Verification & Validation

Verification is getting the system right.

Validation is getting the right system.

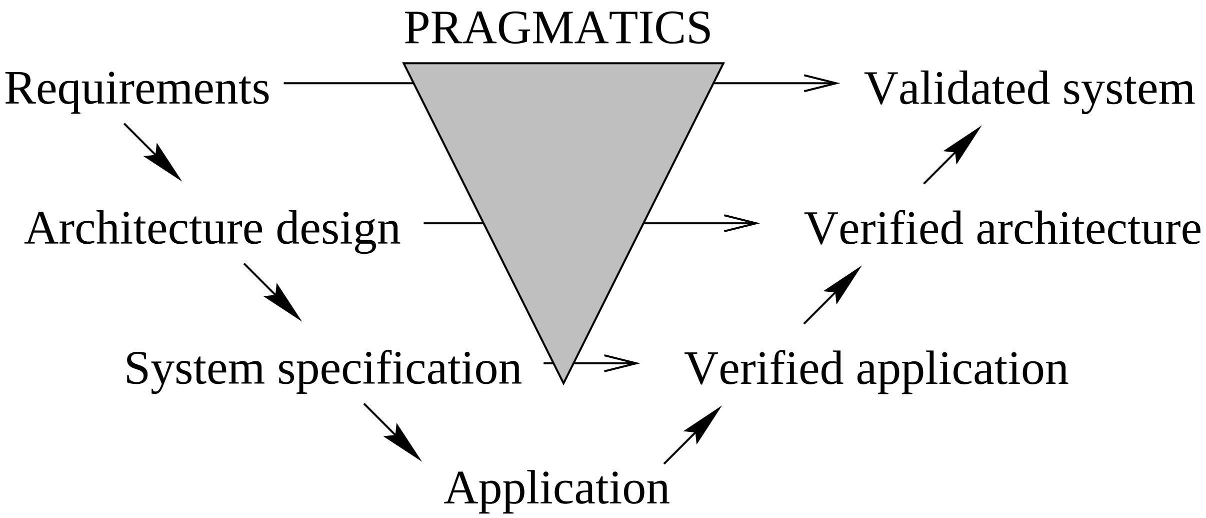
Both things are difficult to do, and difficult to show that you have done appropriately.

We first look at an example of V & V for a very simple system.

Then we consider the problem more generally.

But first, a word about lifecycles ...

V & V in Lifecycles



If arguments for "good engineering" are often by deduction, induction or construction [MacKenzie] then construction currently dominates high levels.

V & V Raw Material

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.

V & V Objectives

Correctness : Is the system fault free?

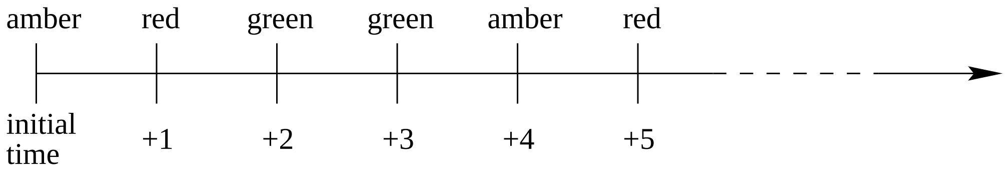
Consistency : Does everything work in harmony?

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

Sufficiency : Is everything essential there?

Performance : Does it do the job well enough?

V & V Example: Traffic Light



What colours are the lights at different time points given specific start states?

Can particular sequences of lights occur at any time?

V & V Example: Testing (1)

$$
\begin{aligned}
light(C, T_i) &\leftarrow init\_time(T_i) \wedge init\_colour(C) && (1)\\
light(red, T) &\leftarrow prev(T, T_p) \wedge light(amber, T_p) && (2)\\
light(green, T) &\leftarrow prev(T, T_p) \wedge light(red, T_p) && (3)\\
light(green, T) &\leftarrow \quad prev(T, T_p) \wedge light(green, T_p) \wedge && (4)\\
&\quad\quad prev(T_p, T_s) \wedge light(red, T_s) &&\\
light(amber, T) &\leftarrow \quad prev(T, T_p) \wedge light(green, T_p) \wedge && (5)\\
&\quad\quad prev(T_p, T_s) \wedge light(green, T_s) &&
\end{aligned}
$$

V & V Example: Testing (2)

$$prev(T, T_p) \leftarrow not(init\_time(T)) \wedge T_p \ is \ T - 1$$

$$init\_time(0) \tag{6}$$

$$init\_colour(amber) \tag{7}$$

V & V Example: Limits of Testing

This goal is easy to test:

$$\exists C.light(C,5) \tag{8}$$

This goal is difficult to test:

$$\exists T,T_b.light(red,T) \wedge before(T,T_b) \wedge light(red,T_b) \tag{9}$$

V & V Example: Adapting Spec

At amber now	$light(amber)$
At red immediately after	$\bigcirc light(red)$
At red i points after initial	$\bigcirc^i light(X)$
Always red after initial	$\Box light(red)$
Sometime red after initial	$\Diamond light(red)$

Combine with normal connectives of FOPC:

$$\Box(\bigcirc light(red) \leftarrow light(amber))$$

Conventional to write this in shorthand as:

$$\bigcirc light(red) \Leftarrow light(amber)$$

V & V Example: Adapted (1)

Specification:

$$\begin{array}{rcll}
\bigcirc light(red) & \Leftarrow & light(amber) & (10) \\
\bigcirc light(green) & \Leftarrow & light(red) & (11) \\
\bigcirc^2 light(green) & \Leftarrow & \bigcirc light(green) \wedge & (12) \\
 & & light(red) & \\
\bigcirc^2 light(amber) & \Leftarrow & \bigcirc light(green) \wedge & (13) \\
 & & light(green) & \\
light(amber) & & & (14)
\end{array}$$

Properties:

$$\bigcirc^5 light(C)$$

$$\Diamond(light(red) \wedge \Diamond light(red))$$

V & V Example: Proof Rules

Goal	Conditions
$A \vdash \bigcirc^i P$	$\left\{ \begin{array}{l} (\bigcirc^i P \leftarrow C) \in A, \\ A \vdash C \end{array} \right\}$
$A \vdash \bigcirc^i P$	$\left\{ \begin{array}{l} (\bigcirc^j P \Leftarrow C) \in A, i \geq j, \\ A \vdash \bigcirc^{i-j} C \end{array} \right\}$
$A \vdash \Diamond(\bigcirc^i P \wedge Q)$	$\left\{ \begin{array}{l} (\Box \bigcirc^j P \leftarrow C) \in A, j \geq i, \\ A \vdash (C \wedge \Diamond \bigcirc^{j-i} Q) \end{array} \right\}$
$A \vdash \Diamond(\bigcirc^i P \wedge Q)$	$\left\{ \begin{array}{l} (\bigcirc^j P \Leftarrow C) \in A, j \geq i, \\ A \vdash \Diamond(C \wedge \bigcirc^{j-i} Q) \end{array} \right\}$
$A \vdash \bigcirc^i P$	$\left\{ \begin{array}{l} (\Box \bigcirc^j P \leftarrow C) \in A, i \geq j, \\ A \vdash C \end{array} \right\}$
$A \vdash \Diamond(\bigcirc^i P \wedge Q)$	$\left\{ \begin{array}{l} (\bigcirc^j P \leftarrow C) \in A, j \geq i, \\ A \vdash (C \wedge \bigcirc^{j-i} Q) \end{array} \right\}$
$A \vdash \Diamond(\bigcirc^i P \wedge Q)$	$\left\{ \begin{array}{l} (\Box \bigcirc^j P \leftarrow C) \in A, i \geq j, \\ A \vdash (C \wedge \Diamond Q) \end{array} \right\}$
$A \vdash \Diamond(\bigcirc^i P \wedge Q)$	$\left\{ \begin{array}{l} (\bigcirc^j P \Leftarrow C) \in A, i \geq j, \\ A \vdash \Diamond(\bigcirc^{i-j} C \wedge Q) \end{array} \right\}$

V & V Example: Proof

$\Diamond(light(red) \land \Diamond light(red))$

4

10

$\Diamond(light(amber) \land \bigcirc\Diamond light(red))$

6

14

$true \land \bigcirc\Diamond light(red)$

equivalent

$true \land \Diamond \bigcirc light(red)$

4

10

$true \land \Diamond light(amber)$

6

14

$true$

V & V Example: Lessons

- Difficult to do extremely convincing V & V, even with highly formal systems.
- Representation and analysis interact.
- Analysis of specification may not transfer to code.
- Nevertheless, we do get insights by formal analysis.

V & V Limitations

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

V & V Approaches

Testing : We saw an example, let's look more closely.

Proof of correctness : We've seen enough for now.

Technical reviews : structured group review meetings.

Simulation and prototyping : Essentially a form of testing.

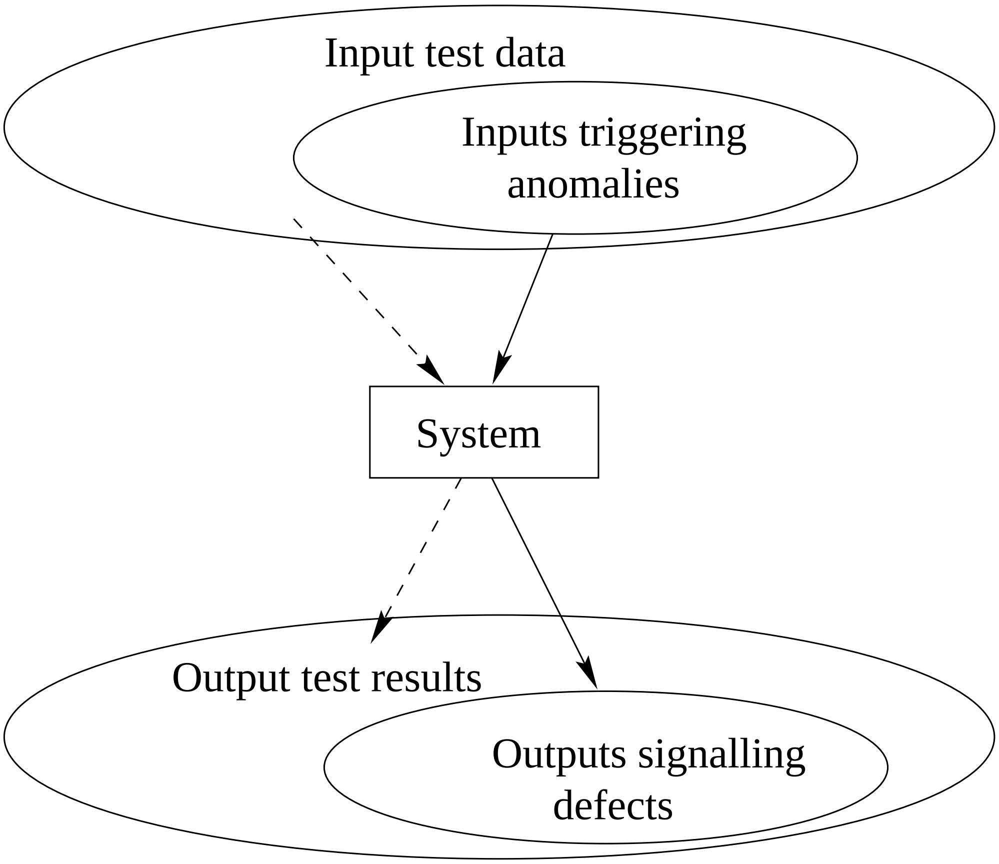
Requirements tracing : Relating requirements to software/design structures (*e.g.* modules, use cases).

Black-box Testing (1)

Tests are derived from the program specification.

System viewed as a black-box.

How do we choose appropriate inputs?



- Guess what's inside the box.
- Form equivalence partitions.

Black-box Testing (2)

Equivalence partitioning relies on the assumption that we can separate inputs into sets which 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.

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.

White-box Testing (1)

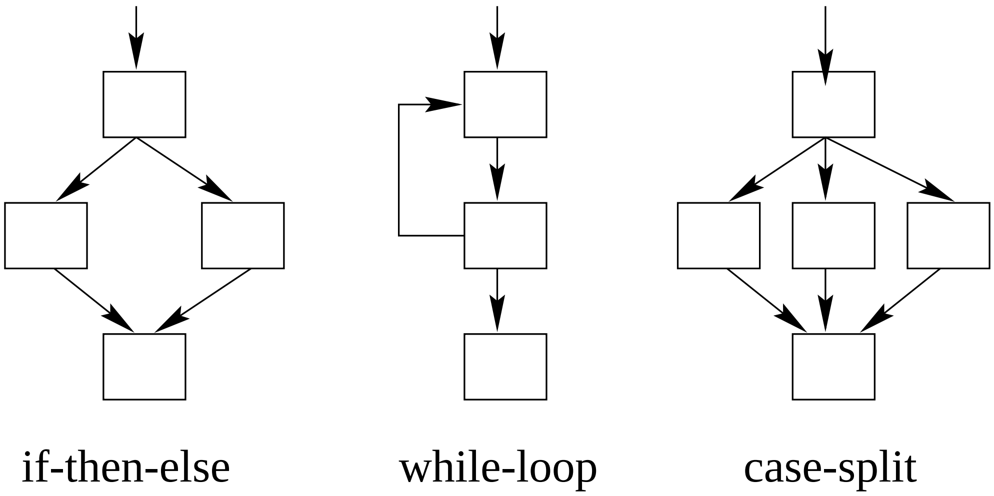
Analyse internal structure of code to derive test data.

Binary search routine from [Sommerville].

```
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  }
```

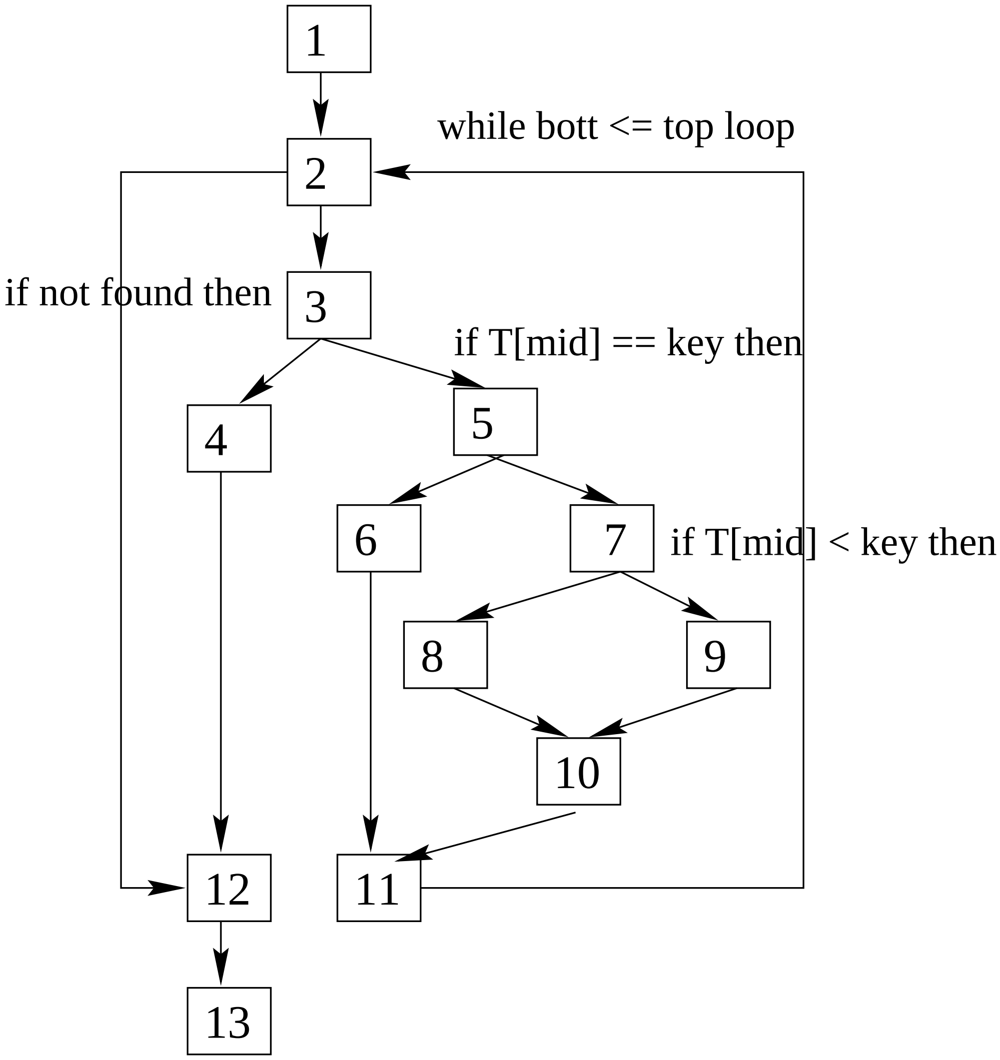
White-box Testing (2)

Think of program in terms of flow graphs.



White-box Testing (3)

Now draw a flow graph for the program.



White-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.

This doesn't take data complexity into account.

Testing Levels

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 functionailty.

Regression Testing

Can't afford to re-do all earlier tests so look for particular 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.

A heuristic is to pay more attention to re-testing older capabilities.

Often have baseline tests, augmented with those specific to the modification.

Integration Strategies (1)

At this stage we are primarily looking for 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.

Integration Strategies (2)

There are numerous ways of organising 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.

Transaction Flow Analysis

Identify key "transactions" seen from users' points of view (*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.

Stress Analysis

Analysing 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.

Failure Analysis

Analysing how the system will react to failures (internal or external).

Often involves a different form of modelling, where incorrect rather than normal operation is being described.

Building a V & V 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.

Cleanroom Software development

Avoid defects by manufacturing in an "ultra-clean" atmosphere.

This needs:

- 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 integratred system.