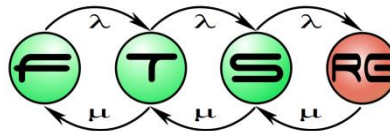


# Advanced test design techniques

**Akos Hajdu, David Honfi,  
Zoltan Micskei, Istvan Majzik**

**Budapest University of Technology and Economics  
Fault Tolerant Systems Research Group**



# Main topics of the course

- Overview (1.5)
  - Introduction, V&V techniques
- Static techniques (1.5)
  - Specification, Verifying source code
- Dynamic techniques: Testing (7)
  - Testing overview, **Test design techniques**
  - Test generation, Automation
- System-level verification (3)
  - Verifying architecture, Dependability analysis
  - Runtime verification

# Test design techniques

**Goal: Select test cases based on test objectives**

## **Specification-based**

- SUT: black box
- Only spec. is known
- Testing specified functionality

## **Structure-based**

- SUT: white box
- Inner structure known
- Testing based on internal behavior

# Coverage metrics

- What % of **testable elements** have been tested
- Testable element
  - Specification-based: requirement, functionality...
  - Structure-based: statement, decision...
- **Coverage criterion**: X % for Y coverage metric
- This is not **fault coverage**!

# How to use coverage metrics?

## Evaluation (measure)

- Evaluate quality of existing tests
- Find missing tests

## Selection (goal)

- Design tests to satisfy criteria

# SPECIFICATION-BASED TESTING

# Learning outcomes

- Describe the goal of specification-based test design techniques (K2)
- Use test design techniques decision tables and pair-wise testing to select test cases (K3)

# Specification-based techniques

Equivalence  
classes

Boundary  
values

Decision  
tables

Combinatorial  
testing

Based on  
use cases

...

# Specification-based techniques

Equivalence  
classes

Boundary  
values

Decision  
tables

Combinatorial  
testing

Based on  
use cases

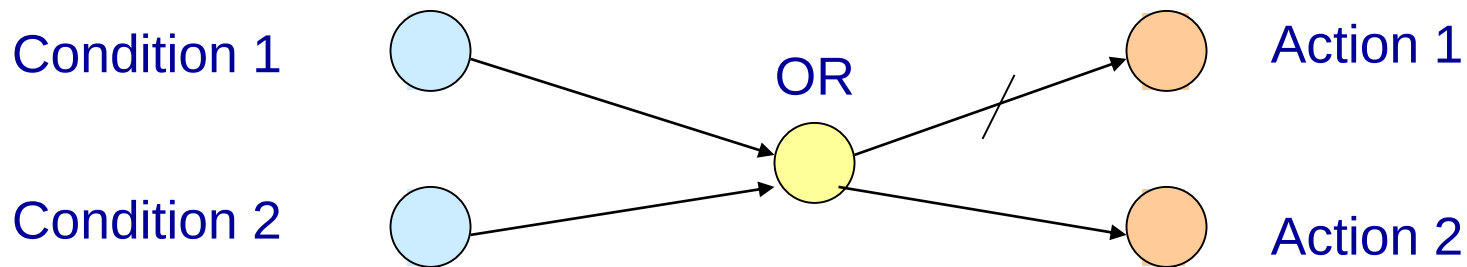
...

# Decision or cause/effect analysis

- Rules for **connecting inputs** and **outputs**
  - Business rules: price calculation, insurance, loan...
  - Technical: authentication, monitoring system...
- Connections for
  - **Condition**/cause: equiv. partitions of input parameters
  - **Action**/effect: equiv. partitions of output parameters
- **Representations:**
  - Cause-effect graphs
  - Decision tables

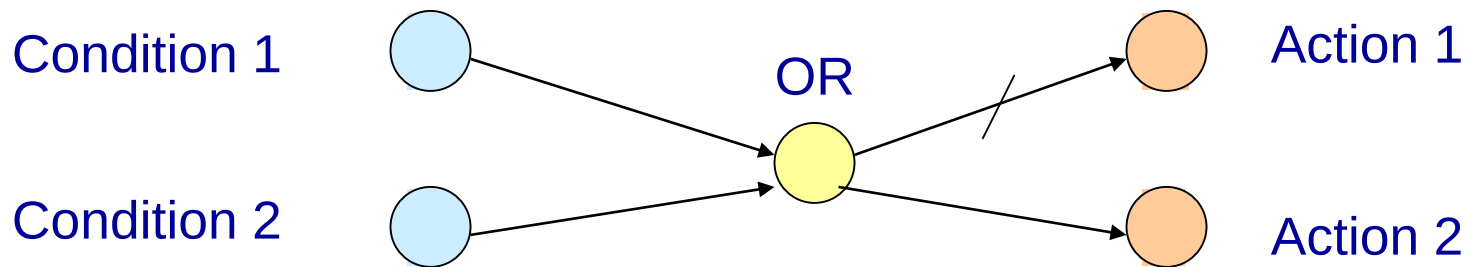
# Cause-effect analysis: representation

- Cause-effect graph (Boole graph)
  - Source: equivalence partitions of input parameters
  - Sink: equivalence partitions of output parameters
  - Intermediate: OR, AND, NOT



# Cause-effect analysis: test design

- Using for **test design**
  - Covering paths in the graph
  - Truth tables (see Digital design)
  - Originated from HW testing



# Decision tables

- Represent each conditions/actions with Booleans
- Conditions/actions in rows, business rules in columns
  - (Or representation can be transposed)
- Rules will be the test cases

|                   | Rule 1 | Rule 2 | Rule N |
|-------------------|--------|--------|--------|
| <b>Conditions</b> |        |        |        |
| Condition 1       | T      | T      |        |
| Condition 2       | F      | T      |        |
| ...               |        |        |        |
| <b>Actions</b>    |        |        |        |
| Action 1          | X      |        |        |
| Action 2          |        | X      |        |
| ....              |        |        |        |

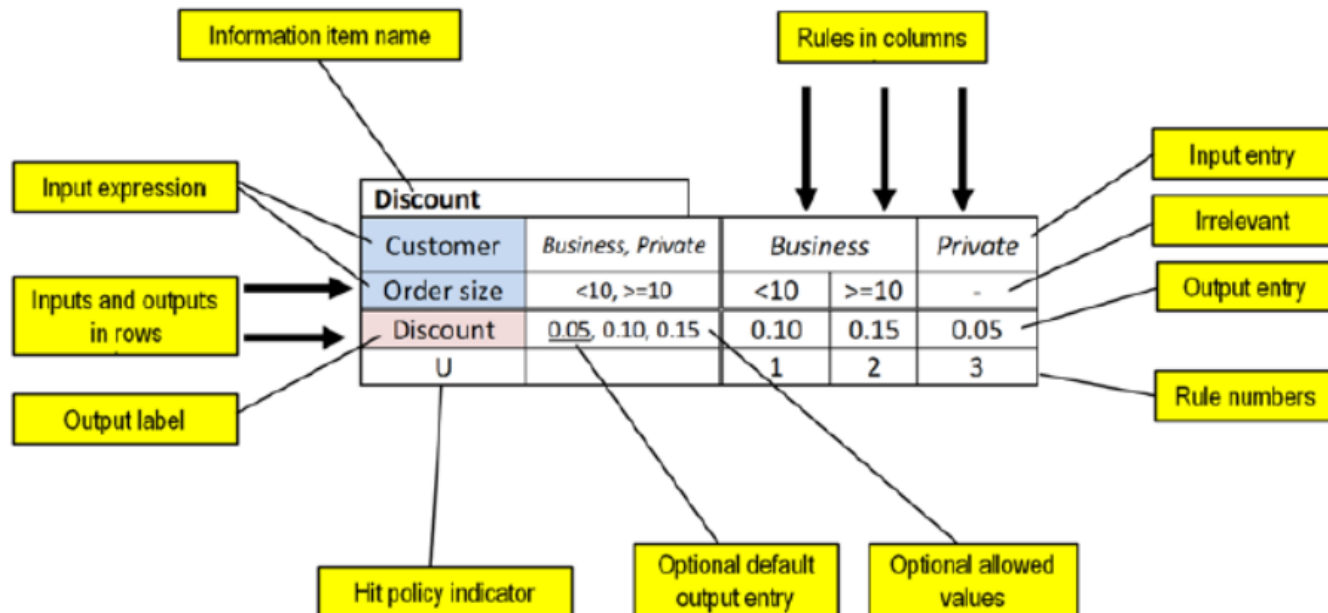
# EXERCISE Decision table

The final price of the order is calculated based on discounts. If the user has a membership card (silver 2%, gold 3%), this global discount is always applied. There are also price dependent discounts. If before applying global discounts the total amount to pay is greater than 100 EUR then the discount is 1%, if it is greater than 200 EUR then the discount is 2%.

**Create a decision table!**

# Standardized notation (decision tables)

- OMG's [Decision Model And Notation](#) (DMN)
- Represent decision' requirements, rules...



Source: OMG

# Specification-based techniques

Equivalence  
classes

Boundary  
values

Decision  
tables

Combinatorial  
testing

Based on  
use cases

...

# When there are many input parameters

- Failures are caused by (specific) combinations
- Testing all combinations: too much test cases
- Rare combinations may also cause failures

The image shows a font configuration dialog box with the following settings:

- Font:** Open Sans Extrabold
- Font style:** Italic
- Size:** 10
- Font color:** Blue
- Underline style:** Dotted
- Underline color:** Red

**Effects:**

- ☒ Strikethrough
- ☐ Double strikethrough
- ☐ Superscript
- ☐ Subscript
- ☒ Small caps
- ☐ All caps
- ☐ Hidden

**Preview:**

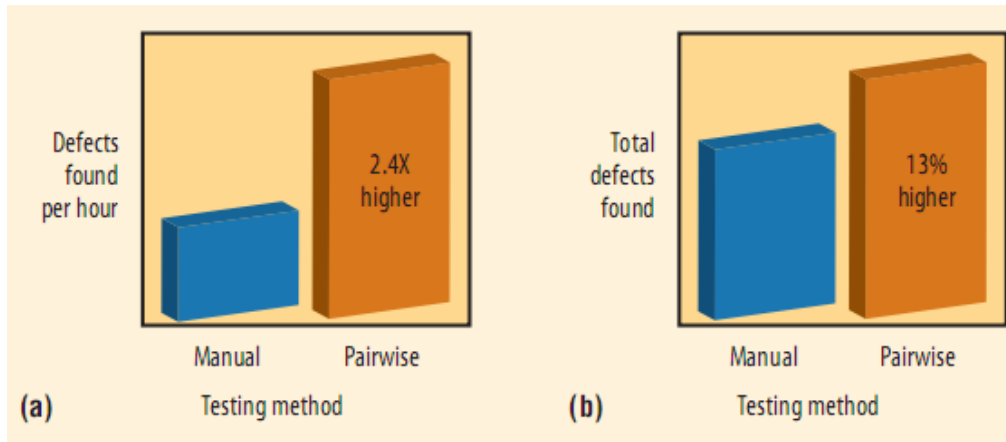
OPEN SANS EXTRABOLD

This is a TrueType font. This font will be used on both printer and screen.

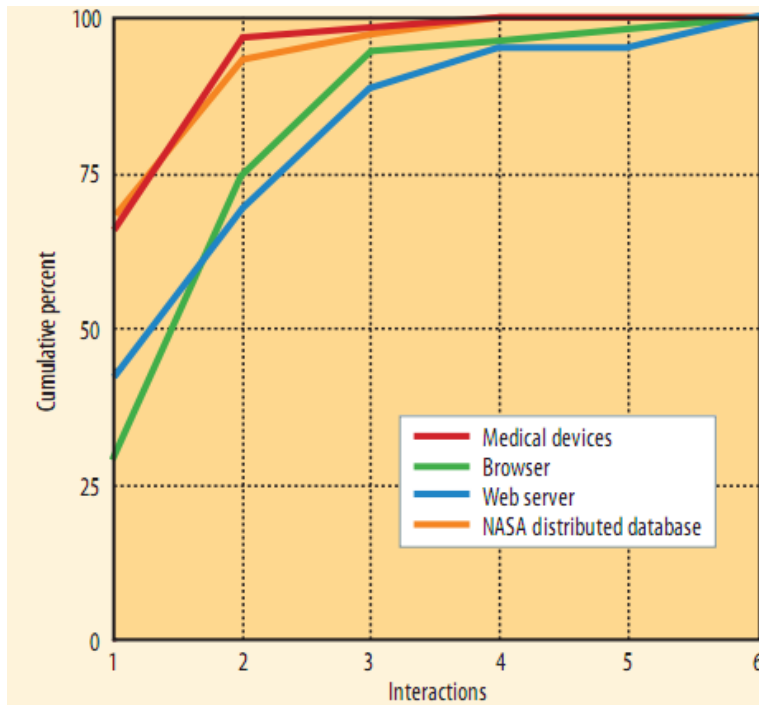
# Combinatorial testing techniques

- **Ad hoc** („best guess”)
  - Intuition, requirements, typical faults...
- **Each choice**
  - Every choice in at least one test
  - Can miss important combination
- **N-wise testing**
  - For each arbitrary **n parameters**, testing all possible combinations of their potential values
  - Special case ( $n = 2$ ): pairwise testing

# Efficiency of n-wise testing



Comparing ad hoc and pairwise testing (10 projects)



Many faults are triggered by specific combinations of at least 2 parameters (or even 3-6)

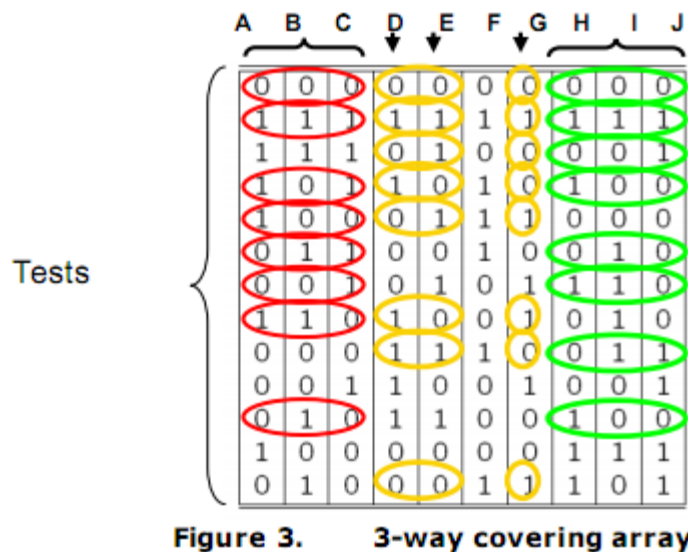
Source: R. Kuhn et al. „Combinatorial Software Testing”, IEEE Computer, 42:8, 2009

# EXERCISE Pair-wise testing

- Given input parameters and potential values:
  - OS: Windows, Linux
  - CPU: Intel, AMD
  - Protocol: IPv4, IPv6
- How many combinations are possible?
- How many test cases are needed for pairwise testing?
- A potential test suite:
  - T1: Windows, Intel, IPv4
  - T2: Windows, AMD, IPv6
  - T3: Linux, Intel, IPv6
  - T4: Linux, AMD, IPv4

# N-wise testing: theory and practice

- Theory: constructing a coverage array



Source: D. R. Kuhn, R. N. Kacker, Y. Lei  
[Practical Combinatorial Testing](#)  
NIST Special Publication 800-142

- Tools (see <http://www.pairwise.org>)
  - [PICT](#): Pairwise Independent Combinatorial Testing (MS)
  - [ACTS](#) - Advanced Combinatorial Testing Suite (NIST)

# STRUCTURE-BASED TESTING

# Structure-based Testing: Outline

- **Recap: basic concepts**
- Control-flow criteria
- Data-flow criteria
- Evaluation of structure-based testing

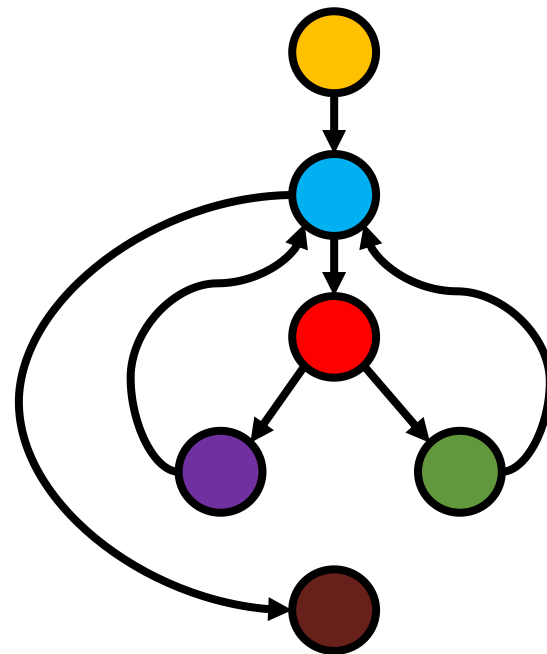
# What is “internal structure”?

- In case of models: structure of the model
- In case of code: structure of the code (CFG)

Source code:

```
int a = 1;
while(a < 16) {
    if(a < 10) {
        a += 2;
    } else {
        a++;
    }
}
a = a * 2;
```

Control-flow graph:



# Basic concepts

```
int t = 1;
```

```
Speed s = SLOW;
```

Statement

Block

```
if (! started){  
    start();  
}
```

```
if (t > 10 && s == FAST){  
    brake();  
} else {  
    accelerate();  
}
```

Condition

Decision

Branch

# Basic concepts

- **Statement**
- **Block**
  - A sequence of one or more consecutive executable statements containing no branches
- **Condition**
  - Logical expression without logical operators (and, or...)
- **Decision**
  - A logical expression consisting of one or more conditions combined by logical operators
- **Branch**
  - Possible outcome of a decision
- **Path**
  - A sequence of events, e.g., executable statements, of a component typically from an entry point to an exit point.

# Example: decision and condition

- A decision with one condition:

```
if (temp > 20) {...}
```

- A decision with 3 conditions:

```
if (temp > 20 && (valveIsOpen || p == HIGH)) {...}
```

# Control Flow Graph (CFG)

- A CFG represents the flow of control
- **$G = (N, E)$  directed graph**
  - Node  $n \in N$  is a basic block
    - Basic block: Sequence of statements with exactly one entry and exit points.
  - Edge  $e = (n_i, n_j) \in E$  is a possible flow of control from basic block  $n_i$  to basic block  $n_j$

# EXERCISE Building a CFG

```
public void insertionSort(int[] a) {  
    for(int i = 0; i < a.size(); i++) {  
        int x = a[i];  
        int j = i - 1;  
        while(j >= 0 && a[j] > x) {  
            a[j+1] = a[j];  
            j = j - 1;  
        }  
        a[j+1] = x;  
    }  
    System.out.println("Finished.");  
}
```

Build the CFG of  
this program  
code

# Structure-based Testing: Outline

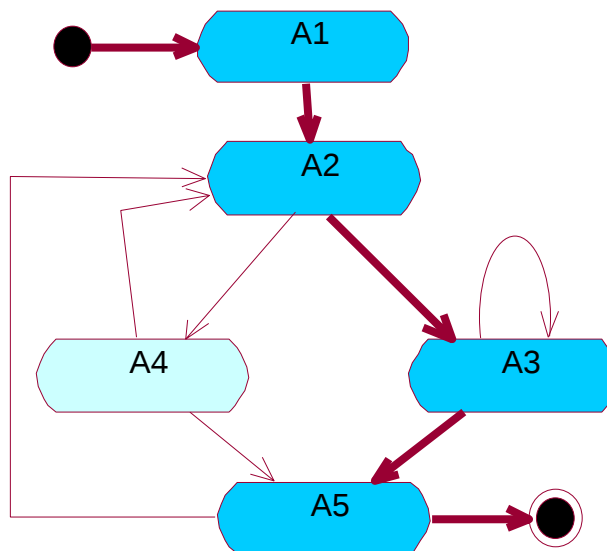
- Recap: basic concepts
- **Control-flow criteria**
- Data-flow criteria
- Evaluation of structure-based testing

# Learning outcomes

- Explain the differences between different control-flow based coverage criteria (K2)
- Design tests using control-flow based coverage criteria for imperative programs (K3)

# 1. Statement coverage

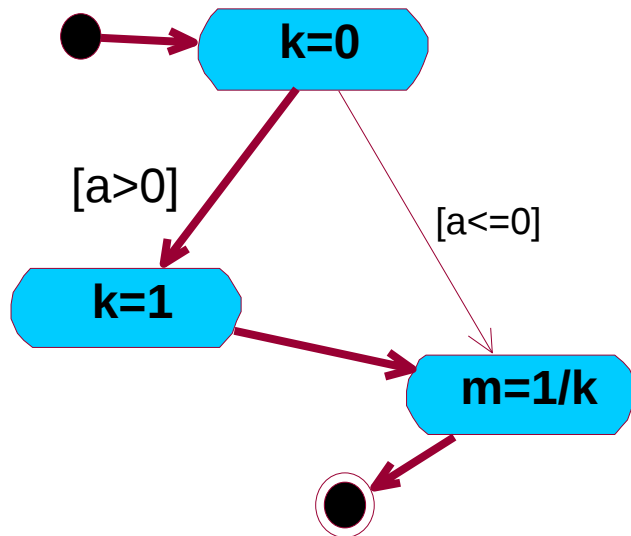
$$\frac{\text{Number of statements executed during testing}}{\text{Number of all statements}}$$



Statement coverage:  $4/5 = 80\%$

# Assessing statement coverage

All statement is executed at least once



Statement coverage: 100%

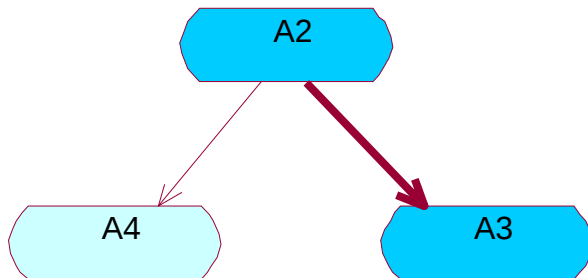
BUT: **[a<=0]** branch missing!

Does not guarantee coverage of empty branches

## 2. Decision coverage

Outcomes of decisions taken during testing

Number of all possible outcomes



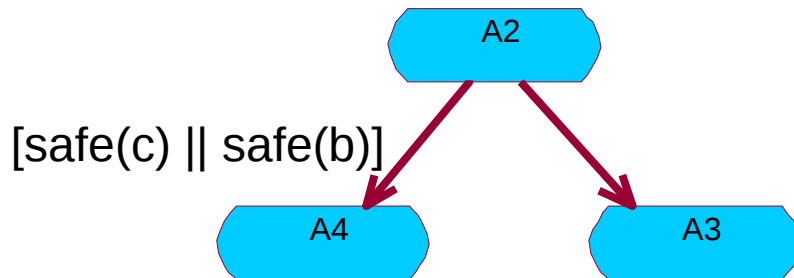
Decision coverage:  $1/2 = 50\%$

How many outcomes can a decision have?

# Assessing decision coverage

All statement is executed at least once

All outcomes of decisions are covered



100% decision coverage:

| # | safe(c) | safe(b) |
|---|---------|---------|
| 1 | T       | F       |
| 2 | F       | F       |

safe(b) == True missing!

Does not take into account all combinations of conditions!

# 3. Condition coverage

Generic coverage metric for conditions:

$$\frac{\text{Number of tested combinations of conditions}}{\text{Number of aimed combinations of conditions}}$$

Definition (what conditions are aimed):

- Every condition must be set to true and false during testing

Other possible definition:

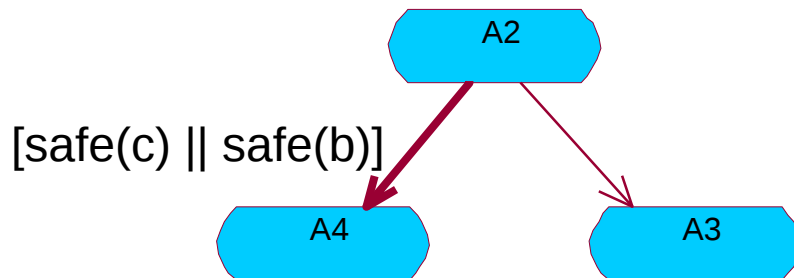
- Every condition is **evaluated** to both true and false
  - Not the same as above due to *lazy evaluation*

# Assessing condition coverage

Every condition has taken all possible outcomes at least once

100% condition coverage:

| # | safe(c) | safe(b) |
|---|---------|---------|
| 1 | T       | F       |
| 2 | F       | T       |



False outcome of decision missing!

Does not yield 100% decision coverage!

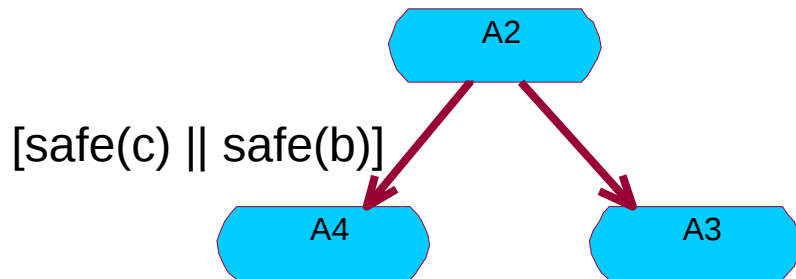
# 4. Condition/Decision Coverage (C/DC)

Combination of condition and decision coverage

# Assessing C/DC Coverage

Every decision has taken all possible outcomes at least once.

Every condition has taken all possible outcomes at least once



100% C/DC coverage:

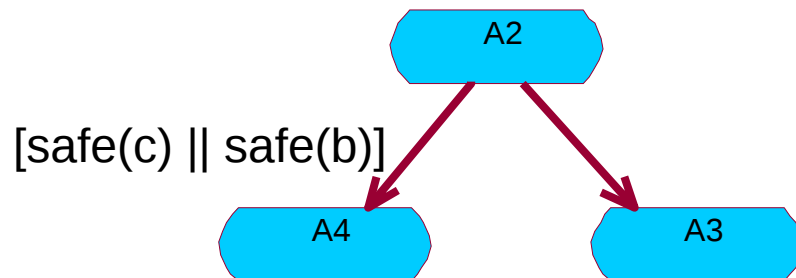
| # | safe(c) | safe(b) |
|---|---------|---------|
| 1 | T       | T       |
| 2 | F       | F       |

Does not take into account whether the condition has any effect!

## 5. Modified Condition/Decision Coverage (MC/DC)

- Each entry and exit point has been invoked at least once,
- every condition in a decision in the program has taken all possible outcomes at least once,
- every decision in the program has taken all possible outcomes at least once,
- each condition in a decision is shown to independently affect the outcome of the decision.

100% MC/DC coverage:



| # | safe(c) | safe(b) |
|---|---------|---------|
| 1 | T       | F       |
| 2 | F       | T       |
| 3 | F       | F       |

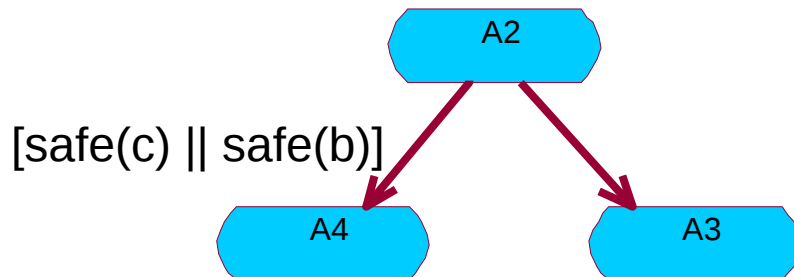
# 6. Multiple Condition Coverage

Every combinations of conditions tried

- For  $n$  conditions  $2^n$  test cases may be necessary!
- (Bit less with lazy evaluation)
- Sometimes not practical, e.g. in avionics systems there are programs with more than 30 conditions!

100% MCC coverage:

| # | safe(c) | safe(b) |
|---|---------|---------|
| 1 | F       | F       |
| 2 | F       | T       |
| 3 | T       | F       |
| 4 | T       | T       |



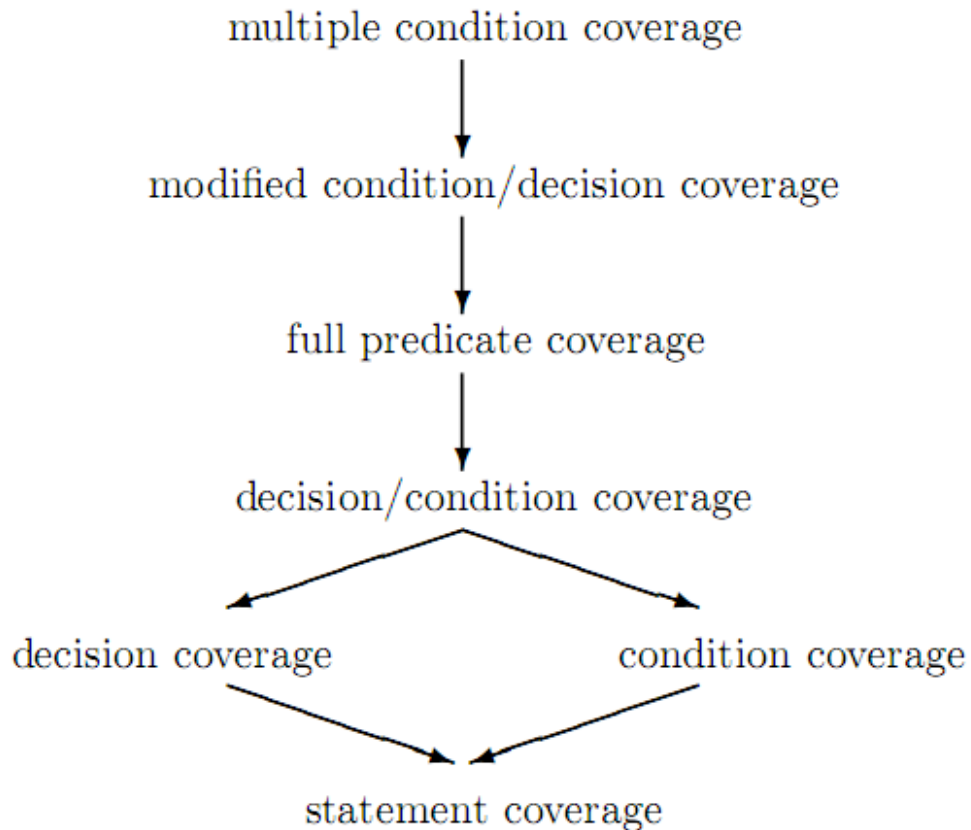
# Comparing control-flow criteria

Table 1. Types of Structural Coverage

| Coverage Criteria                                                                            | Statement Coverage | Decision Coverage | Condition Coverage | Condition/Decision Coverage | MC/DC | Multiple Condition Coverage |
|----------------------------------------------------------------------------------------------|--------------------|-------------------|--------------------|-----------------------------|-------|-----------------------------|
| Every point of entry and exit in the program has been invoked at least once                  |                    | •                 | •                  | •                           | •     | •                           |
| Every statement in the program has been invoked at least once                                | •                  |                   |                    |                             |       |                             |
| Every decision in the program has taken all possible outcomes at least once                  |                    | •                 |                    | •                           | •     | •                           |
| Every condition in a decision in the program has taken all possible outcomes at least once   |                    |                   | •                  | •                           | •     | •                           |
| Every condition in a decision has been shown to independently affect that decision's outcome |                    |                   |                    |                             | •     | • <sup>8</sup>              |
| Every combination of condition outcomes within a decision has been invoked at least once     |                    |                   |                    |                             |       | •                           |

Source: Kelly J. Hayhurst et al. „A Practical Tutorial on Modified Condition/Decision Coverage”, NASA/TM-2001-210876, 2001

# Comparing control-flow criteria



Source: S. A. Vilkomir and J. P. Bowen, "From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria," *Formal Aspects of Computing*, vol. 18, no. 1, pp. 42-62, 2006.

# EXERCISE Structure-based test design

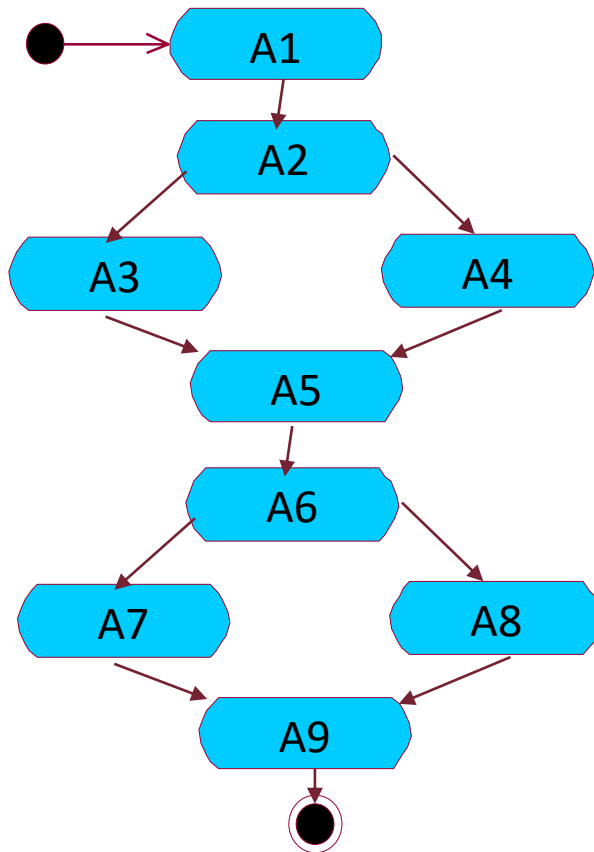
```
Product getProduct(String name, Category cat){  
    if (name == null || ! cat.isValid)  
        throw new IllegalArgumentException();  
  
    Product p = ProductCache.getItem(name);  
  
    if (p == null){  
        p = DAL.getProduct(name, cat);  
    }  
  
    return p;  
}
```

Design tests for

1. Statement
2. Decision
3. C/DC coverage

# 7. Basis path coverage

Number of independent paths traversed during testing  
Number of all independent paths



## Tests

1. A1, A2, A3, A5, A6, A7, A9
2. A1, A2, A4, A5, A6, A8, A9

Statement coverage: ?

Decision coverage: ?

Path coverage: ?

# Assessing full path coverage

- 100% path coverage implies:
  - 100% statement coverage, 100% decision coverage
  - 100% multiple condition coverage is not implied
- Full path coverage is usually not practical in case of loops

# Additional coverage criteria

- Loop
  - Executing loops 0, 1 or more times
- Race
  - Executions from multiple threads on code
- ...

# Calculating coverage in practice

- Every tool uses **different definitions**
- Implementation
  - **Instrument** source/byte code
  - Adding instructions to count coverage

```
if (a > 10){  
    CoveredBranch(1, true);  
    b = 3;  
} else {  
    CoveredBranch(1, false);  
    b = 5;  
}  
send(b);
```

See also: [Is bytecode instrumentation as good as source code instrumentation](#), 2013.

# Structure-based Testing: Outline

- Recap: basic concepts
- Control-flow criteria
- **Data-flow criteria** (*optional*)
- Evaluation of structure-based testing

# Learning outcomes

- Summarize the basic ideas of data-flow coverage criteria (K2)

# Goal of data-flow coverage

## ■ Idea:

- Track the assignment and usage of variables
- Label CFG with data-flow events

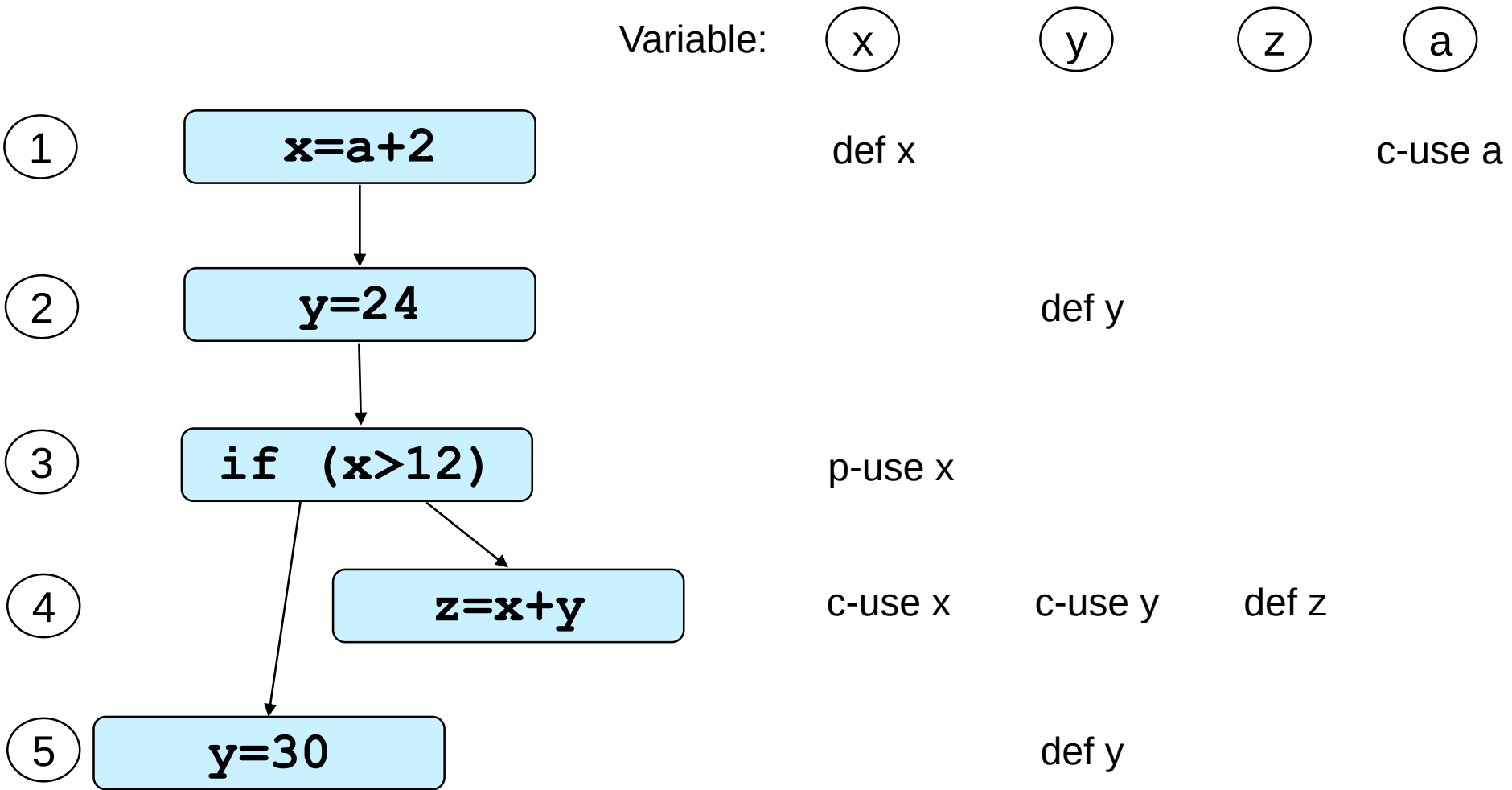
## ■ Faults to detect:

- Erroneous assignments
- Effect of assignments

# Labeling the control flow graph

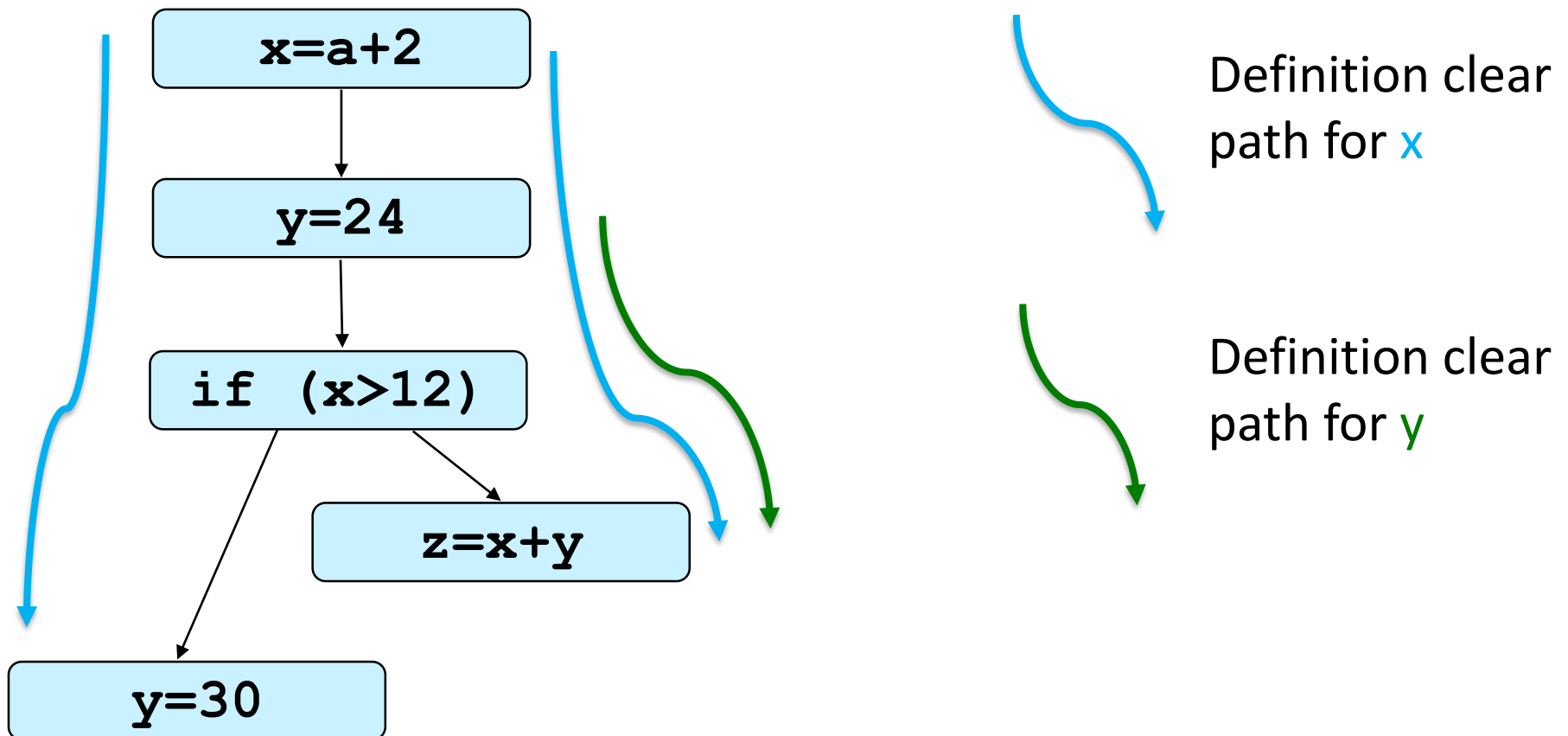
- **def(v)**: variable **v** is assigned in the given location
- **use(v)**: variable **v** is used in the given location
  - **p-use(v)**: value of variable **v** is used in a condition
  - **c-use(v)**: value of variable **v** is used in a computation

# EXERCISE Labeling variable def and use



# Program paths

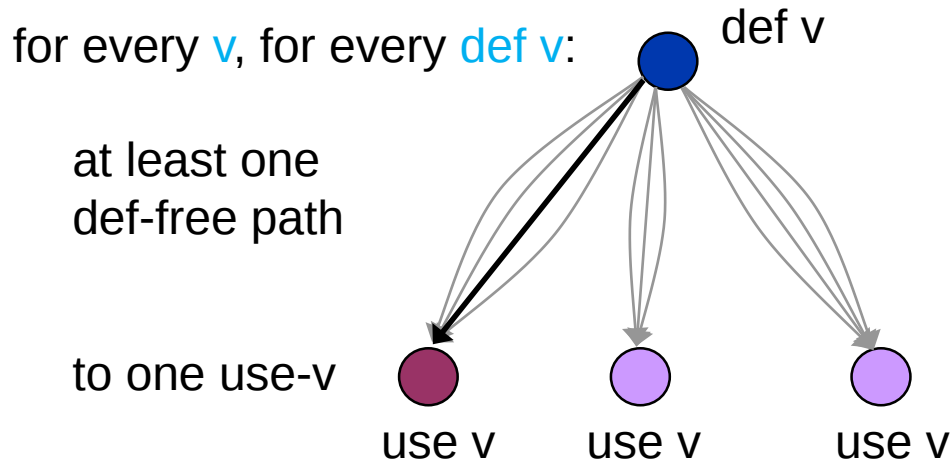
- **Definition clear path** for variable **v**
  - **v** is not assigned in the nodes of the path



# Data-flow criteria

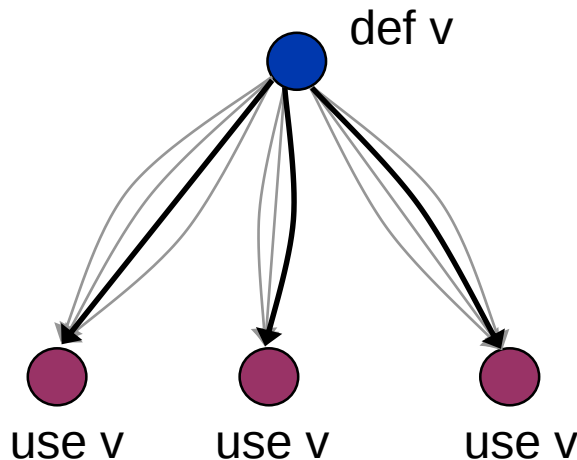
## ■ All-defs:

- def  $v$
- use  $v$

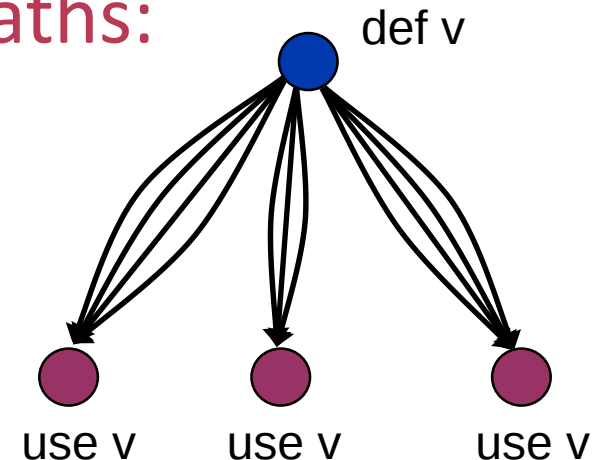


## ■ All-uses:

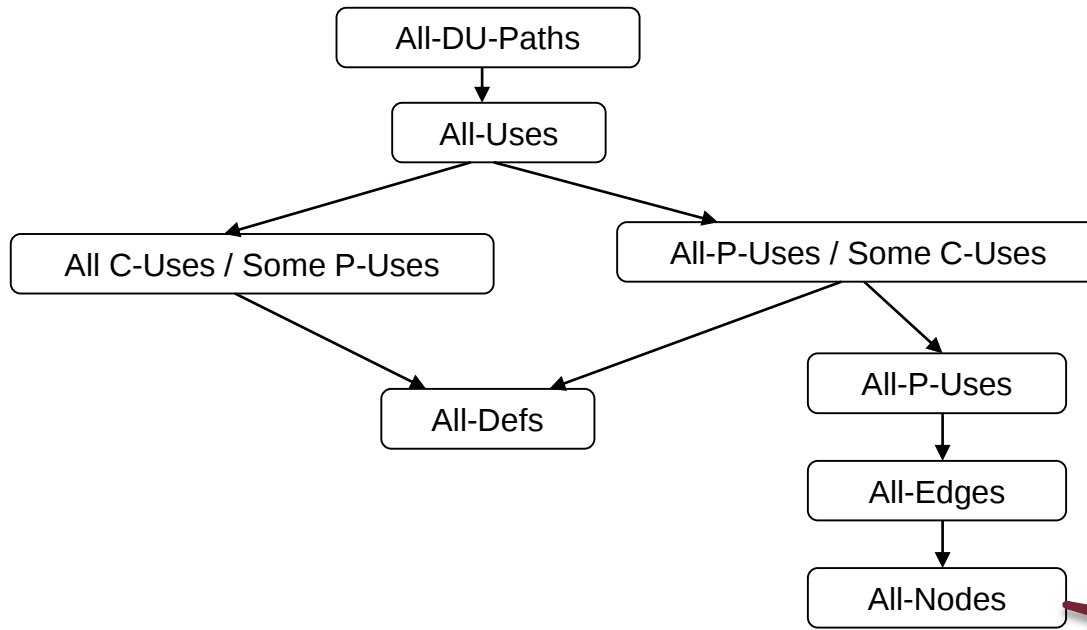
- p-uses,
- c-uses



## ■ All-paths:



# Comparing structural coverage criteria



Standards for safety-critical prescribe more complex criteria

Average projects do not measure coverage or aim only for statement coverage

# Structure-based Testing: Outline

- Recap: basic concepts
- Control-flow criteria
- Data-flow criteria
- **Evaluation of structure-based testing**

# Using structural test coverage criteria

## ■ Can be used for:

- Finding not tested parts of the program
- Measuring “completeness” of test suite
- Basis for exit criteria
- [Spoiler] Test generation (see lectures later)

## ■ Cannot be used for:

- Finding/testing missing or not implemented requirements
- Only indirectly connected to code quality

# Using structural test coverage criteria

## ■ Experience from Microsoft

- „Test suite with **high code coverage** and **high assertion density** is a good indicator for code quality.”
- „**Code coverage alone** is generally **not enough** to ensure a good quality of unit tests and should be used with care.”
- „The **lack of code coverage** to the contrary clearly indicates a **risk**, as many behaviors are untested.”

(Source: „Parameterized Unit Testing with Microsoft Pex”)

## ■ Related case studies:

- „*Coverage Is Not Strongly Correlated with Test Suite Effectiveness*”, 2014. DOI: [10.1145/2568225.2568271](https://doi.org/10.1145/2568225.2568271)
- „*The Risks of Coverage-Directed Test Case Generation*”, 2015. DOI: [10.1109/TSE.2015.2421011](https://doi.org/10.1109/TSE.2015.2421011)

# SUMMARY

# Test design techniques

- Specification and structure based techniques
  - Many orthogonal techniques
  - Every techniques need practice!
- Only basic techniques are used commonly ☹
  - Exception: safety-critical systems  
(e.g. DO178-B requires MC/DC coverage analysis)
- **Combination of techniques** is useful:
  - Example (Microsoft report):  
specification based: 83% code coverage  
+ exploratory: 86% code coverage  
+ structural: 91% code coverage