Test case design techniques I: Whitebox testing
Overview

- What is a test case
- Sources for test case derivation
- Test case execution
- White box testing
- Flowgraphs
- Test criteria/coverage
  - Statement / branch / decision / condition / path coverage
- Loop testing
  - Data flow testing
- Def-use pairs
- Efficiency of different criteria
Types of Testing

- security
- reliability
- robustness
- performance
- user-friendliness
- functional behaviour

- unit
- module
- integration
- system

- white-box
- black-box

- characteristics
- method

level of detail
V - Model

requirements

specification

architecture spec

detailed design

implementation code

acceptance test spec

system test spec

integration test spec

module test spec

unit test spec

acceptance test

system test

integration test

module test

unit-test
What is a Test?

Test Cases

Test Data

Output

Software under Test

Oracle

Correct result?
Development of Test Cases

Complete testing is impossible

Testing cannot guarantee the absence of faults

How to select subset of test cases from all possible test cases with a high chance of detecting most faults?

Test Case Design Strategies
Sources for test case design

- The requirements to the program (its specification)
  - An informal description
  - A set of scenarios (use cases)
  - A set of sequence diagrams
  - A state machine
- The program itself
- A set of selection criteria
- Heuristics
- Experience
White-Box Testing

- Testing based on program code
- Extent to which (source) code is executed, i.e. *Covered*
- Different kinds of coverage:
  - statement coverage
  - path coverage
  - (multiple-) condition coverage
  - decision / branch coverage
  - loop coverage
  - definition-use coverage
  - .....
White box testing: flow graphs

- Syntactic abstraction of source code
- Resembles classical flow charts
- Forms the basis for white box test case generation principles
- Purpose of white box test case generation: Coverage of the flow graph in accordance with one or more test criteria
White-Box : Statement Testing

- Execute every statement of a program
- Relatively weak criterion
- Weakest white-box criterion
Example: Statement Testing

\( result = 0+1+\ldots+|value|, \) if this \( \leq \) \( maxint \), error otherwise

```plaintext
1 PROGRAM maxsum ( maxint, value : INT )
2     INT result := 0 ; i := 0 ;
3     IF value < 0
4        THEN value := - value ;
5     WHILE ( i < value ) AND ( result <= maxint )
6        DO i := i + 1 ;
7        result := result + i ;
8     OD ;
9     IF result <= maxint
10        THEN OUTPUT ( result )
11     ELSE OUTPUT ( “too large” )
12 END.
```
PROGRAM maxsum ( maxint, value : INT )
    INT result := 0 ; i := 0 ;
    IF value < 0
        THEN value := - value ;
    WHILE ( i < value ) AND ( result <= maxint )
        DO i := i + 1 ;
            result := result + i ;
    OD;
    IF result <= maxint
        THEN OUTPUT ( result )
    ELSE OUTPUT ( “too large” )
    END.
Flow graph: Cyclomatic complexity

- \#edges - \#nodes + 2
- Defines the maximal number of test cases needed to provide statement coverage
- Mostly applicable for Unit testing
- Strategy for statement coverage:
  1. Derive flow graph
  2. Find cyclomatic complexity \#c
  3. Determine at most \#c independent paths through the program (add one new edge for each test case)
  4. Prepare test cases covering the edges for each path (possibly fewer than \#c cases)
Example: Statement Testing

Test for complete statement coverage:

<table>
<thead>
<tr>
<th>maxint</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
White-Box : Path Testing

• Execute every possible path of a program,
  i.e., every possible sequence of statements
• Strongest white-box criterion
• Usually impossible: infinitely many paths (in case of loops)
• So: not a realistic option
• But note: enormous reduction w.r.t. all possible test cases
  (each sequence of statements executed for only one value)
Example: Path Testing

Path:

\textit{start}

\texttt{i:=i+1;}
\texttt{result:=result+i;}
\texttt{\ldots}
\texttt{i:=i+1;}
\texttt{result:=result+i;}
\texttt{\ldots}
\texttt{i:=i+1;}
\texttt{result:=result+i;}
\texttt{output(result);}
\texttt{exit}

\texttt{value < 0}

\texttt{\textcolor{red}{value:= -value;}}

\texttt{i:=i+1; result:=result+i;}

\texttt{(i<value) and (result<=maxint)}

\texttt{\textcolor{red}{i:=i+1; result:=result+i;}}

\texttt{result<=maxint}

\texttt{output(result);} \quad \texttt{output("too large");}

\texttt{exit}
White-Box : Branch Testing

- Branch testing == decision testing
- Execute every branch of a program:
  each possible outcome of each decision occurs at least once
- Example:
  - IF b THEN s1 ELSE s2
  - IF b THEN s1; s2
  - CASE x OF
    1 : ....
    2 : ....
    3 : ....
Tests for complete statement coverage:

\[
\begin{array}{c|c}
\text{maxint} & \text{value} \\
10   & -1 \\
0    & -1 \\
\end{array}
\]

is not sufficient for branch coverage;

Take:

\[
\begin{array}{c|c}
\text{maxint} & \text{value} \\
10   & 3 \\
0    & -1 \\
\end{array}
\]

for complete branch coverage.

\[i := i + 1; \quad \text{result} := \text{result} + i;\]

\[\text{value} := -\text{value};\]

\[\text{output}(\text{result});\]

\[\text{output}("\text{too large}");\]
Example: Branch Testing

```
i := i + 1;
result := result + i;
(value < 0) then value := -value;
((i < value) and (result <= maxint)) then i := i + 1; result := result + i;
(output(result);) if result <= maxint then
(output("too large");) else
exit
```

<table>
<thead>
<tr>
<th>maxint</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

But:
No green path!

Needed:
Combination of decisions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>-3</td>
</tr>
</tbody>
</table>
Example: Branch Testing

Sometimes there are infeasible paths (infeasible combinations of conditions)

```plaintext
i:=i+1;
result:=result+i;
value:=-value;
(i<value) and (result<=maxint)
result<=maxint

output(result);
output("too large");
exit
```
White-Box: Condition Testing

- Design test cases such that each possible outcome of each condition in each decision occurs at least once.

- Example:
  - decision \((i < value) \land (result \leq maxint)\)
    - consists of two conditions: \((i < value) \land (result \leq maxint)\)
    - test cases should be designed such that each gets value true and false at least once.
Example: Condition Testing

But \((i = \text{result} = 0)\):

<table>
<thead>
<tr>
<th>maxint</th>
<th>value</th>
<th>i&lt;value</th>
<th>result</th>
<th>(\leq) maxint</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>false</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

gives condition coverage for all conditions

But it does not preserve decision coverage

always take care that condition coverage preserves decision coverage:

decision / condition coverage
White-Box: Multiple Condition Testing

• Design test cases for each combination of conditions

• Example:

  • ( i < value )  (result <= maxint )
    - false  false
    - false  true
    - true   false
    - true   true

• Implies decision-, condition-, decision/condition coverage

• But: exponential blow-up

• Again: some combinations may be infeasible
White-box: loop testing

- Statement and branch coverage are not sufficient
- Single loop strategy:
  - Zero iterations
  - One iteration
  - Two iterations
  - Typical number of iterations
  - \(n-1, n, \) and \(n+1\) iterations (\(n\) maximum number of allowable iterations)
- Nested loop strategy:
  - Single loop strategy often intractable
  - Select minimum values for outer loop(s)
  - Treat inner loop as a single loop
  - Work ‘outwards’ and choose typical values for inner loops
- Concatenated loops:
  - Treat as single, if independent
  - Treat as nested, if dependent
Example: Loop testing

Tests for complete loop coverage:

<table>
<thead>
<tr>
<th>maxint</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
</tr>
</tbody>
</table>
White-box testing: Data Flow criteria

• Basic idea: For each variable definition (assignment), find a path (and a corresponding test case), to its use(s). A pair (definition,use) is often called a DU pair.

• Three dominant strategies:
  • All-defs (AD) strategy: follow at least one path from each definition to some use of it
  • All-uses (AU) strategy: follow at least one path for each DU pair
  • All-du-uses strategy (ADUP): follow all paths between a DU pair

• Complements the testing power of decision coverage
Example: All-uses coverage

```pascal
PROGRAM maxsum ( maxint, value : INT )
    INT    result := 0 ;   i := 0 ;
    IF  value < 0
    THEN   value  :=  - value ;
    WHILE ( i < value ) AND ( result <= maxint )
    DO i  :=  i + 1 ;
        result  :=  result + i ;
    OD;
    IF   result <= maxint
    THEN   OUTPUT ( result )
    ELSE   OUTPUT ( "too large" )
END.
```

Def-use pairs:  
1-3,1-5,1-9,1-4,2-5,2-9,2-6,4-5,6-5,6-9,6-11,6-5-6

Tests for complete all-uses coverage:
maxint value
0 0
0 -1
10 1
10 2
White-Box: Overview

- Statement coverage
- Decision (branch) coverage
- Decision/condition coverage
- Multiple-condition coverage
- Path coverage
- Condition coverage
White-Box: Overview

- Decision (branch) coverage
- All uses coverage
- All defs coverage
- Statement coverage
- All du paths coverage
- Path coverage
Additional techniques: mutation and random testing

- **Mutation testing:**
  - Intended for evaluating the test cases
  - Create a set of slightly modified mutants of the original program containing errors
  - Run the test cases against the mutants
  - Criteria
    - All mutants must fail (strong)
    - All mutants will eventually fail (weak)

- **Random testing:**
  - Basic idea: run the program with arbitrary inputs
  - Inherent problems: How to define the oracle for arbitrary inputs and how to decide to stop?
  - Advantage: The program structure can be ignored
### Efficiency of white-box techniques: two studies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>#test cases</th>
<th>%bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>35</td>
<td>93.7</td>
</tr>
<tr>
<td>Branch</td>
<td>3.8</td>
<td>91.6</td>
</tr>
<tr>
<td>All-uses</td>
<td>11.3</td>
<td>96.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>#test cases</th>
<th>%bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>100</td>
<td>79.5</td>
</tr>
<tr>
<td>Branch</td>
<td>34</td>
<td>85.5</td>
</tr>
<tr>
<td>All-uses</td>
<td>84</td>
<td>90.0</td>
</tr>
</tbody>
</table>
Test case design techniques II: Blackbox testing
Overview

- Black-box testing (or functional testing):
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing
  - Behavioural testing
  - Random testing
  - Error guessing etc…

- How to use black-box and white-box testing in combination
- Basics: heuristics and experience
Black box testing

- SUT
- Input
- Events
- Requirements
- Output

Domain testing
Black-box: Three major approaches

• Analysis of the input/output domain of the program:
  • Leads to a logical partitioning of the input/output domain into ‘interesting’ subsets
• Analysis of the observable black-box behaviour:
  • Leads to a flow-graph-like model, which enables application of techniques from the white-box world (on the black-box model)
• Heuristics
  • Techniques like risk analysis, random input, stress testing
Black-box : Equivalence Partitioning

• Divide all possible inputs into classes (partitions) such that
  • There is a finite number of input equivalence classes
  • You may reasonably assume that
    • the program behaves analogously for inputs in the same class
    • a test with a representative value from a class is sufficient
    • if representative detects fault then other class members
      will detect the same fault
Black-box : Equivalence Partitioning

Strategy :

• Identify input equivalence classes
  • Based on conditions on inputs / outputs in specification / description
  • Both valid and invalid input equivalence classes
  • Based on heuristics and experience
    • “input x in [1..10]” → classes : x < 1, 1 ≤ x ≤ 10, x > 10
    • “enumeration A, B, C” → classes : A, B, C, not{A,B,C,}
  • ……..

• Define one / couple of test cases for each class
  • Test cases that cover valid eq. classes
  • Test cases that cover at most one invalid eq. class
Example: Equivalence Partitioning

• Test a function for calculation of absolute value of an integer

• Equivalence classes:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Valid eq. classes</th>
<th>Invalid eq. Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>nr of inputs</td>
<td>1</td>
<td>0, &gt; 1</td>
</tr>
<tr>
<td>Input type</td>
<td>integer</td>
<td>non-integer</td>
</tr>
<tr>
<td>particular abs</td>
<td>&lt; 0, &gt;= 0</td>
<td></td>
</tr>
</tbody>
</table>

• Test cases:

  x = -10, x = 100
  x = “XYZ”, x = - x = 10 20
A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene (uligesidet), isosceles (ligebenet), or equilateral (ligesidet).

• Write a set of test cases to test this program.
A Self-Assessment Test  [Myers]

Test cases for:

1. valid scalene triangle ?
2. valid equilateral triangle ?
3. valid isosceles triangle ?
4. 3 permutations of previous ?
5. side = 0 ?
6. negative side ?
7. one side is sum of others ?
8. 3 permutations of previous ?
9. one side larger than sum of others ?
10. 3 permutations of previous ?
11. all sides = 0 ?
12. non-integer input ?
13. wrong number of values ?
14. for each test case: is expected output specified ?
15. check behaviour after output was produced ?
Example : Equivalence Partitioning

- Test a program that computes the sum of the first value integers as long as this sum is less than maxint. Otherwise an error should be reported. If value is negative, then it takes the absolute value.

- Formally:

  Given integer inputs maxint and value compute result:

  \[
  \text{result} = \sum_{K=0}^{\lfloor \text{value} \rfloor} k \quad \text{if this} \leq \text{maxint}, \quad \text{error otherwise}
  \]
Example: Equivalence Partitioning

• Equivalence classes:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Valid eq. classes</th>
<th>Invalid eq. classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr of inputs</td>
<td>2</td>
<td>&lt; 2, &gt; 2</td>
</tr>
<tr>
<td>Type of input</td>
<td>int, int</td>
<td>int, no-int, no-int</td>
</tr>
<tr>
<td>Abs(value)</td>
<td>value &lt; 0, value ≥ 0</td>
<td></td>
</tr>
<tr>
<td>maxint</td>
<td>∑ k ≤ maxint,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>∑ k &gt; maxint</td>
<td></td>
</tr>
</tbody>
</table>

• Test Cases:

<table>
<thead>
<tr>
<th>maxint</th>
<th>value</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>55</td>
</tr>
<tr>
<td>100</td>
<td>-10</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>error</td>
</tr>
<tr>
<td>Invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>error</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>“XYZ”</td>
<td>10</td>
<td>error</td>
</tr>
<tr>
<td>100</td>
<td>9.1E4</td>
<td>error</td>
</tr>
</tbody>
</table>
Based on experience / heuristics:

- Testing *boundary conditions* of eq. classes is more effective i.e. values directly on, above, and beneath edges of eq. classes.
- Choose input boundary values as tests in input eq. classes instead of, or additional to arbitrary values.
- Choose also inputs that invoke *output boundary values* (values on the boundary of output classes).
- Example strategy as extension of equivalence partitioning:
  - choose one \((n)\) arbitrary value in each eq. class
  - choose values exactly on lower and upper boundaries of eq. class
  - choose values immediately below and above each boundary (if applicable)
Example: Boundary Value Analysis

- Test a function for calculation of absolute value of an integer

- Valid equivalence classes:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Valid eq. classes</th>
<th>Invalid eq. Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>particular abs</td>
<td>&lt; 0,  &gt;= 0</td>
<td></td>
</tr>
</tbody>
</table>

- Test cases:
  - class \( x < 0 \), arbitrary value: \( x = -10 \)
  - class \( x \geq 0 \), arbitrary value: \( x = 100 \)
  - classes \( x < 0, x \geq 0 \), on boundary: \( x = 0 \)
  - classes \( x < 0, x \geq 0 \), below and above: \( x = -1, x = 1 \)
A Self-Assessment Test  [Myers]

Test cases for:

1. valid scalene triangle?
2. valid equilateral triangle?
3. valid isosceles triangle?
4. 3 permutations of previous?
5. side = 0?
6. negative side?
7. one side is sum of others?
8. 3 permutations of previous?
9. one side larger than sum of others?
10. 3 permutations of previous?
11. all sides = 0?
12. non-integer input?
13. wrong number of values?
14. for each test case: is expected output specified?
15. check behaviour after output was produced?
Example: Boundary Value Analysis

• Given integer inputs \( \text{maxint} \) and \( \text{value} \) compute \( \text{result} \):

\[
\text{result} = \sum_{k=0}^{\text{|value|}} k \quad \text{if this } \leq \text{maxint}, \quad \text{error otherwise}
\]

• Valid equivalence classes:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Valid eq. Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Abs(value)} )</td>
<td>( \text{value} &lt; 0, \ \text{value} \geq 0 )</td>
</tr>
<tr>
<td>( \text{maxint} )</td>
<td>( \sum k \leq \text{maxint}, \ \sum k &gt; \text{maxint} )</td>
</tr>
</tbody>
</table>

• Should we also distinguish between \( \text{maxint} < 0 \) and \( \text{maxint} \geq 0 \)?

| \( \text{maxint} \) | \( \text{maxint} < 0, \ \ 0 \leq \text{maxint} < \sum k, \ \text{maxint} \geq \sum k \) |
Example: Boundary Value Analysis

- Valid equivalence classes:
  
  - \( \text{Abs}(value) \):
    - \( value < 0, \ value \geq 0 \)
  
  - \( \text{maxint} \):
    - \( \text{maxint} < 0, \ 0 \leq \text{maxint} < \sum k, \ \text{maxint} \geq \sum k \)

- Test Cases:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{maxint}</td>
<td>\text{value}</td>
<td>\text{result}</td>
<td>\text{maxint}</td>
<td>\text{value}</td>
<td>\text{result}</td>
</tr>
<tr>
<td>55</td>
<td>10</td>
<td>55</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>10</td>
<td>error</td>
<td>100</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>56</td>
<td>10</td>
<td>55</td>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>

- How to combine the boundary conditions of different inputs? Take all possible boundary combinations? This may blow-up.
Black-box : Cause Effect Graphing

- Black-box testing technique to analyse combinations of input conditions
- Identify *causes* and *effects* in specification
  
  ↓  ↓
  
  inputs outputs
  
  current state new state

- Make Boolean Graph linking causes and effects
- Annotate impossible combinations of causes and effects
- Develop decision table from graph with in each column a particular combination of inputs and outputs
- Transform each column into test case
Black-Box: Cause Effect Graphing

\[ \sum k \leq \text{maxint} \]
\[ \sum k > \text{maxint} \]
\[ \text{value} < 0 \]
\[ \text{value} \geq 0 \]

- Causes:
  - \( \sum k \leq \text{maxint} \)
  - \( \sum k > \text{maxint} \)
  - \( \text{value} < 0 \)
  - \( \text{value} \geq 0 \)

- Inputs:
  - \( \sum k \leq \text{maxint} \)
  - \( \sum k > \text{maxint} \)
  - \( \text{value} < 0 \)
  - \( \text{value} \geq 0 \)

- Effects:
  - \( \sum k \)
  - \( \text{error} \)

<table>
<thead>
<tr>
<th>Causes</th>
<th>( \sum k \leq \text{maxint} )</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum k &gt; \text{maxint} )</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \text{value} &lt; 0 )</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( \text{value} \geq 0 )</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effects</th>
<th>( \sum k )</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{error} )</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Black-box : Cause Effect Graphing

• Systematic method for generating test cases representing combinations of conditions
• Combinatorial explosion of number of possible combinations
• Some heuristics to reduce this combinatorial explosion
• Starting point is effects (outputs) then working ‘backwards’
• ‘light-weight’ formal methods: transformation into semi-formal Boolean graph
• A technique : to be combined with others
Black-box: behavioural specifications

- Many systems are partly specified through the interaction with an environment, e.g.:
  - Phone switches (dialing sequences)
  - Typical PC applications (GUI dialogues)
  - Consumer electronics (mobile phones)
  - Control systems (cruise, navigation)
- Typical specification formalisms:
  - Use cases
  - Sequence diagrams
  - State machines
- In many situations, abstract test cases can be derived directly from such specifications
Example: Use case

One test per use case:
1. Subscribe
2. Place call
3. Answer call
4. Unsubscribe
Example: sequence diagrams

Test:
1. Key-digit
2. Key-digit
3. Key-digit
4. Key-digit
5. key-on } 3 sec

{ 5 sec

The homeowner arms the alarm by entering the correct code then by pressing the on key.

In the code is correct, then after a delay (EXIT_TIME) the alarm will be armed.

During the exit time delay, the Red LED will flash on and off.

During this time, detection of doors opening and of movement in rooms will be ignored.

Once a key has been pressed, has upto five seconds to express the on key.

Once a code has been entered, be pressed within three seconds.

LED will flash at rate of 2Hz.
Example: state machine

Tests:
1. evArm
2. evDoor
3. evDisarm
Black-box: syntax testing

- Many kinds of program inputs are syntax driven, e.g.:
  - Command line input
  - Web forms
  - Language definitions
- Normally, such inputs are analysed by standard parsers, however:
  - Boundary conditions may still be useful to apply in order to check correct error handling
- The techniques for behavioural testing can be used
Syntax testing example

• Commands ::= put | get

Some tests:
1. p,u,t
2. g,e,t
3. q,u,t
4. p,u
5. p,u,s
6. ........
Black-box: random/stochastic

- Basic idea: Drive the system through typical scenarios, extreme scenarios, and rare scenarios in a random way.
- Motivation: Increase the chance of ‘hitting’ system faults.
- Application areas:
  - Systems that run forever in some nondeterministic way, e.g. control systems and communication systems
  - Systems with huge input domains
- Examples:
  - Random mouse clicking/typing towards a GUI.
  - Typical browser-user behaviour: (click;read;)* with a typical random distribution of waiting time
  - Random walk through a specification state model while testing
Black-box: stress testing

- Basic idea: Let the environment behave in an extreme way towards the system in order to identify faults.
- Examples:
  - Emulate an extreme number of web users of a given application
  - Denial of service attacks
  - Push ‘on/off’ on the cars cruise control a number of times followed by a turn-off of the motor and a ‘on’ push.
  - Send a huge amount of buffers on a network connection as fast as possible
  - Power off the washing machine in any state
Black-box: Error Guessing

- Just ‘guess’ where the errors are ....
- Intuition and experience of tester
- Ad hoc, not really a technique
- Strategy:
  - Make a list of possible errors or error-prone situations
    (often related to boundary conditions)
  - Write test cases based on this list
Black-box : Error Guessing

- More sophisticated ‘error guessing’ : Risk Analysis
- Try to identify critical parts of program (high risk code sections):
  - parts with unclear specifications
  - developed by junior programmer while his wife was pregnant ……
  - complex code :
    measure code complexity - tools available (McGabe, Logiscope,…)
- High-risk code will be more thoroughly tested
  ( or be rewritten immediately ….)
Black-Box Testing: Which One?

• Black-box testing techniques:
  • Equivalence partitioning
  • Boundary value analysis
  • Cause-effect graphing
  • Error guessing
  • Test derivation from formal specifications
  • ..........

• Which one to use?
  • None is complete
  • All are based on some kind of heuristics
  • They are complementary
Black-Box Testing: Which One?

- Always use a combination of techniques
  - When a formal specification is available try to use it
  - Identify valid and invalid input equivalence classes
  - Identify output equivalence classes
  - Apply boundary value analysis on valid equivalence classes
  - Guess about possible errors
  - Cause-effect graphing for linking inputs and outputs
White-Box testing: How to Apply?

- Don’t start with designing white-box test cases!
- Start with black-box test cases
  (equivalence partitioning, boundary value analysis, cause effect graphing, test derivation with formal methods, …)
- Check white-box coverage
  (statement-, branch-, condition-, … coverage)
- Use a coverage tool
- Design additional white-box test cases for not covered code
A Coverage Tool: *tcov*

- Standard Unix tool *tcov*
- Only *statement coverage*
- Compile your program under test with a special option
- Run a number of test cases
- A listing indicates how often each statement was executed and percentage of statements executed