Verification & Validation

Verification is getting the system right. Validation is getting the right system. Both things are difficult to do, and difficult to show that you have done appropriately. Figure 1 depicts one way in which validation and verification is integrated with the lifecycle of software design.

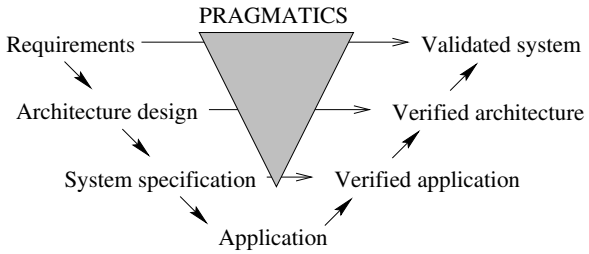

Figure 1: A "V" model of software lifecycle

The raw material for validation and verification of software in lifecycles like the one in Figure 1 is obtained from the following sources:

Requirements : Informal (normally) description of users' needs.

Specifications : Formal/informal description of properties of the system.

Designs : Describe how the specification will be satisfied.

Implementations : Source code (normally) of the system.

Changes : Modifications to correct errors or add functionality.

Using this raw material we then attempt to satisfy a number of objectives:

Correctness : Is the system fault free?

Consistency : Does everything work in harmony?

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

Sufficiency : Is everything essential there?

Performance : Does it do the job well enough?

These objectives are general and it probably will not be possible to prove we have attained them (for example it is very difficult to be sure that traditional software is fault free). Nevertheless, they are the aspirations of validation and verification. Why is attainment so difficult? Here are some reasons:

- Usually it is impractical to test a program on all possible inputs.

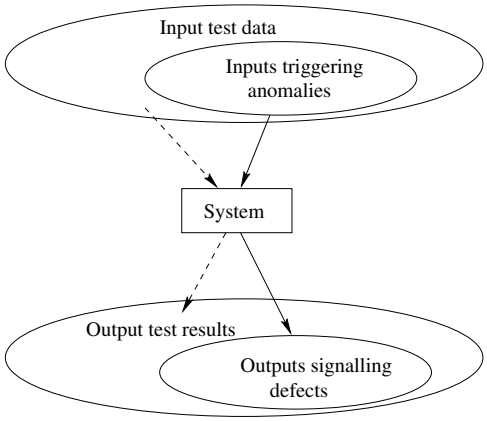

Figure 2: A black box component

- Even if we can enumerate the inputs, it may be impractical to test all execution paths.

- Proofs of equivalence between programs may be easier, but

- that's not the same as proving absolute correctness.

0.1 Black-box Testing

Suppose you have a software component that is too complex, or obscure to analyse by looking at its internal structure - it is a "black box'. You do, however, have some specification of what the component is supposed to do. How, then, do we choose appropriate inputs for testing the component thoroughly? One way is to guess what's inside the box, then form equivalence partitions for inputs. The aim is to isolate a subset of the possible inputs that is just sufficient to trigger all those outputs signaling defects in the component (a concept illustrated by Figure 2).

Equivalence partitioning relies on the assumption that we can separate inputs into sets which will produce similar system behaviour. Then we methodically choose test cases from each partition. For instance, one such method is to choose cases from midpoint (typical) and boundary (atypical) of each partition.

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.

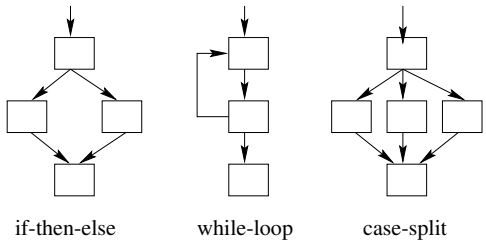

Figure 3: Flow graph structures

0.2 White-box Testing

In black box testing we assumed that we couldn't analyse the code in the component "box". In white box testing we are allowed to look at the internal structure of code to derive test data. This is perhaps best via an example taken from Ian Sommerville's book "Software Engineering". Consider the binary search routine below.

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

If we think of program in terms of flow graphs then we can produce the analysis shown in Figure 4, which uses the structural components shown in Figure 3.

The paths through the flow graph in Figure 4 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

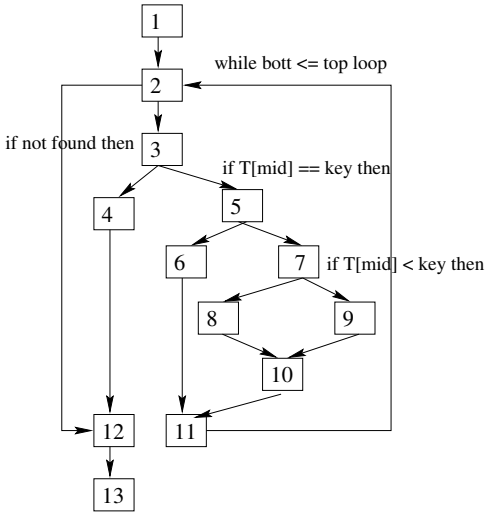

Figure 4: A flow graph for our code example

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.

This doesn't, however, take data complexity into account.

0.3 Levels of Testing

Software systems usually consist of more than one component and testing of the system has also to take into account the system lifecycle, so we have different levels of testing:

Module testing : Local conformance 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.

0.3.1 Making Regression Testing Affordable

It may be prohibitively costly to re-do all our earlier tests during regression testing. If so, we may look for particular types of errors, *e.g.*:

- Data corruption errors (*e.g.* from shared data).
- Control sequencing errors (*e.g.* removing item from a queue before it is placed there).
- Resource contention (*e.g.* deadlocks).
- Performance deficiencies.

Another heuristic for reducing the amount of re-testing is to pay more attention to re-testing older capabilities. Yet another approach is to have baseline tests (always done), augmented with those specific to the modification.

0.3.2 Integration Strategies

The aim of integration testing is primarily to find errors in interfaces between components, *e.g.*:

Import/export type/range errors : some of these can be detected by compilers.

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

Domain errors : when an input follows the wrong path due to incorrect control flow.

Computation errors : input follows the right path but error in assignment causes the wrong function to be computed.

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

There are numerous ways of organising an integration testing regime so that it 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.

0.4 Building a V & V Plan

In order to coordinate the testing of your software system you will want (may be required) to produce a plan of operation. The following are the sorts of tasks you may include in such a plan.

- Identify V & V goals.
- Select appropriate techniques at different levels.
- Assign organisational responsibilities:
  - Development organisation (prepares and executes test plans).
  - Independent test organisation (runs the tests).
  - Quality assurance organisation (considers effect on process/product quality).
- Integrate your techniques within the product lifecycle.
- Put in place a system for tracking problems uncovered.
- Institute a log of test activities.

0.5 Exercise

According to MacKenzie, arguments for "good engineering" are made by deduction, induction or construction. In the context of validation and verification this would mean that our methods involve proving required behaviours follow from our designs (deductively); generalising required behaviours from specific instances of behaviours observed in the software (inductively); or following means of construction which are trusted from experience (in construction). Can all the methods for validation and verification discussed above fit into this way of understanding software engineering?