4. Testing
Testing vs. Model Checking

- Testing (usually) means checking the correctness of source code
- Model Checking means verifying the properties of a model given in some formal (not program code) notation
- Attention: things can overlap!
Testing Characteristics

- **Continuity Problem:**
  Small Changes (can) have big effects

- **Context Problem:**
  Environment produces (almost) random effects

- Testing can only prove the presence of defects never their absence (in contrast to model checking or theorem proving)
procedure binary-search (key: in element; table: in elementTable; found: out Boolean) is
begin
  bottom := table'first; top := table'last
  while bottom < top loop
    if (bottom + top) mod 2 ≠ 0 then
      middle := (bottom + top – 1) / 2;
    else
      middle := (bottom + top) / 2;
    end if;
    if key ≤ table (middle) then
      top := middle;
    else
      bottom := middle + 1;
    end if;
  end loop;
  found := key = table (middle);
end binary-search;
Example: Continuity Problem

- When leaving out the first else-statement, the program still works for arbitrary large odd numbers…
**Example: Context Problem**

```plaintext
if x = 0 then
    write ("abnormal")
else
    write ("normal")
end if;
```

- x not initialised
- Effect depends on context (programming language, compiler, operating system)
Some basic notations (1/3)

- Let $P$ be a program, $D$ its input domain, $R$ its range.
- $P$ behaves as a partial function and could be extended to a relation which means $P(d)$ is the result of executing $P$ on input datum $d$.
- Let $OR$ denote the output requirement on output values of $P$ as given in $P$’s specification.
- For a given $d$ in $D$, $P$ is said to be correct for $d$, if $P(d)$ satisfies $OR$.
- $P$ is correct iff it is correct for all $d$ in $D$.
Some basic notations (2/3)

- A **test case** is an element of D. A **test set** T is a finite set of test cases.
- P is correct for T if it is correct for all elements in T (T is **successful** for P).
- A test set T is called **ideal** if, whenever P is incorrect, there exists a d in T such that P is incorrect for d.
- A (test) **selection criterion** is a finite subset of the power set of D, i.e. it specifies a condition which must be satisfied by a test set.
Some basic notations (3/3)

- A selection criterion C is **consistent**, if, for any pair of test sets T1 and T2, both satisfying C, T1 is successful if and only if T2 is successful.
- A selection criterion C is **complete**, if, whenever P is incorrect, there is an unsuccessful test set that satisfies C.
- A selection criterion C1 is **finer** than C2, if, for any P, for every test set T1 satisfying C1, there exists a subset of T1 which satisfies C2.
- Note that consistency and completeness are undecidable problems
White Box vs. Black Box Testing

- **White box** testing means full knowledge of the code when defining test cases

- **Black box** testing means knowledge of the interface (specification) when defining test cases
4.1 White-box testing

**Statement Coverage Criterion**

- Select a test set $T$ such that, by executing $P$ for each $d$ in $T$, each elementary statement of $P$ is executed at least once.
- Elementary statements are assignments, I/O statements and method calls.
Example: Algorithm of Euclid

begin
  read (x); read (y);
  while x ≠ y loop
    if x > y then
      x := x - y;
    else
      y := y - x;
    end if;
  end loop;
  gcd := x;
end;

- While loop suggests two test cases where x=y and x ≠ y
- Statement coverage requires a test set where we consider the cases x < y and x > y.
Construction of minimal test sets

read (x); read (y);
if x > 0 then
  write ("1");
else
  write ("2");
end if;
if y > 0 then
  write ("3");
else
  write ("4");
end if;

- A test set
  \{(x=2, y=3), (x=-13, y=51), (x=97, y=-17), (x=-1, y=-1)\}
  provides statement coverage but is not minimal.
- \{(x=-13, y=51), (x=97, y=-17)\} would suffice
Weakness of statement coverage

\[
\text{if } x < 0 \text{ then} \\
\quad x := -x; \\
\text{end if}; \\
z := x;
\]

- Would only require a test where \( x \) is negative
- But it fails to check the virtual else branch:

\[
\text{if } x < 0 \text{ then} \\
\quad x := -x; \\
\text{else} \\
\quad \text{null;} \\
\text{end if}; \\
z := x;
\]
**Edge Coverage Criterion**

- Select a test set $T$ such that, by executing $P$ for each $d$ in $T$, each edge of $P$'s control flow graph is traversed at least once.
Construction of the control flow graph

- For each elementary statement (an assignment, an I/O statement or a method call, for example) a graph with two nodes and an edge is built, see Figure (a).
- An edge represents the statement.
- Nodes represent entry into and exit from statement.
- Labels on statements and edges can be used to make correspondence explicit.
Construction of the control flow graph

- Let $S_1$ and $S_2$ denote two statements, and $G_1$ and $G_2$, denote their corresponding graphs.
- Then, the graph of Figure (b) is associated with the statement

  ```
  if \text{cond} \text{ then }
  S_1;
  \text{else}
  S_2;
  \text{end if};
  ```

- The graph of Figure (c) is associated with the statement

  ```
  if \text{cond} \text{ then }
  S_1;
  \text{end if};
  ```
Construction of the control flow graph

- Let S1 and S2 denote two statements, and G1 and G2, denote their corresponding graphs.

- Then, the graph of Figure (d) is associated with the statement

  ```
  while cond loop
  S1;
  end loop;
  ```

- The graph of Figure (e) is associated with the statement

  ```
  S1;
  S2;
  ```
Weakness of edge coverage

found := false;

if number_of_items ≠ 0 then
    counter := 1;
    while (not found) and counter < number_of_items loop
        if table (counter) = desired_element then
            found := true;
            end if;
            counter := counter + 1;
        end loop;
    end if;

if found then
    write ("the desired element exists in the table");
else
    write ("the desired element does not exist in the table");
end if;
Weakness of edge coverage

- Test Cases:
  - A table with no items, i.e., `number_of_items = 0`  
    $\Rightarrow$ execution of the fragment without any iteration of the loop body.
  - A table with three items, the second being the desired one  
    $\Rightarrow$ execution of the loop body twice, the first time without executing the `then` branch, the second time executing it.
  Notice that both branches of the final `if-then-else` statement are executed, so that the edge coverage criterion is fulfilled.

- The cases provide edge coverage
- But these test cases miss the problem that the counter can be equal to `number_of_items`
Condition Coverage Criterion

- Select a test set \( T \) such that, by executing \( P \) for each element in \( T \), each edge of \( P \)'s control flow graph is traversed, and all possible values of the constituents of compound boolean conditions are exercised at least once.

- In the example on slide 20, test cases are needed where all values for both conditions \(!found \) and \( counter < number_{of\_items} \) are tested.
Weakness of condition coverage

```plaintext
if x ≠ 0 then
  y := 5;
else
  z := z - x;
end if;
if z > 1 then
  z := z / x;
else
  z := 0;
end if;
```

Possible test sets:
- {(x = 0, z = 1), (x = 1, z = 3)} provides for condition (and edge(!)) coverage …
- but division by zero is possible
- {(x = 0, z = 1), (x = 1, z = 3), (x = 0, z = 3), (x = 1, z = 1)}
Path coverage criterion

- Select a test set $T$ such that, by executing $P$ for each $d$ in $T$, all paths leading from the initial to the final node of $P$'s control flow graph are traversed.

Paths to provide for Edge Coverage

Additional paths to provide for Path Coverage
Empirical Guidelines for testing loops

1. Zero times

2. A maximum number of times
   (does not work for reactive systems which contain infinite loops, see model checking)

3. An average number of times
4.2 Black-box testing

Blackbox Testing based on Equivalence Classes

_Idea:_

Classes of correct inputs

Classes of incorrect inputs

Component

Classes of outputs
Example

- Only the interface, preconditions and commentary are known

```plaintext
procedure binary-search (key: in element;
                        table: in elementTable; found: out Boolean)
```

- Commentary
  - „Binary search in an array“

- Preconditions:
  - Last index of table > first index of table.
  - The table is ordered.
  - The first element is at index 0.
Possible equivalence classes

- Preconditions suggest four equivalence classes
  1. Inputs conform to pre-conditions and element is in the array
  2. Inputs conform to pre-conditions and element is not in the array
  3. Inputs do not conform to pre-conditions and element is in the array
  4. Inputs do not conform to pre-conditions and element is not in the array

- Classes 3 and 4 may be combined as the program may return an error message if the preconditions are not met
Possible equivalence classes

- Use of a sorted array suggests three equivalence classes:
  1. Array contains one element
  2. Array contains even number of elements
  3. Array contains odd number of elements ($\neq 1$)

- Boundary values should be tested which suggests two equivalence classes:
  1. Element is the first element in the array
  2. Element is the last element in the array
The combined Equivalence Classes

1. Array of length 1, element in array
2. Array of length 1, element not in array
3. Array of even length, element 1st element in array
4. Array of even length, element last element in array
5. Array of even length, element is not first or last one
6. Array of even length, element not in array
7. - 10. analogous with array of odd length
11. Precondition does not hold
Summary: Equivalence Testing

- Define Equivalence Classes for all possible test cases based on functional specification
- Take one test case for each class
- Consider correct and incorrect inputs
- Consider boundary values
Decision table-based testing

Consider the following informal specification for a word processor:

- The word processor may preset portions of text in three different formats: plain text (p), boldface (b), and italics (i).
- The following commands may be applied to each portion of text: make text plain (P), make boldface (B), make italics (I), emphasize (E), super-emphasize (SE).
- Commands are available to dynamically set E to mean either B or I. (We denote such commands as E = B and E = I, respectively.)
- Similarly, SE can be dynamically set to mean either B (command SE = B), or I (command SE = I), or B and I (command SE = B+I).

### Decision table for the example

<table>
<thead>
<tr>
<th>Conditions</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>p</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
</tr>
<tr>
<td>I</td>
<td>i</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>SE</td>
<td></td>
</tr>
<tr>
<td>E=B</td>
<td></td>
</tr>
<tr>
<td>E=I</td>
<td></td>
</tr>
<tr>
<td>SE=B</td>
<td></td>
</tr>
<tr>
<td>SE=I</td>
<td></td>
</tr>
<tr>
<td>SE=B&amp;I</td>
<td></td>
</tr>
</tbody>
</table>

#### Condition alternatives
- Example: If super-emphasize is selected and set to B&I the text is printed bold and in italics.

#### Action entries
- Action entries
The cause-effect graph technique

The following (incomplete) cause-effect graph shows under which circumstances a word is printed bold.

B
I
P
E
E = B
SE
E = I
SE = B
SE = I
SE = B+I

Cause-effect graphs can be built systematically from decision tables: Combine all conditions in each column with an AND node and connect all causes leading to the same action with an OR node, which is connected to the action.
Graphical representation of constraints among local variables

For causes a, b, c and effects x, y the following constraints can be used:

- **E** (exclusive) - At most one must be true:
  - a
  - b
  - c

- **I** (at least one) - At least one must be true:
  - a
  - b
  - c

- **O** (one and only one) - Only one must be true:
  - a
  - b
  - c

- **R** (requires) - a implies b:
  - a
  - b

- **M** (mask) - x implies not y:
  - x
  - y
The cause-effect graph with constraints