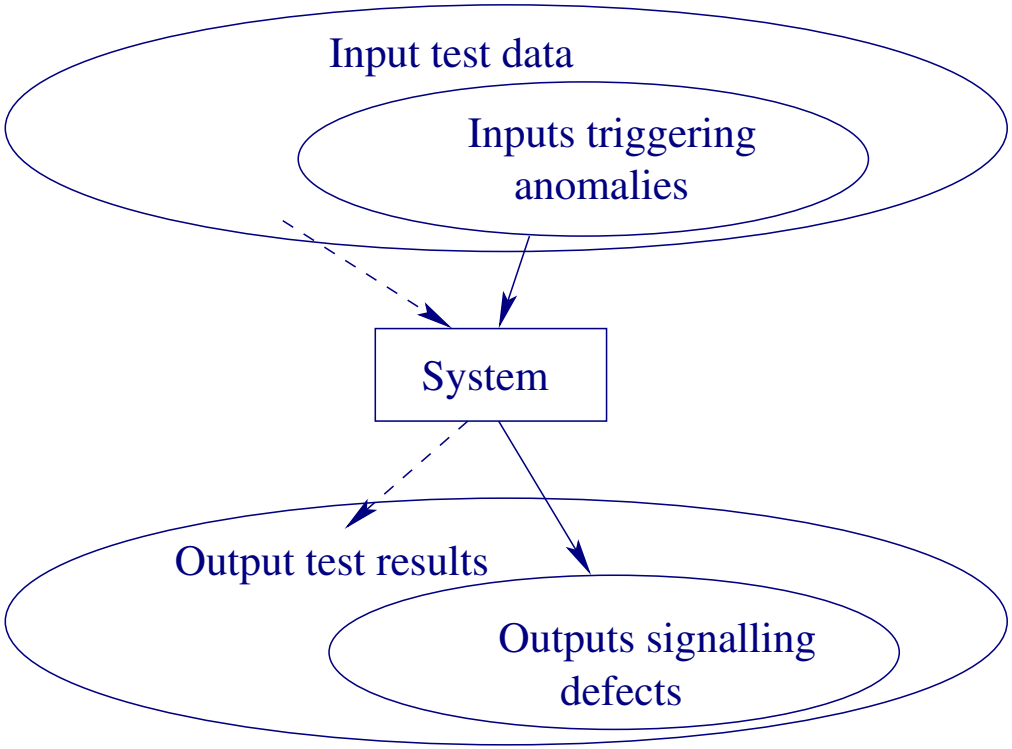
Large, complicated (e.g. GUI-based) systems usually use testing for most V & V, as opposed to formal deduction.

Testing is not straightforward either, because it is usually impractical to test a program on all possible inputs and on all possible execution paths (which are both potentially infinite).

How do we choose which finite and feasibly small set of inputs to test? Consider black-box and clear-box (aka white-box) approaches.

Black-Box Testing (1)

In black-box testing, tests are derived from the program specification, viewing the system as a black box:



- Guess what's inside the box
- Form equivalence partitions over input space

Black-Box Testing (2)

Equivalence partitioning relies on the assumption that we can separate inputs into sets that will produce similar system behaviour.

Then methodically choose test cases from each partition. One method is to choose cases from midpoint (typical) and boundary (atypical) of each partition.

The choice of inputs should not depend on understanding the algorithm inside the box, but on understanding the properties of the input space.

Black-Box Example

For example, suppose we are testing a search algorithm which uses a lookup key to find an element in a (non-empty) array.

One partition of the test cases for this example is between inputs which output a found element and those for which there is no element in the array.

Both of the partitions would then be tested.

Clear-Box Testing (1)

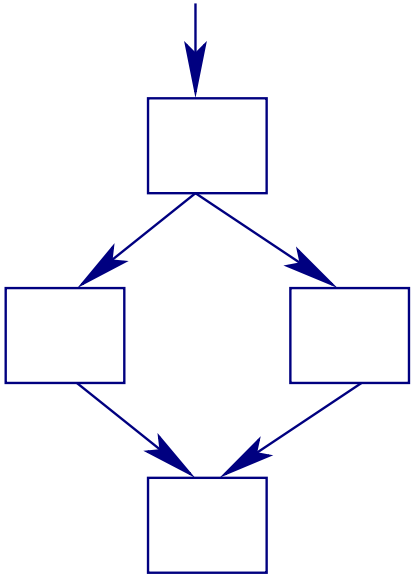
Analyse internal structure of code to derive test data.

Example: Binary search routine (Sommerville 2004)

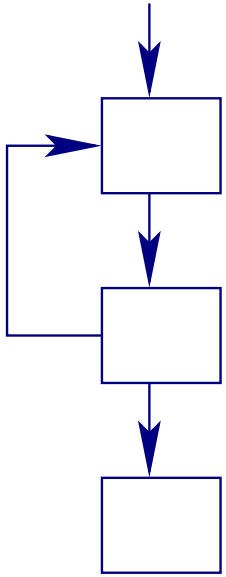
```
void Binary_search (elem key, elem* T, int size, boolean &found, int &L)
{
   int bott=0;
   int top=size - 1;
   int mid=0;
   L = (top + bott) / 2;
   found = (T[L] == key);
   while (bott <= top && !found) {
      mid = top + bott / 2;
      if ( T[mid] == key ) {
         found = true;
         L = mid;
      }
      else if ( T[mid] < key )
        bott = mid - 1;
   }
}
```

Clear-Box Testing (2)

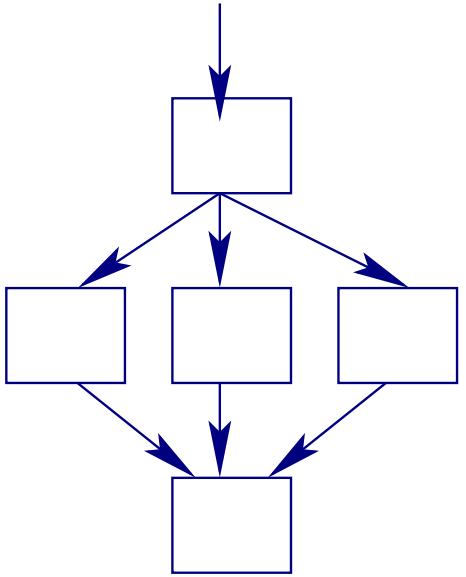
Think of program in terms of flow graphs.



if–then–else



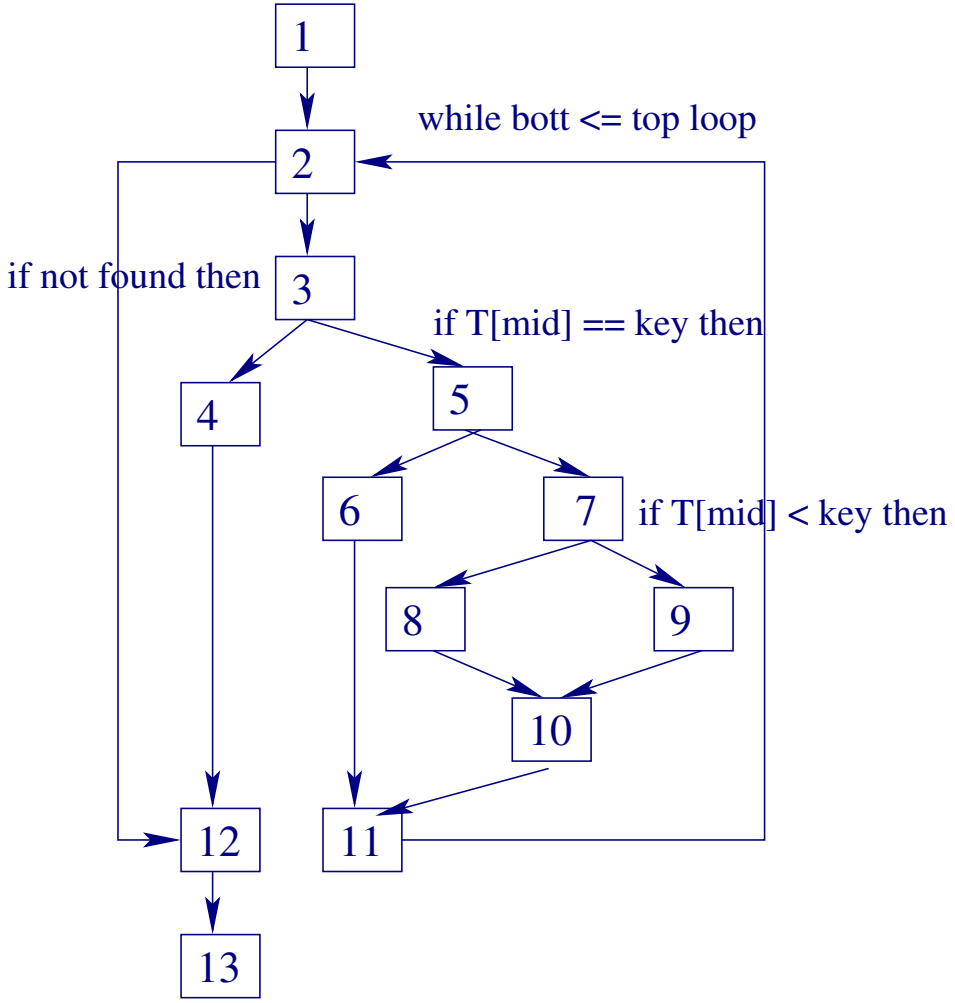
while–loop



case–split

Clear-Box Testing (3)

Now draw a flow graph for the program:



Clear-Box Testing (4)

The paths through this flow graph are:

- 1,2,3,4,12,13
- 1,2,3,5,6,11,2,12,13
- 1,2,3,5,7,8,10,11,2,12,13
- 1,2,3,5,7,9,10,11,2,12,13

If we follow all these paths we know:

- Every statement in the routine has been executed at least once
- Every branch has been exercised for a true/false condition

These tests complement black-box testing.

Testing Levels

Unit testing: Conformance of module to specification

Integration testing: Checking that modules work together

System testing: Concentrates on system rather than component capabilities

Regression testing: Re-doing previous tests to confirm that changes haven't undermined functionality

Unit Testing

It is extremely useful to develop unit tests while developing components (or even before), and to preserve them permanently. Unit tests:

- Help specify what the component should do
- Are faster and simpler for debugging than whole systems
- Preserve test code you would be writing anyway, for obscure cases, bug fixes, etc.
- Help show if a change to the component has broken it
- Provide an example of using the component outside of the system for which it has been developed

Integration Testing

Integration testing focuses on errors in interfaces between components, e.g.:

Import/export type/range errors: some of these can be detected by compilers or static checkers

Import/export representation errors: e.g. an “elapsed time” variable exported in milliseconds and imported as seconds

Timing errors: in real-time systems where producer and consumer of data work at different speeds

Managing Integration Testing

There are numerous ways of organizing an integration testing regime which follows product development:

Top-down: Start with topmost component, simulating lower level components with stubs. Repeat process downwards.

Bottom-up: Start with low level components and place test rigs around these. Then replace test rigs with actual components.

Threaded: Identify major functions and test these, working out from a “backbone” system.

System Testing (ST)

The user cares mainly about the system working as promised, but testing the entire system is even more difficult than testing the components.

Simple system tests are straightforward, but achieving anything like full coverage is extremely difficult.

System tests in e.g. GUI-based systems can be more difficult to automate than unit tests or integration tests.

ST: Transaction Flow Analysis

Identify key “transactions” seen from users’ points of view (like “user stories” in XP, e.g. a request to print a file).

Then follow the paths of consequences of these transactions through the control flow of the program.

Then decide what to test on these paths, e.g.:

- Every link on the path
- Each loop for some number of iterations
- Combinations of paths between transactions
- Looking for unexpected combinations of paths

ST: Stress Analysis

Analyzing the behaviour of the system when its resources are saturated (e.g. for an operating system, request as much memory as the system has available).

First identify which resources should be stressed
(e.g. file space, I/O buffers, processing time).

Then build stress rigs
(e.g. by writing generators for large volumes of data).

Now see what happens when the system is pushed beyond the limits you anticipated.

Regression Testing

Extremely good idea: Build up an automated library of tests that are run regularly to uncover newly introduced bugs.

E.g. most unit tests should go directly into a regression test library, omitting only particularly expensive tests.

Goal is to uncover bugs as soon as possible so that it is clear which changes caused the problems, and so they can be fixed before causing secondary problems.

Some projects run regression tests before allowing code to be checked in (which may be too extreme); others run tests nightly or weekly and mail out the results.

Failure Testing

Most tests are designed to verify normal operation, but it is important to verify correct handling of abnormal cases too:

- Generation of error messages and warnings
- Graceful responses to abnormal inputs
- Responses to failures in distributed components
- Responses to missing libraries or other components
- In mission-critical applications: bugs in some of the system's own components

Such tests can be even trickier to write than typical tests.

Building a V & V Plan

- Identify V & V goals
- Select appropriate techniques at different levels
- Assign organizational responsibilities:
  - Development organization

    (prepares and executes test plans)
  - Independent test organization (runs the tests)
  - Quality assurance organization

    (considers effect on process/product quality)
- Put in place a system for tracking problems uncovered
- Set up logging of test activities

Summary

- Verification is hard, validation even harder
- Complete formal proofs are rare for complex programs
- Useful to test at multiple levels, exhaustively where possible at low levels, strategically at higher levels
- Unit testing and regression testing make component development and maintenance much easier
- Testing is not likely to find all bugs

References

Linger, R. C. (1994). Cleanroom process model. *IEEE Software*, *11* (2), 50–58.

Robertson, D., & Agusti, J. (1999). *Software Blueprints: Lightweight Uses of Logic in Conceptual Modeling*. Reading, MA: Addison-Wesley.

Sommerville, I. (2004). *Software Engineering* (7th Ed.). Reading, MA: Addison-Wesley.