Structure-based test design

David Honfi, Zoltan Micskei, Istvan Majzik

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

Main topics of the course

- Overview (1)
  - V&V techniques, Critical systems
- Static techniques (2)
  - Verifying specifications
  - Verifying source code
- **Dynamic techniques: Testing (7)**
  - Developer testing, **Test design techniques**
  - Testing process and levels, 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

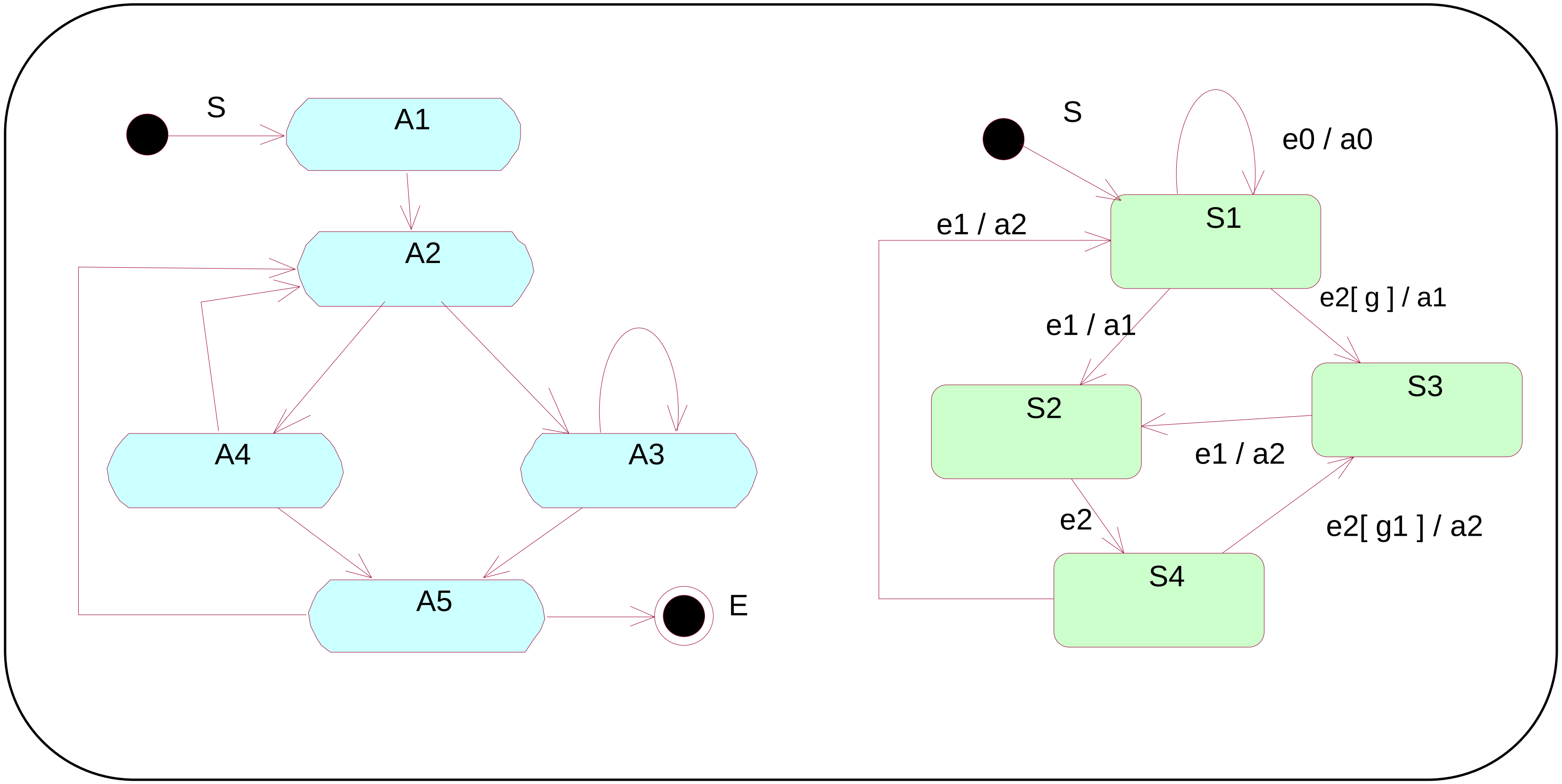
Structure-based

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

STRUCTURE-BASED TESTING

What is “internal structure”?

- In case of models: structure of the model



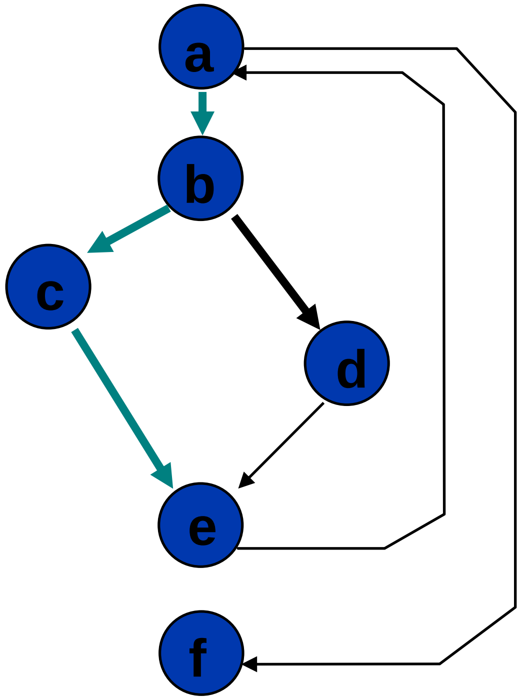
What is “internal structure”?

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

Source code:

```
a:  for (i=0; i<MAX; i++) {
b:          if (i==a) {
c:        n=n-i;
            } else {
d:        m=n-i;
            }
e:          printf("%d\n",n);
      }
f:  printf("Ready.")
```

Control-flow graph:



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)	Selection (goal)
• Evaluate quality of existing tests • Find missing tests	• Design tests to satisfy criteria

CONTROL-FLOW CRITERIA

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)

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
- 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)

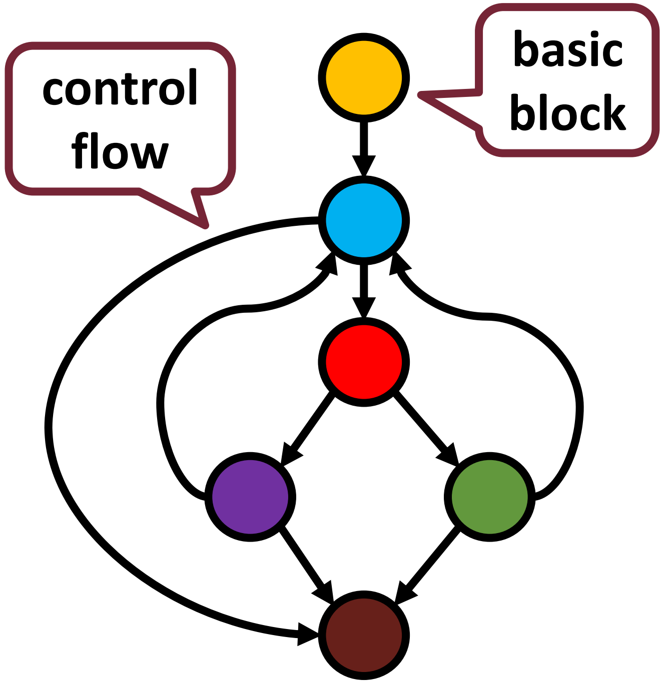
- A CFG represents the flow of control
- ***G = (N, E)* directed graph**
  - Node $n \in N$ is a basic block
    - Basic block: Sequence of statements with exactly one entry and exit points.
  - Edge $e = (n_i, n_j) \in E$ is a possible flow of control from basic block n_i to basic block n_j

Example: Control Flow Graph

Source code:

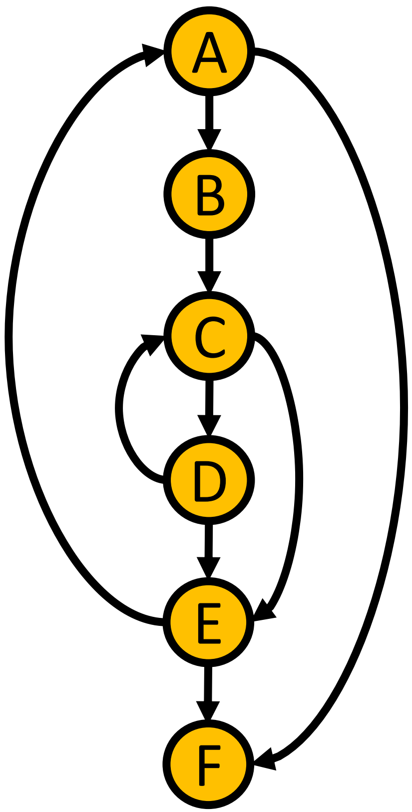
```
int a = 1;
while(a < 16) {
  if(a < 10) {
   a += 2;
  } else {
   a++;
  }
}
a = a * 2;
```

Control-flow graph:



EXERCISE Building a CFG

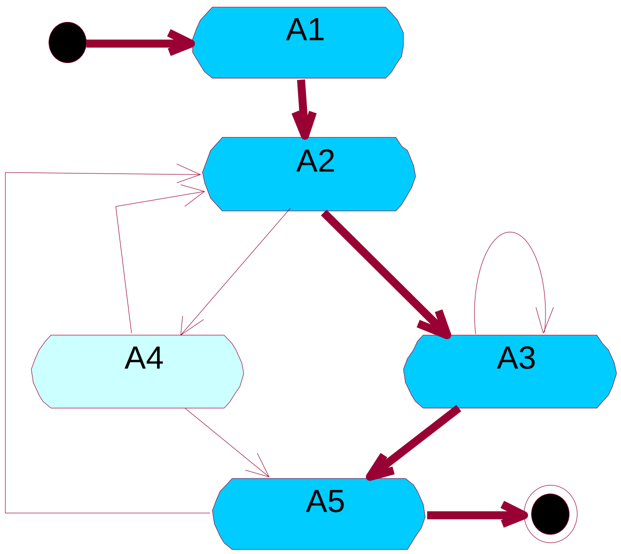
```
public void insertionSort(int[] a) {
   for(int i = 0; i < a.size(); i++) {
      int x = a[i];
      int j = i - 1;
      while(j >= 0 && a[j] > x) {
         a[j+1] = a[j];
         j = j - 1;
      }
      a[j+1] = x;
   }
   System.out.println("Finished.");
}
```



Build the CFG of this program code

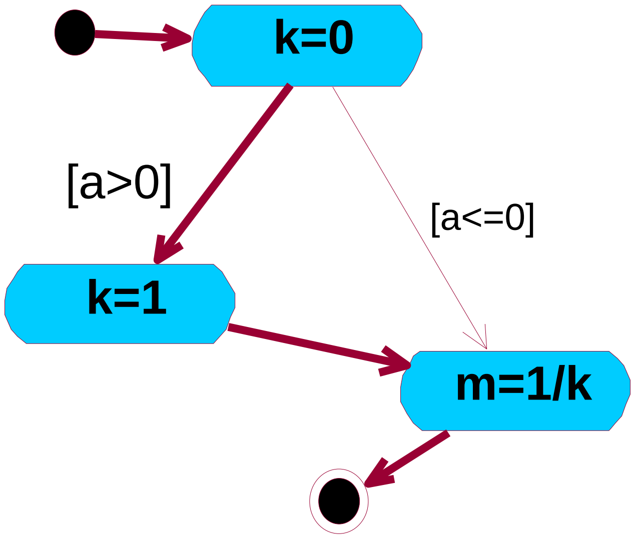
1. Statement coverage

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



Statement coverage: 80%

Assessing statement coverage



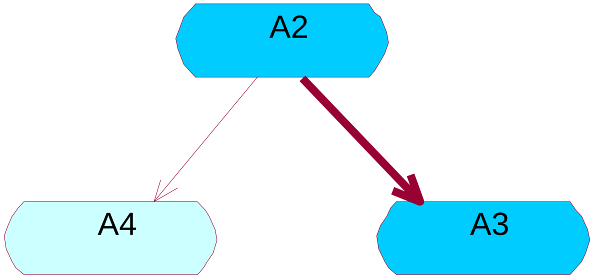
Statement coverage: 100%

Missing: [a<=0] branch

Does not require to cover empty branches!

2. Decision coverage

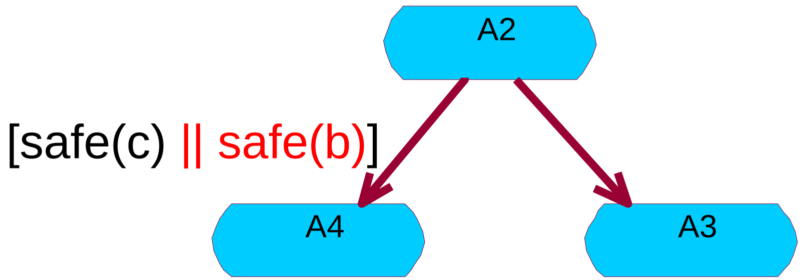
$$\frac{\text{Outcomes of decisions taken during testing}}{\text{Number of all possible outcomes}}$$



Decision coverage: 50%

Assessing decision coverage

Does not take into account all combinations of conditions!



100% decision coverage:

1. safe(c) = true, safe(b) = false
2. safe(c) = false, safe(b) = false

3. Condition coverage

Generic coverage metric for conditions:

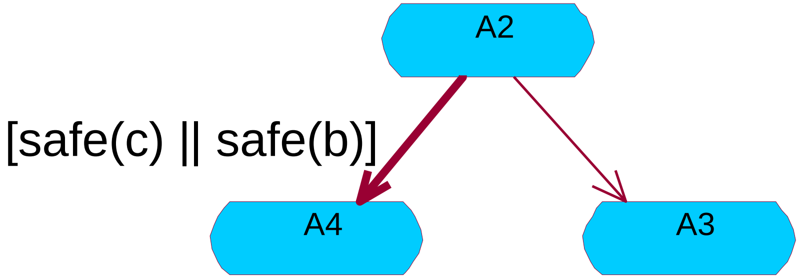
$$\frac{\text{Number of tested combinations of conditions}}{\text{Number of aimed combinations of conditions}}$$

Definition (what conditions are aimed):

- Every condition must be set to true and false during testing
  - Does not yield 100% decision coverage!

Example of 100% condition coverage:

1. safe(c) = true, safe(b) = false
2. safe(c) = false, safe(b) = true



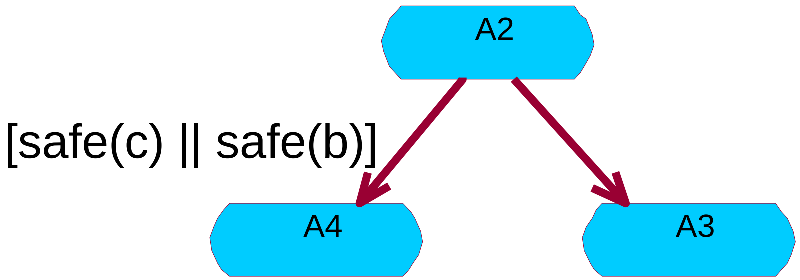
Other possible definition:

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

4. Condition/Decision Coverage (C/DC)

- Every condition in a decision in the program has taken all possible outcomes at least once, and
- every decision in the program has taken all possible outcomes at least once.

100%-os C/DC coverage:
1. safe(c) = true, safe(b) = true
2. safe(c) = false, safe(b) = false

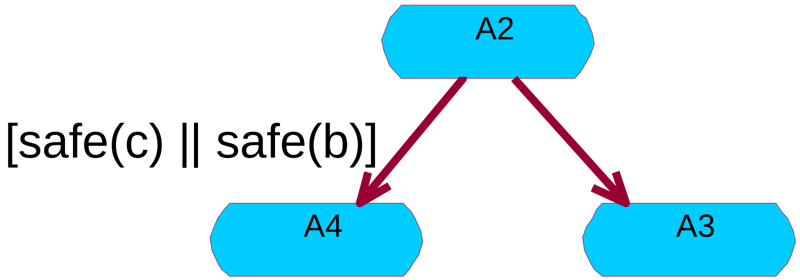


Does not take into account whether the condition has any effect!

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.

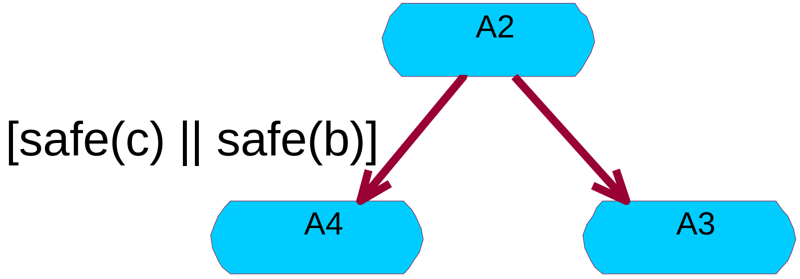
100%-os MC/DC coverage:
1. safe(c) = true, safe(b) = false
2. safe(c) = false, safe(b) = true
3. safe(c) = false, safe(b) = false



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% multiple condition coverage:
1. safe(c) = true, safe(b) = false
2. safe(c) = false, safe(b) = true
3. safe(c) = false, safe(b) = false
4. safe(c) = true, safe(b) = true



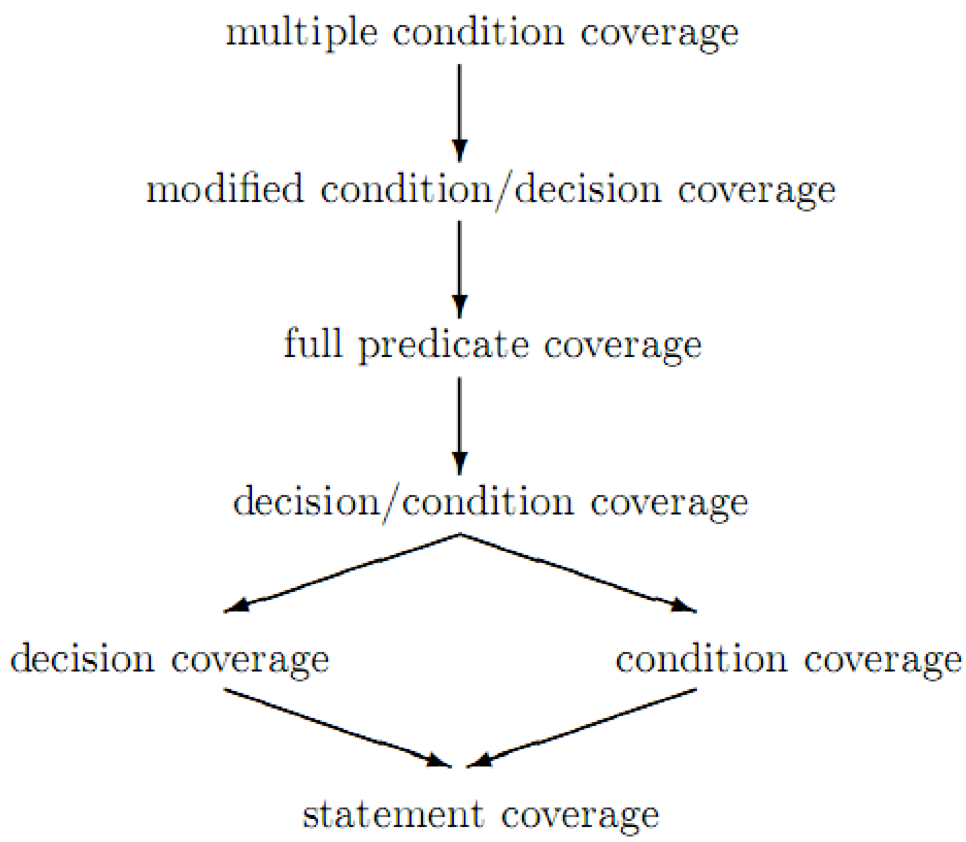
Comparing control-flow criteria

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					•	•[8]
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 Specification-based test design

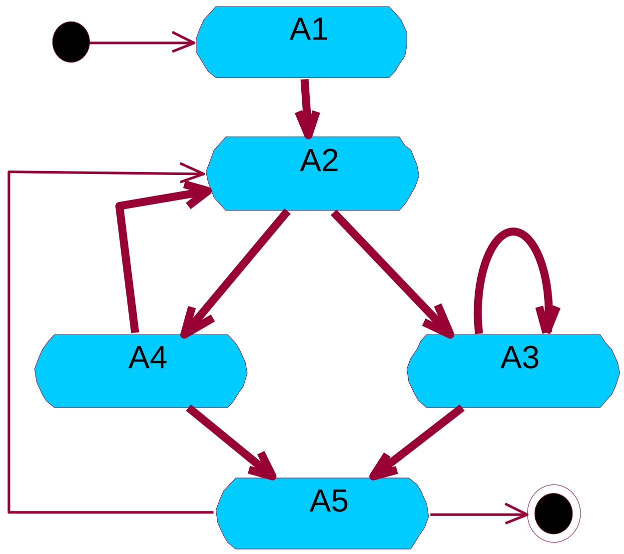
```
Product getProduct(String name, Category cat){
  if (name == null || ! cat.isValid)
    throw new IllegalArgumentException();

  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

$$\frac{\text{Number of independent paths traversed during testing}}{\text{Number of all independent paths}}$$



Path coverage: 80%

Statement coverage: 100%

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

A structure based testing technique

- Goal: Covering independent paths
  - Independent paths from the point of view of testing:
    There is a statement or decision in the path,
    that is not included in the other path
- The maximal number of independent paths:
  - CK, cyclomatic complexity
  - In regular control flow graphs:
    $CK(G)=E-N+2$, where

    E: number of edges

    N: number of nodes in the control flow graph G
    (connected graph, with 1-1 initial and final node)
- The set of independent paths is not unique

Generating structure based test sequences

- Algorithm:
  - Selecting max. CK independent paths
  - Generating inputs to traverse the paths, each after the other
- Problems:
  - Not all paths can be traversed (see conditions)
    - Is it possible to generate a proper input sequence?
    - It is possible to set the internal variables in a proper way to traverse the selected path?
  - Cycles: Traversal shall be limited (minimized)
- There are no fully automated tools to generate test sequences for path coverage

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.

DATA-FLOW COVERAGE

Learning outcomes

- Summarize the basics of data-flow coverage criteria (K2)

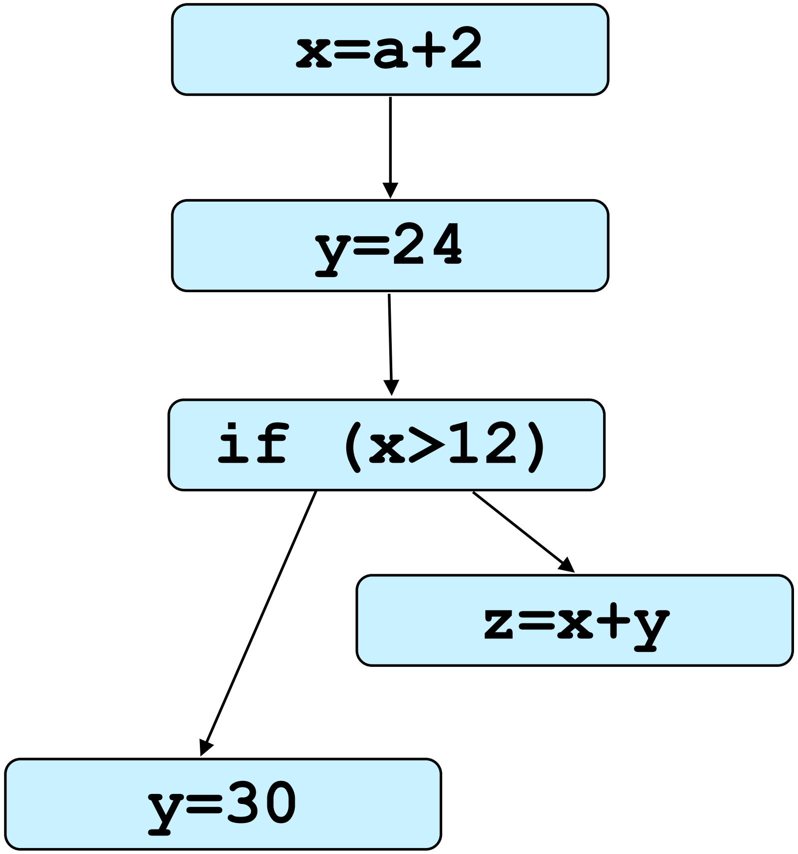
Goal of data-flow coverage

- 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

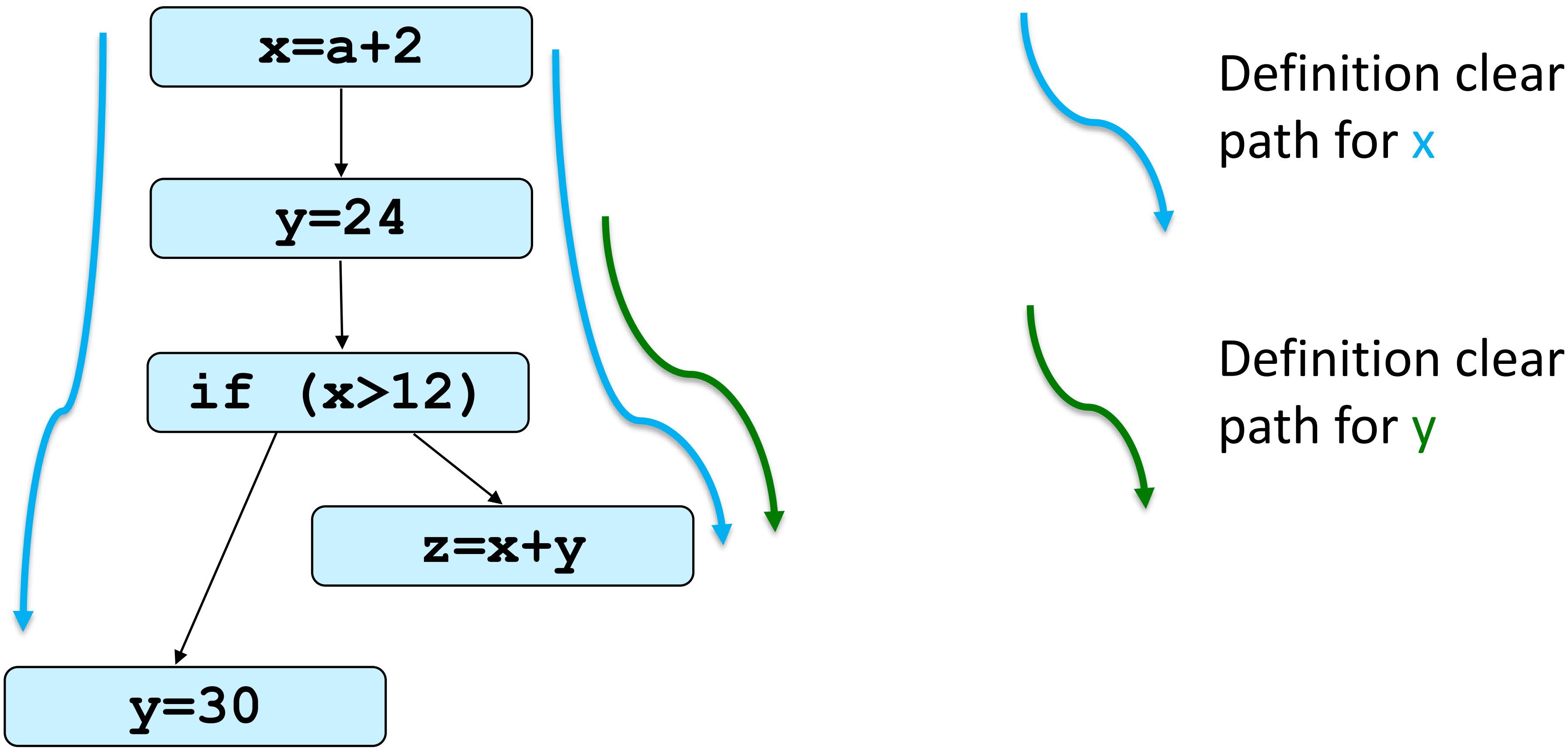
EXERCISE Labeling variable def and use



Variable:	x	y	z	a
	def x			c-use a
		def y		
	p-use x			
	c-use x	c-use y	def z	
		def y		

Program paths

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



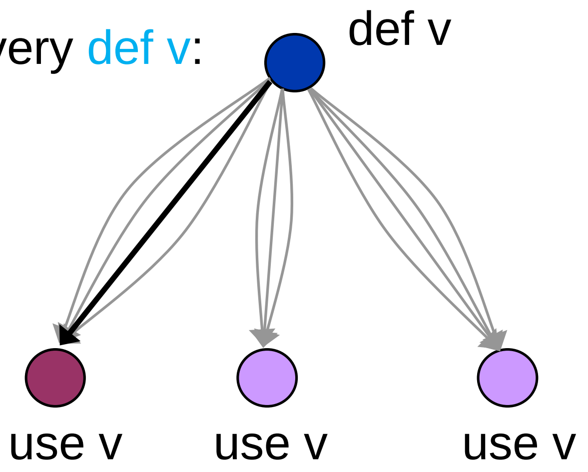
Data-flow criteria

- All-defs:
  - def v
  - use v

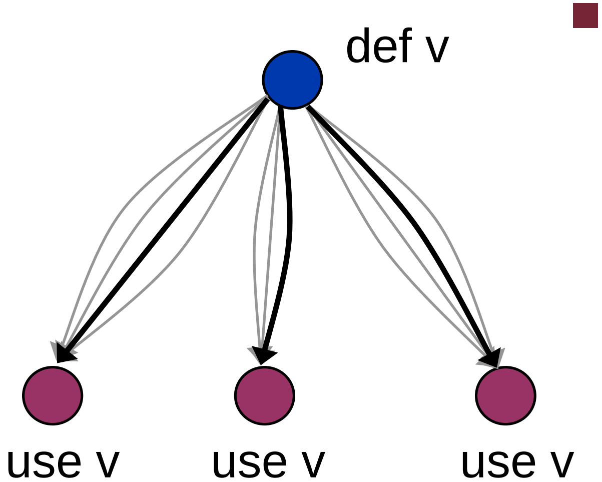
for every v, for every def v:

at least one
def-free path

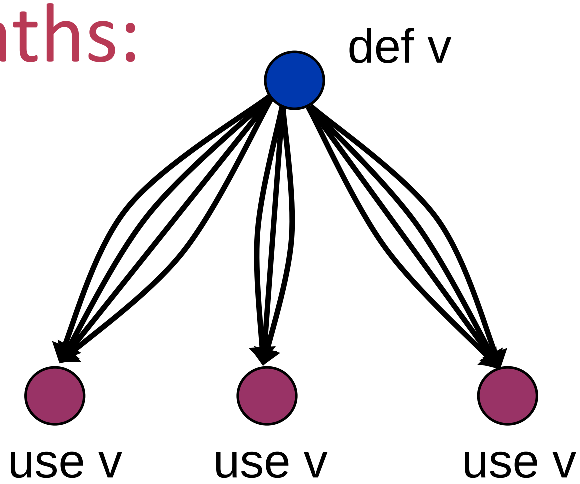
to one use-v



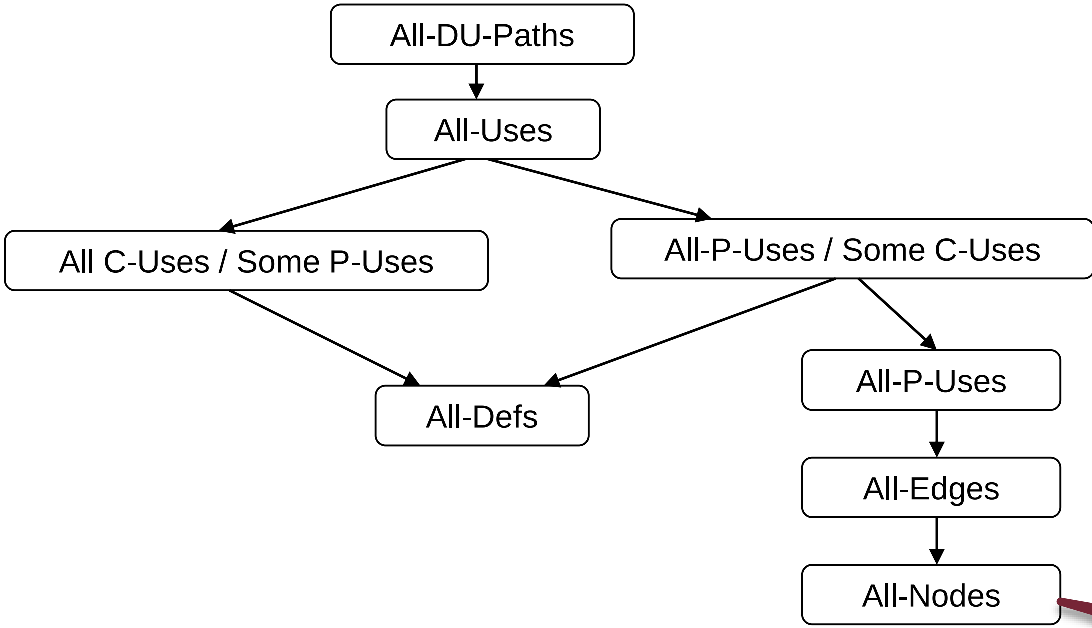
- All-uses:
  - p-uses,
  - c-uses



- 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

SUMMARY

Using test coverage criteria

- Can be used for:
  - Find not tested parts of the program
  - Measure “completeness” of test suite
  - Can be basis for exit criteria

- Cannot be used for:
  - Finding/testing missing or not implemented requirements
  - Only indirectly connected to code quality

Using 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
  - „*The Risks of Coverage-Directed Test Case Generation*”, 2015. DOI: 10.1109/TSE.2015.2421011