



Software Testing III

Prof. Dr. Holger Schlingloff

Humboldt-Universität zu Berlin

and

Fraunhofer Institute of Computer Architecture
and Software Technology FIRST

Outline of this Lecture Series

- 2006/11/24: Introduction, Definitions, Examples
 - 2006/11/25-1: Functional testing
 - **2006/11/25-2: Structural testing**
 - 2006/11/26-1: Model-based test generation
 - 2006/11/26-2: Specification-based test generation
-
- Next week: Your turn!

Structural Testing

- Main problem addressed
 - **How many test cases are needed?**
(e.g. for certification issues, for financial estimations, for project planning, ...)
- Resorting to the SUT structure (the program text)
 - construction of test cases from or with reference to the program
 - estimation of test coverage with respect to the program text
- Test coverage = number of reached goals / number of set goals
 - e.g. 100% of all statements are executed at least once
 - e.g. 30% of all loops are traversed at least three times

Basis for Structural Tests

- control flow oriented
 - starting point: control flow graph of a program
 - test case: path through the graph
- data flow oriented
 - starting point: access of variables in a program
 - test case: pair of writing and reading



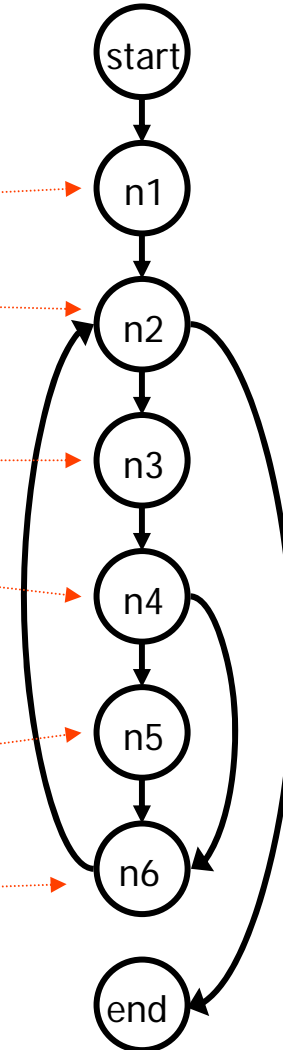
! structural testing can not find missing features !

control flow graph

```

void countChar (int& vocNumber,
               int& totalNumber){
    char chr;
    cin >> chr;
    while ((chr >= `A`) &&
           (chr <= `Z`) &&
           (totalNumber < INT_MAX)){
        totalNumber += 1;
        if ((chr == `A`) ||
            (chr == `E`) ||
            (chr == `I`) ||
            (chr == `O`) ||
            (chr == `U')){
            vocNumber += 1;
        }
        cin >> chr;
    }
}

```



Ex. taken from Liggesmeyer (f) / Balzert

Coverage

- Criteria
 - statement coverage
 - branch coverage
 - condition coverage
 - simple condition coverage
 - multiple condition coverage
 - minimal condition coverage
 - path coverage

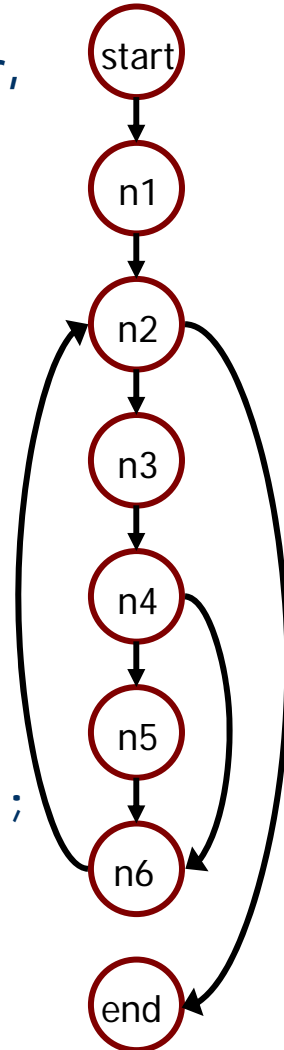
Statement Coverage

- C_0 -Test
- Each program statement must be executed in at least one test case
- Example: (A, 1) yields path (start, n1, n2, n3, n4, n5, n6, n2, end)
- Edge (n4, n6) is not traversed!

```

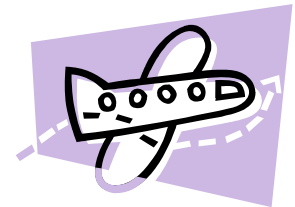
void countChar (int& vocNumber,
int& totalNumber){
char chr;
cin >> chr;
while ((chr >= `A `) &&
(chr <= `Z `) &&
(totalNumber < INT_MAX)){
totalNumber += 1;
if ((chr == `A `) ||
(chr == `E `) ||
(chr == `I `) ||
(chr == `O `) ||
(chr == `U `)){
vocNumber += 1;
}
}
cin >> chr
}
}

```



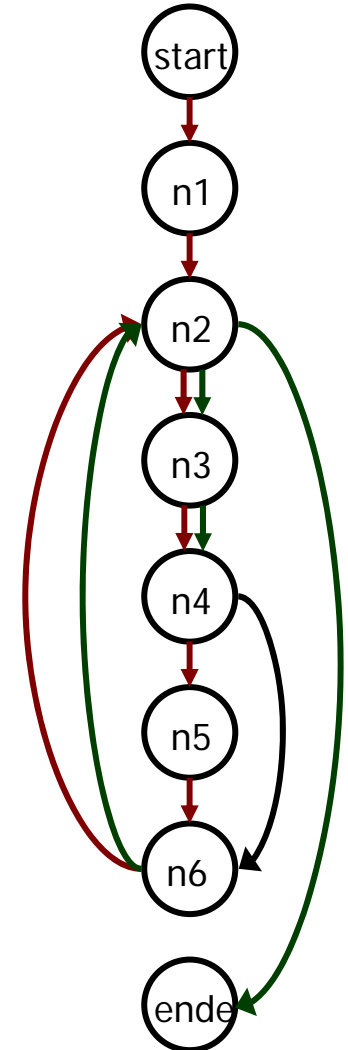
Critique Statement Coverage

- Often „*complete statement coverage*“ is the absolutely minimal criterium for the construction of a test suite
 - in theory it is an undecidable problem whether a certain statement is reachable at all!
- Percentage measured in number of reached / number of all program statement; (desirable: 100%)
- e.g. in DO-178B (above level C)
- often used, easy to calculate
- weak criterion (18% discovered errors)
- e.g. $(x > 5)$ for $(x \geq 5)$ not discovered

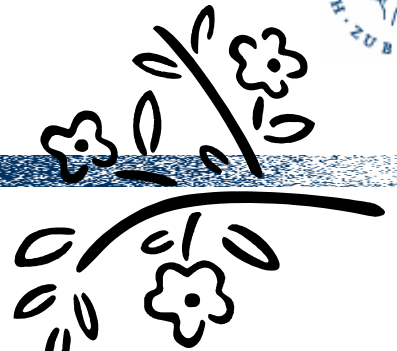


Branch Coverage

- C_1 -Test
- each edge between two nodes must be traversed at least once
- e.g. (A,B,1) yields path (start, n1, n2, n3, n4, n5, n6, n2, n3, n4, n6, n2, end)
- Coverage: Percentage of traversed edges



Critique Branch Coverage



- subsumes statement coverage
- Still, loops are insufficiently tested (e.g. only once)
- Each branching condition must be true in one and false in another test case
- Edges can be weighted to correlate the coverage with the number of test cases

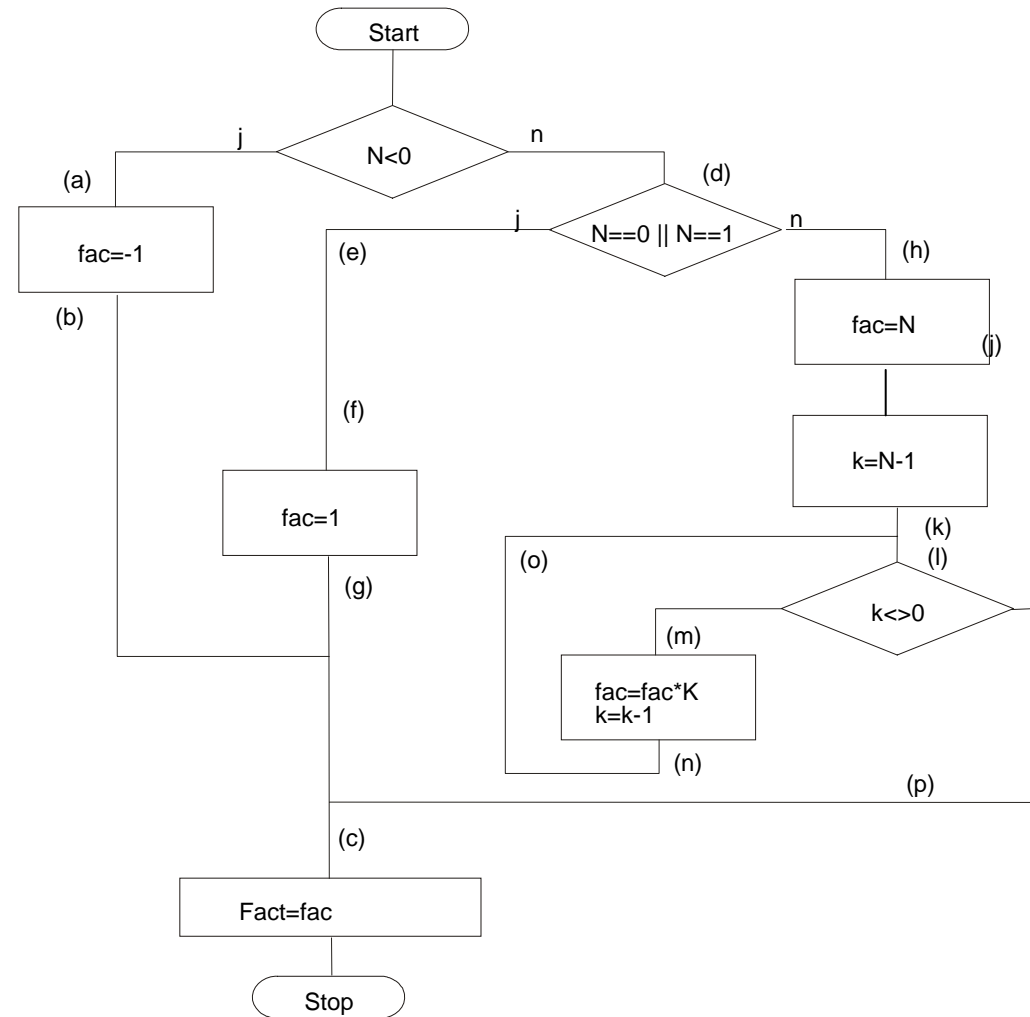
- well suited to find logical errors, not so well for data errors
- easy to implement with tool support (code annotation)

Exercise / Homework

```

int Fact (int N){
    if (N<0){
        fac=-1
    }
    else if (N==0 || N==1){
        fac=1
    }
    else {
        fac=N; K=N-1;
        while(k<>0){
            fac=fac*k;
            k=k-1
        }
    }
    Fact=fac;
}

```



Condition Coverage

- Decisions in the program text
- Several variants:
 - simple condition coverage (C_2): each atomic condition must be true at least once and false at least once
 - multiple condition coverage (C_3 or $C_2(M)$): all combinations of atomic conditions
 - minimal condition coverage: each decision in the flow graph

```
void countChar (int& vocNumber,  
int& totalNumber){  
    char chr;  
    cin >> chr;  
    while ((chr >= `A `) &&  
           (chr <= `Z `) &&  
           (totalNumber < INT_MAX)){  
        totalNumber += 1;  
        if ((chr == `A `) ||  
            (chr == `E `) ||  
            (chr == `I `) ||  
            (chr == `O `) ||  
            (chr == `U `)){  
            vocNumber += 1;  
        }  
        cin >> chr  
    }  
}
```

simple condition coverage

- each atomic condition must be true at least once and false at least once
 - e.g. $(p \ \& \ q \ || \ r)$ yields six test cases
- condition coverage is often combined with branch coverage , so called condition/decision coverage
- problem: could be compiler dependent! (incomplete evaluation of conditions)
- problem: how to control the program flow such that it yields the required conditions (dependent variables)
- problem: total condition might be always the same

Multiple Condition Coverage

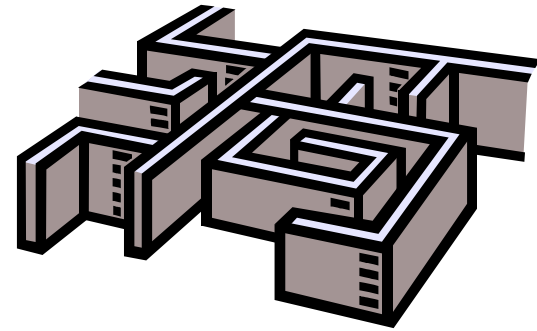
- all variations of atomic conditions
 - e.g. $(p \ \& \ q \ || \ r)$ yields eight test cases
- total decision is guaranteed to vary
- exponentially many possibilities
- problem: possible dependencies of variables!
(e.g. $(chr == \text{'A'}) \ || \ (chr == \text{'E'})$) can not both be true)
- not a feasible coverage criterion!

Minimal Condition Coverage

- Evaluation with respect to the structure of the formula (each subformula true and false)
- compound decisions will be evaluated compoundly
- **problem: $((A \& \& B) \mid \mid C)$ is covered by (www) and (fff), but not tested satisfactorily**
- **Modification: additional proof that each atomic decision is relevant (e.g. by one positive and one negative test case)**
- in combination with branch coverage used for flight critical software (MC/DC); **most important coverage criterion to date**

Path coverage

- Each path through the control flow graph
- In general there are infinitely many paths!
(Coverage?)
- even if loops are restricted: „very many“
- structured path coverage: equivalence classes of paths (similar to boundary values)
 - each loop executed no time, one time, more than two times (*boundary* or *interior* condition)
- additionally manual constructed test cases
- Tool support?



Coverage Tools



Bullseye Coverage - The Authority in C++ Code Coverage

Semantic Designs: Test Coverage - SEMANTIC DESIGNS, INC. Test Coverage Tools

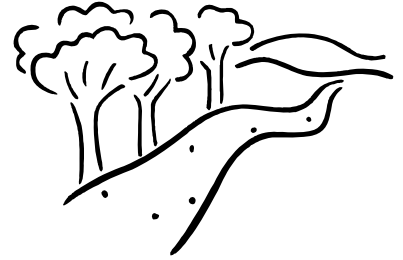
McCabe Associates - McCabeTest - Analyze the Effectiveness

Java-Source.net - Open Source Code Coverage Tools in Java

Verifysoft - CTC++ Test Coverage Analyser

Data Flow Oriented Testing

- Variables and parameter
 - life cycle:
create – (write – read*)* – destruct
 - computational use vs. predicative use
 - Assignment of data flow attributes to the nodes of the control flow graph
- Calculation of variable uses
 - for each node n defining a variable x calculate the sets $c\text{-use}(x,n)$ and $p\text{-use}(x,n)$ of all nodes n' where x is used (read or written)

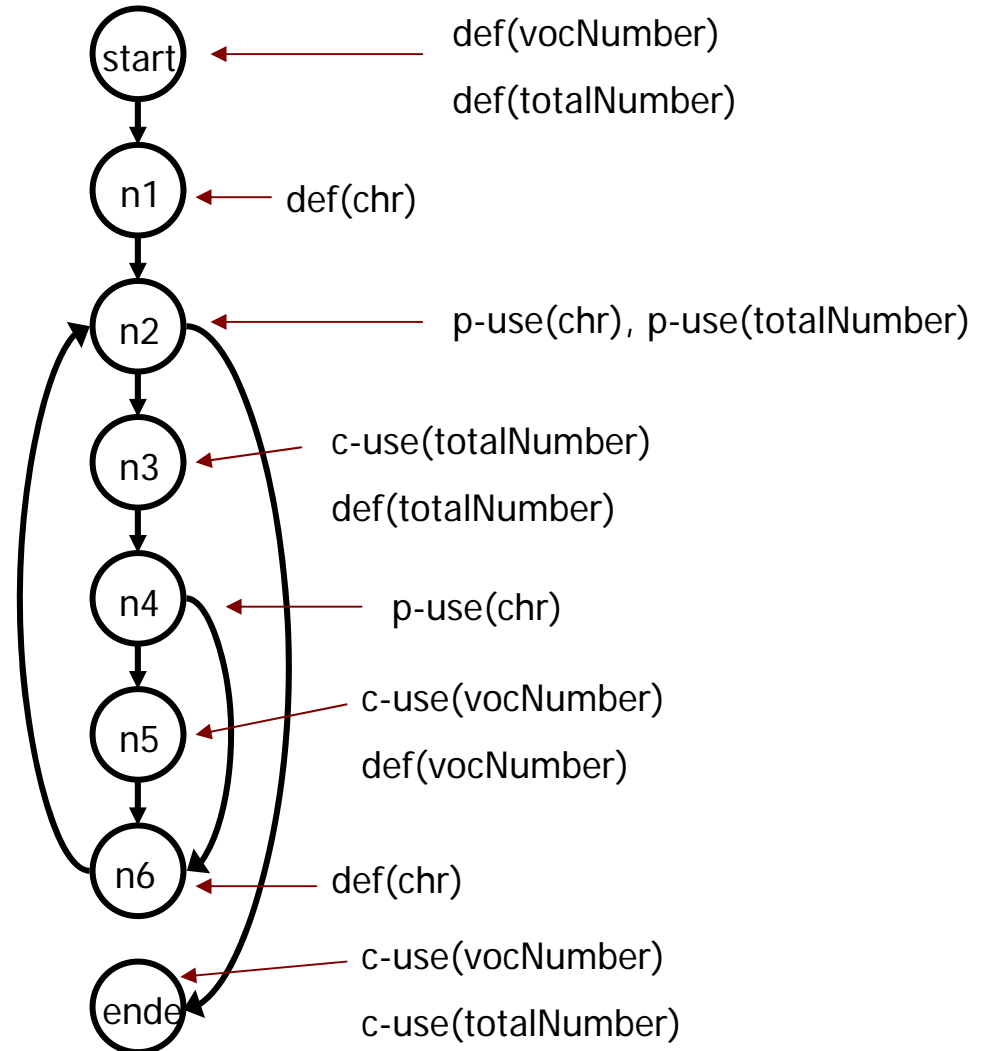


Attributed Control Flow Graph

```

void countChar (int& vocNumber,
               int& totalNumber){
    char chr;
    cin >> chr;
    while ((chr >= `A`) &&
           (chr <= `Z`) &&
           (totalNumber < INT_MAX)){
        totalNumber += 1;
        if ((chr == `A`) ||
            (chr == `E`) ||
            (chr == `I`) ||
            (chr == `O`) ||
            (chr == `U`)){
            vocNumber += 1;
        }
        cin >> chr
    }
}

```



Defs/Uses-Criteria for Test Coverage

- Test case generation
 - compute all paths between definition and use
 - unused definitions mark errors
- Criteria
 - *all defs*: Each path from a definition to at least one use
 - *all p-uses*: Each path from a definition to some predicative use
 - subsumes branch coverage
 - *all c-uses*: analog with computational use
 - *all uses*: each use
 - *du-paths*: Restriction to repetition-free paths

Critique Data Flow Coverage

- More powerful than control flow methods
- Well suited for object-oriented programs
- *all c-uses* better than *all p-uses* better than *all-defs*
- Tool support essential
- Not enough tools available

Critique Structural Testing

- Omission errors (including missing exception handling etc.) can not be detected
- Construction of test cases may be arbitrarily difficult
- „dead code“ (never executed) is normally detected
- Tool support necessary
- Possibility to optimise code by improving frequently used code fragments; regression test necessary!
- Literature: *How to Misuse Code Coverage*; Brian Marick;
<http://www.testing.com/writings/coverage.pdf>

Integration Testing

- Goal: Systematic exploration of the interaction between modules
- „higher level module test“
- Increasingly important, since programming often means composition of predefined modules
- White-Box on the level of modules
 - components and connections are visible
 - internals of components cannot be accessed
- Precondition: all elementary modules are well-tested and assumed to be error-free
 - newly detected faults are likely to be caused by incompatibilities between module interfaces
- Method: stubs and driver

Stubs

- A stub is a pseudo-module emulating the functionality of a not yet implemented or integrated component
- Implementation possibilities
 - e.g. return of a constant value instead of a calculated („correct“) function value
 - e.g. show the input values and prompt the tester for a value to return
 - e.g. synthesize or implement with little effort a rapid prototype which gives some „approximative“ value

Driver and oracles

- A test driver is a module which calls a module of the SUT and provides the input data for it
 - cf. `public class IMathTest extends TestCase`
- Oracle problem: decision whether the return values are correct
 - in general non computable (manual assignment of test verdict)
 - often trivially solvable (simple equality check), thus automatic evaluation of test results within the test driver
- Examples for solvable and unsolvable oracle problems
 - test of an addition function
 - send „i“, check whether the i^{th} Turing machine terminates
 - is the given program text correctly compiled?
 - is the calculation finished within 1 ms?

Results of Integration Testing

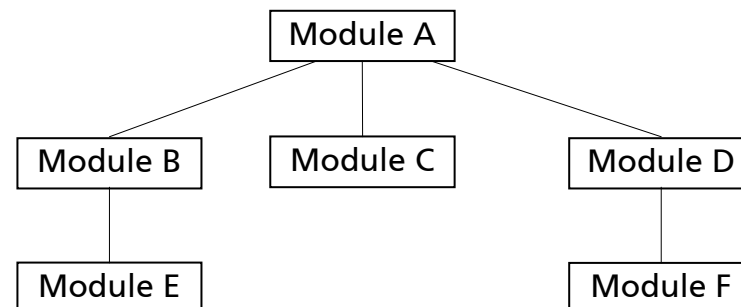
- Typical errors during SUT integration
 - undocumented side effects of some modules
 - e.g. certain „blank“ data fields being used
 - memory leaks due to operating systems properties
 - deadlocks, synchronisation problems (e.g. Mars Polar Lander)
 - timing
- Often additional „glueware“ is being used for integration which had not been included in module testing
 - additional potential for errors

Integration Testing How-to (1)

- Integration methods: Big-Bang, Top-down, Bottom-Up, Sandwich
- Big-Bang
 - all individual components are assembled and tested on the same day
 - sounds adventurous? Sometimes no other possibilities exist (e.g. availability of specific resources, organisational reasons)
 - localisation of faults problematic
- Top-Down
 - first, stubs for all components are prepared
 - then, the top-level module is being tested with stubs
 - iteratively, stubs are replaced by „real“ modules
 - breadth-first or depth-first
 - during integration of modules, tests must be re-run to guarantee that the stubbing was correct

Integration Testing How-to (2)

- Bottom-Up
 - first, drivers for all components are prepared
 - basis modules are grouped into so-called „builds“ and integrated
 - ideally the driver evaluates the testing results
 - drivers are successively replaced, functionality increases
- Incremental
 - the system grows together from both sides
 - stubs for high-level, driver for low-level modules are needed
 - stubs and driver are replaced as integration progresses



Advantages / Disadvantages (1)

- Big-bang
 - ☺ no additional work for stub and driver implementation
 - ☹ unsystematic method, testing problematic
 - ☹ error localisation may be difficult

- Top-down
 - ☺ early availability of a prototype SUT for the user
 - ☺ interleaving of design and implementation
 - ☹ stub programming is challenging
 - ☹ interaction of SUT, HW and application layer is tested rather late
 - ☹ with deeply nested hierarchies it is hard to construct test cases

Advantages / Disadvantages (2)

- Bottom-up
 - ☺ no stubs necessary
 - ☺ testing environment can be provided easily
 - ☺ testing results easy to interpret
 - ☺ testing of exception handling is easy
 - ☹ no prototypes, complete SUT only towards the end
 - ☹ design errors are recognized rather late, high error correction costs
- incremental
 - ☺ integration as soon as a component is implemented
 - ☺ easy construction of test cases, coverage can be estimated
 - ☹ requires many stubs and drivers, high development costs