Summary of Test Case Design Techniques

Brian Nielsen,
Arne Skou

{bnielsen | ask}@cs.auc.dk
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
Whitebox Testing Overview

- White box testing
  - Flowgraphs
  - Test criteria/coverage
    - Statement / branch / decision / condition / path coverage
    - MC/DC coverage
  - Looptesting
  - Def-use pairs
  - Efficiency of different criteria

**BEWARE**: Expected outcome cannot be determined from code!
White-Box : Statement Testing

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

\[ \text{result} = 0+1+...+|\text{value}|, \text{if this } \leq \text{ maxint, error otherwise} \]

```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.
```
Flowgraph

Node = statement (blocks)
Edges = possible successor (control flow)

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.
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)
Cyclomatic complexity?

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.
Example: Statement Testing

Tests 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)
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:

- **maxint** value
  - 10 -1
  - 0  -1

is not sufficient for branch coverage;

Take:

- **maxint** value
  - 10  3
  - 0  -1

for complete branch coverage
Example: Branch Testing

```
start

value < 0
  yes: value := -value;
  no:
    (i < value) and (result <= maxint)
     yes: i := i + 1; result := result + i;
     no:
       yes: result <= maxint
         yes: output(result);
         no: output("too large");
       no: output("too large");

exit

maxint  value
-1   -1
100   3

But: No green path!

Needed: Combination of decisions
10   -3
```
White-Box: Condition Testing

- Design test cases such that each possible outcome of each condition in each decision occurs at least once
- Example:
  - decision \(( i < \text{value} \) \( \text{AND} \) \((\text{result} \leq \text{maxint})\))
    consists of two conditions: \(( i < \text{value} \) \( \text{AND} \) \((\text{result} \leq \text{maxint})\))
    test cases should be designed such that each gets value true and false at least once
Example: Condition Testing

\[
(i = \text{result} = 0) : \\
maxint \text{ value } i < \text{value} \quad \text{result} \leq \text{maxint} \\
-1 \quad 1 \quad \text{true} \quad \text{false} \\
1 \quad 0 \quad \text{false} \quad \text{true} \\
gives \text{condition coverage for all conditions}
\]

But it does not preserve decision coverage

\[\downarrow\]
always take care that condition coverage preserves decision coverage: \textit{decision / condition coverage}
White-Box: Multiple Condition Testing

- Design test cases for each combination of conditions

- Example:
  - \( (i < \text{value}) \)  \( (\text{result} \leq \text{maxint}) \)
  - 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>15</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

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.

Def-use pairs:
Tests for complete all-uses coverage:
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

maxint value
0 0
0 -1
10 1
10 2
**White-Box: Overview**

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

- Statement coverage
- Decision (branch) coverage
- All defs coverage
- All uses 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>
<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>
What is MC/DC?

- MC/DC stands for Modified Condition / Decision Coverage
- Main idea: Each condition must be shown to independently affect the outcome of a decision, i.e. the outcome of a decision changes as a result of changing a single condition.
1. MC/DC Requirement

- The decision has taken all possible outcomes at least once.

If \(((a) \& (b) \mid (c))\) then...

- We could also say we cover both the true and the false branch (like we do in Branch Testing).
2. MC/DC Requirement

- Every condition in the decision has taken all possible outcomes at least once.

\[
\text{If } (a \& (b) \mid (c)) \text{ then...}
\]

- Aims to cover compound conditions (like Condition/Decision Coverage)
3. MC/DC Requirement

- Every condition in the decision independently affects the decision’s outcome.

If \((a) \& (b) \mid (c)\) then...

Change the value of each condition individually while keeping all other conditions constant.
Creating MC/DC test cases

If \((A \text{ and } B)\) then...

- (1) create truth table for conditions.
- (2) Extend truth table so that it indicated which test cases can be used to show the independence of each condition.

<table>
<thead>
<tr>
<th>A B</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT</td>
<td>T</td>
</tr>
<tr>
<td>TF</td>
<td>F</td>
</tr>
<tr>
<td>FT</td>
<td>F</td>
</tr>
<tr>
<td>FF</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
<th>A B</th>
<th>result</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TT</td>
<td>T</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>TF</td>
<td>F</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>FT</td>
<td>F</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>FF</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating MC/DC test cases cont’d

- Show independence of **A**:
  - Take 1 + 3

- Show independence of **B**:
  - Take 1 + 2

- Resulting test cases are
  - 1 + 2 + 3
  - (T, T) + (T, F) + (F, T)

<table>
<thead>
<tr>
<th>number</th>
<th>A B</th>
<th>result</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T T</td>
<td>T</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>T F</td>
<td>F</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>F T</td>
<td>F</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More advanced example

If (A and (B or C)) then...

<table>
<thead>
<tr>
<th>number</th>
<th>ABC</th>
<th>result</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TTT</td>
<td>T</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TTF</td>
<td>T</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TFT</td>
<td>T</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TFF</td>
<td>F</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>FTT</td>
<td>F</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>FTF</td>
<td>F</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>FFT</td>
<td>F</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>FFF</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: We want to determine the MINIMAL set of test cases.

Here:

{2,3,4,6}
{2,3,4,7}

Non-minimal set is:

{1,2,3,4,5}
Argument in the software industry

- Federal Aviation Administration’s requirement that test suites be MC/DC adequate (DO-178B).

- In the avionics domain, complex Boolean expressions are much more common.

<table>
<thead>
<tr>
<th>N</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6-10</th>
<th>11-15</th>
<th>&gt;15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Tools</td>
<td>446</td>
<td>72</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Booch Components</td>
<td>9048</td>
<td>402</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EFIS avionics display</td>
<td>1343</td>
<td>182</td>
<td>38</td>
<td>16</td>
<td>18</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 4  Complexity of expressions in representative domains
Argument in the software industry

- Federal Aviation Administration’s requirement that test suites be MC/DC adequate.

- Argument – Too expensive
  
  "For example, one of our industrial partners reports that for one of its products of about 20,000 lines of code, the MC/DC-adequate test suite requires seven weeks to run."

- Counter Argument – Significantly more effective
  
  A recent empirical study performed during the real testing of the attitude-control software for the HETE-2 (High Energy Transient Explorer) found that the test cases generated to satisfy the MC/DC coverage requirement detected important errors not detectable by functional testing.
Blackbox Testing Overview

- Black-box testing (or functional testing):
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing
  - Behavioural testing / state transition testing
  - Random testing, Error guessing etc…
  - Others: Classification Tree Method, Pairwise testing, Use Case Testing, Syntax testing, ..

- Basics: heuristics and experience

- How to use black-box and white-box testing in combination
Black box testing

requirements

input

events

SUT

output

domain testing

CISS
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
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 inputs:

- \( x = -10, \ x = 100 \)
- \( x = \text{"XYZ"}, \ x = - \ x = 10, 20 \)
“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 \textit{value} integers as long as this sum is less than \textit{maxint}. Otherwise an error should be reported. If \textit{value} is negative, then it takes the absolute value.

- Formally:

  Given integer inputs \textit{maxint} and \textit{value} compute \textit{result}:

  \[
  \text{result} = \sum_{K=0}^{\lfloor \text{value} \rfloor} K \quad \text{if this} \leq \text{maxint}, \text{ error otherwise}
  \]
### 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 int</td>
</tr>
<tr>
<td>Abs(value)</td>
<td>value &lt; 0, value (\geq 0)</td>
<td></td>
</tr>
<tr>
<td>maxint</td>
<td>(\sum k \leq \text{maxint,})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\sum k &gt; \text{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>
Black-box: Boundary Value Analysis

- 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: $x = -1$, $x = 1$
Example: Boundary Value Analysis

- Given integer inputs `maxint` and `value` compute `result`:

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

- Valid equivalence classes:

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

- Should we also distinguish between `maxint < 0` and `maxint \geq 0`?

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

- Valid equivalence classes:
  
  \[
  \begin{align*}
  \text{Abs}(\text{value}) & \quad \text{value} < 0, \quad \text{value} \geq 0 \\
  \text{maxint} & \quad \text{maxint} < 0, \quad 0 \leq \text{maxint} < \sum k, \quad \text{maxint} \geq \sum k
  \end{align*}
  \]

- Test Cases:

<table>
<thead>
<tr>
<th>maxint</th>
<th>value</th>
<th>result</th>
<th>maxint</th>
<th>value</th>
<th>result</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>-10</td>
<td>1</td>
<td>error</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
  \[ \downarrow \quad \downarrow \]
  \[
  \begin{array}{cc}
  \text{inputs} & \text{outputs} \\
  \text{current state} & \text{new state}
  \end{array}
  \]
- 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

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

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

| Effects     | \( \sum k \)                     | 1 | 1 | 0 | 0 |
| outputs     | error                            | 0 | 0 | 1 | 1 |

Testcase
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: 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
  - ........

- 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 (because requirements must be tested anyway)
  - 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 – maybe combined with a Unit framework
- Design additional white-box test cases for not covered code
Blackbox vs. Whitebox

Specified Behavior (Blackbox)
- I: Specified, not implemented
- II: Implemented and specified
- III: Implemented, but not specified

Implemented Behavior (Whitebox)
- I: Specified, not implemented
- II: Implemented and specified
- III: Implemented, but not specified
A Coverage Tool: gcov

- Standard Gnu tool gcov
- Only statement coverage, but may be applied manually for multiple condition 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
A Coverage Tool: BullseyeCoverage

- Supports *decision/condition*
- Compile your program under test with a special option
- Run a number of test cases
- A listing indicates the coverage