Automated Testing

MŰEGYETEM 1782

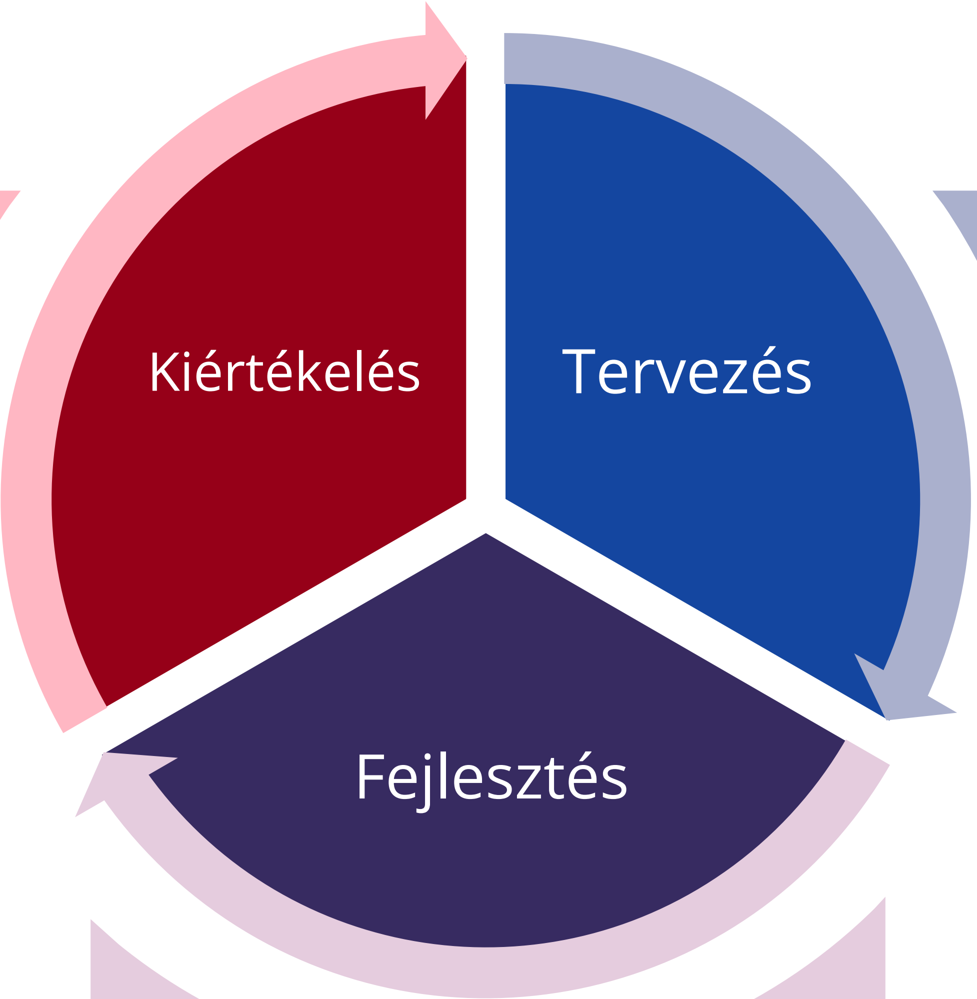
BME FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

ftsrg Critical Systems Research Group

- Idő:
  - Most péntek: nincs tanítás.
  - Jövő Hétfő: hf.
  - Jövő Kedd: FONTOS!
    - Meghívott előadó
    - Vizsgakonzultáció, mintavizsga
    - 10 perc önlab tárgy adminisztratív eligazítás
- TDK5

Overview

- Performance evaluation
- Data Analysis
- Code Quality
- Static Analysis
- Testing & Coverage



- Graph-Based Modeling
- Textual Modeling
- Code Generation
- Model Intelligence
- Model Checking

- Build Automation
- Code Generation
- Code Intelligence

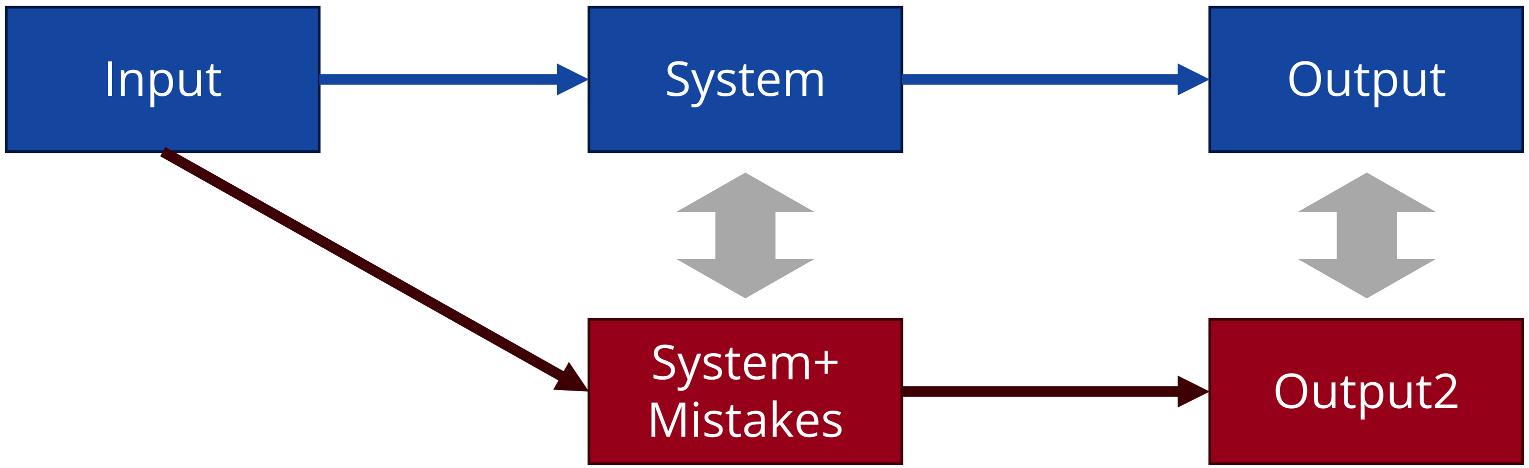
Basic concepts

Testing setup

- Normal setup

Testing setup

- **Fault hypothesis:** instead of a correct system, we have an system with faults that can lead to failures



Mutation operators

- Potential faults are characterized by a set of mutations.

$$O = \{op_1, op_2, \dots, op_n\}$$

e.g.:

  - Use + instead of –, Use = instead of ==

or more exotic:

  - Accidentally delete an edge from system model
  - Create biased training set
  - Miss a requirement

- Mutation operations can create mutants form a system S.

$$M = \{op_1(S), op_2(S), \dots, op_n(S)\}$$

This M can be **huge**, so sometimes sampled, or used in statistics

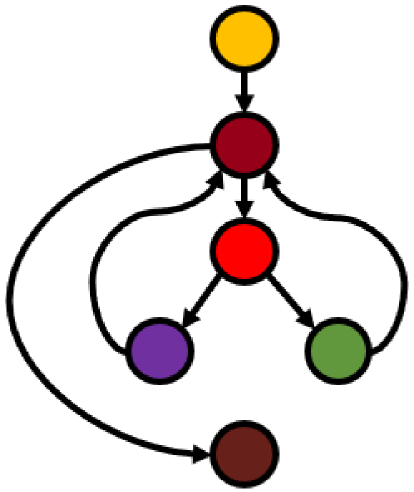
Usage of mutations

- identify weakly tested pieces of code
- identify weak tests
- compute the mutation score: $\frac{mutants\ killed}{\#mutants}$
- error propagation and state infection in the program

My takeaway: mutation is the basis of a testing activity.

- E.g., code coverage = proportion of mutants possible to reach.
- If new testing technique is required → mutation testing

Structure-based Testing



Structure-based Testing: Outline

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

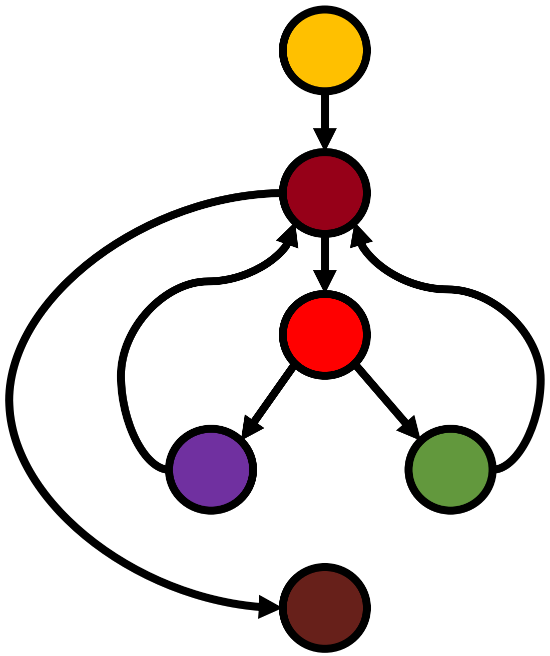
What is “internal structure”?

- 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;

if (! started){
  start();
}

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

Statement

Block

Condition

Decision

Branch

Basic concepts

- Statement
- Block
  - A sequence of one or more consecutive executable statements containing no branches or function calls
- 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)

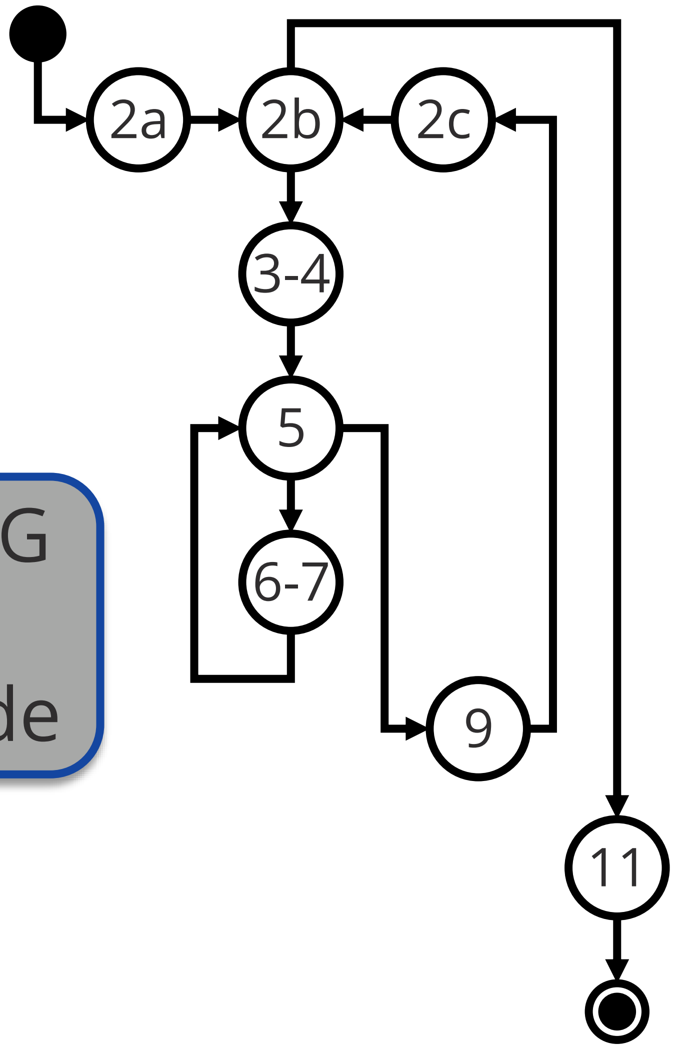
- A CFG represents the flow of control

- ***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

```java
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 *(optional)*
- Evaluation of structure-based testing

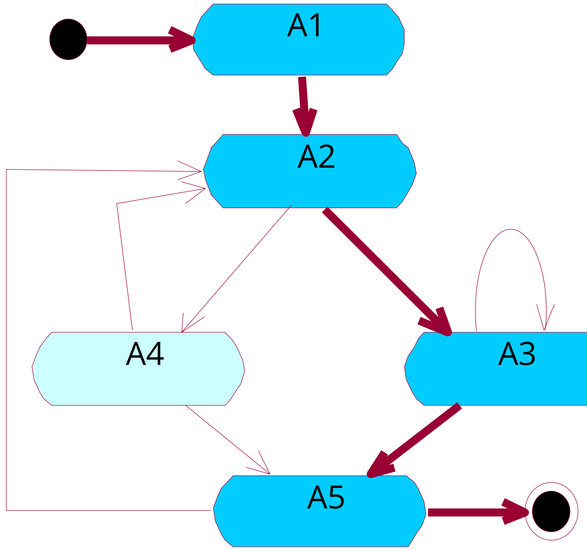
Learning outcomes

- 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 (recap)

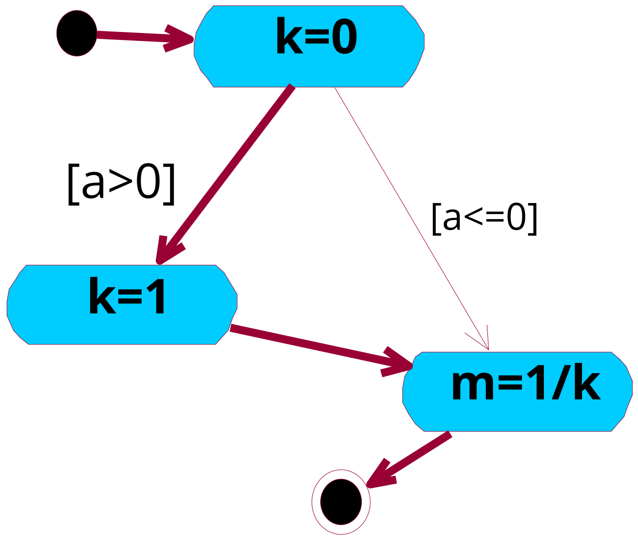
$$\frac{\text{Number of statements executed during testing}}{\text{Number of all statements}}$$



Statement coverage: 4/5 = 80%

Assessing statement coverage

All statement is executed at least once



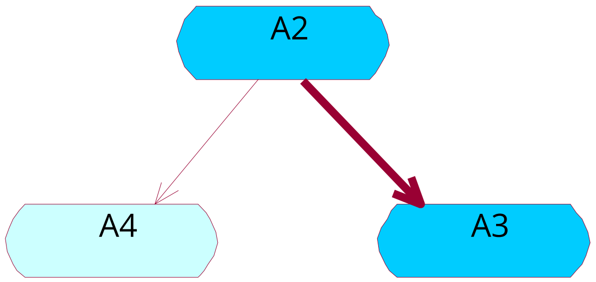
Statement coverage: 100%

BUT: [a<=0] branch missing!

Does not guarantee coverage of empty branches

2. Decision coverage (recap)

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



Decision coverage: 1/2 = 50%

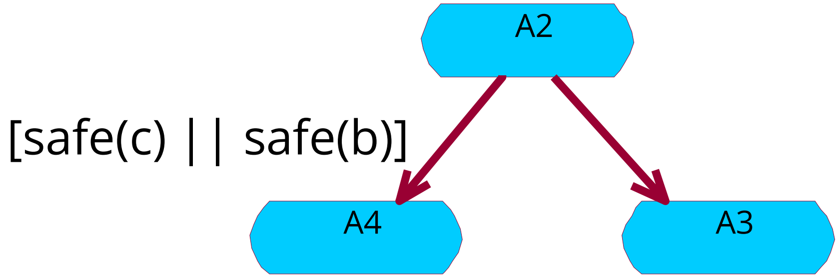
How many outcomes can a decision have?

Note: covering combinations of decisions is not required

Assessing decision coverage

All statement is executed at least once

All outcomes of decisions are covered



100% decision coverage:

#	safe(c)	safe(b)	decision
1	T	F	T
2	F	F	F

`safe(b) == True` missing!

Does not take into account all combinations of conditions!

3. Condition coverage

Generic coverage metric for conditions:

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

Definition (what conditions are aimed):

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

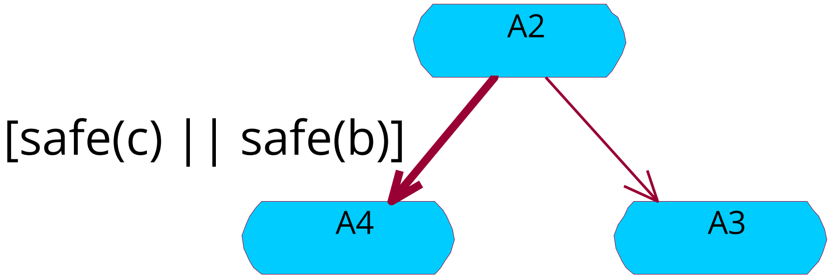
Other possible definition:

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

a	b	a || b
F	F	F
F	T	T
~~T~~	~~F~~	~~T~~
~~T~~	~~T~~	~~T~~
T	*	T

Assessing condition coverage

Every condition has taken all possible outcomes at least once



100% condition coverage:

#	safe(c)	safe(b)	decision
1	T	F	T
2	F	T	T

False outcome of decision missing!

Does not yield 100% decision coverage!

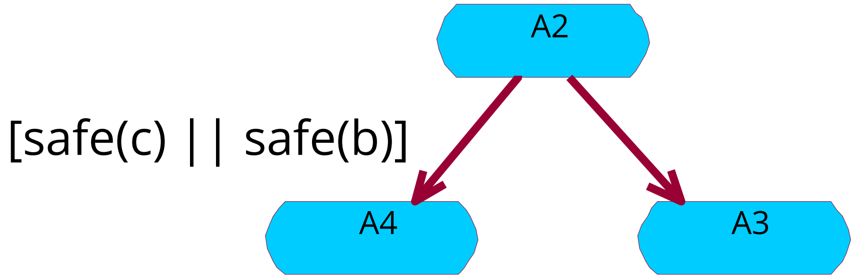
4. Condition/Decision Coverage (C/DC)

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



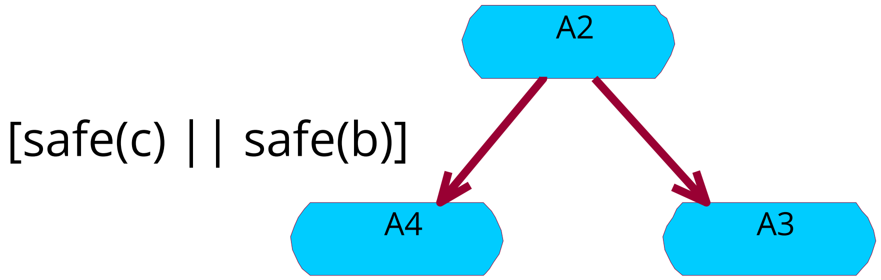
100% C/DC coverage:

#	safe(c)	safe(b)	decision
1	T	T	T
2	F	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.



#	safe(c)	safe(b)	decision
1	T	F	T
2	F	T	T
3	F	F	F

100% MC/DC coverage

Test generation techniques

Motivation

- **Given a software to test**
  - Availability: source code or binary
- **Extend existing testing**
  - Cover incomplete parts, find specific bugs etc.
- **Idea: generate tests somehow!**
  - Based on various criteria (e.g., coverage)

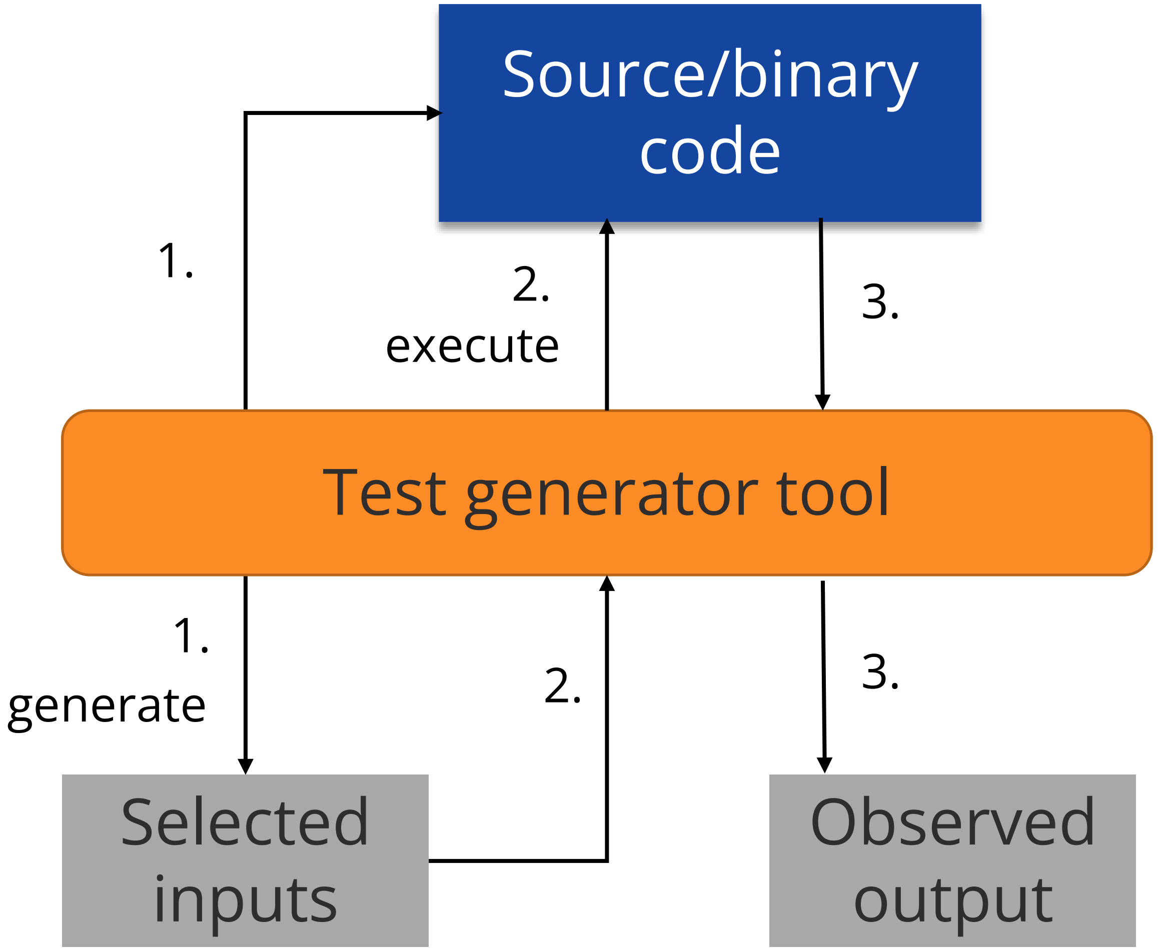
Test selection based on source code

```
   int fun1(int a, int b){
     if (a == 0){
1      printf(ERROR_MSG);
2      return -1;
     }
     if (b > a)
3      return b*a + 5;
     else
4     return (a+b) / 2;
   }
```

a	b	statement
0	*	1, 2
a!=0	b > a	3
a!=0	b <= a	4

This can be (easily) automated!

Idea of white-box test generation



What is missing?

test case = input + *test oracle*

What can be checked without expectations?

- Generic, implicit errors (exception, segfault…)
- Failing assert statement for different inputs
- Manually extending assertions can improve this
- Derive from already existing outputs
  - Regression testing, different implementations

see Test oracles lecture

Techniques

- Random generation
- Search-based
- Symbolic execution

Overview of techniques and goals

Two main questions:

- How to select/search test inputs?
- What to cover with selected inputs? How to evaluate?

		What to cover with search? Evaluation?		
		Code coverage	*Crash, exception…*	*Property, assertion*
How to search?	*Random*			
	Search-based (Metaheuristics)			
	Symbolic execution			

Basic techniques

- Random generation
- Search-based
- Symbolic execution

Basic techniques

- Random generation
- Search-based
- Symbolic execution

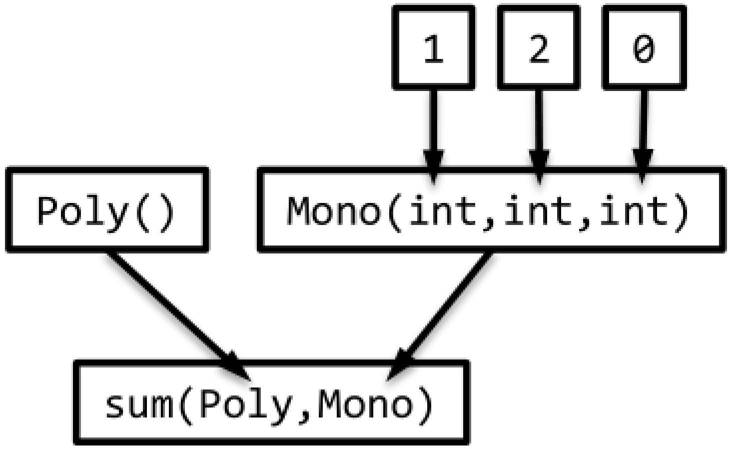
Random test generation

Random selection from input domain

- Advantage:
  - Very fast, very cheap
  - Surprisingly easy to find bugs
- Ideas (guiding random selection):
  - Try to define input format
  - If no error found: trying different parts of domain
  - Selection based on: ”diff”, ”distance”, etc.
- Tool for Java:

Randoop: feedback-driven generation

- Generation of method sequence calls
- Compound objects:



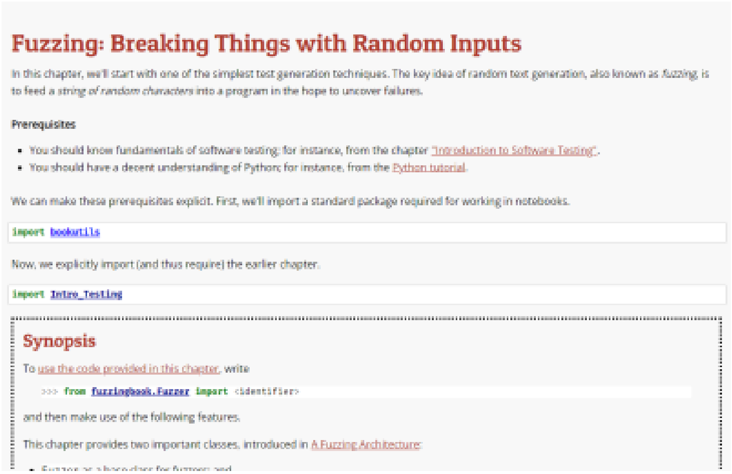
- Heuristics:
  - Execution of selected case
  - Throwing away invalid, redundant cases

Cases studies of random testing

- **Robustness testing**
  - Fuzz: random inputs for console programs ($\rightarrow$ original assignment)
    - Unix (1990), Unix (1995), MacOS (2007)
  - NASA: flash file system
    - Simulating HW errors, comparison with references
    - (Model checking did not scale well)
- **Randoop**
  - JDK, .NET libraries: checks for basic attributes (e.g.: `o.equals(o) returns true`)
  - Comparison of JDK 1.5 and 1.6
  - Was able to found bugs in well-tested components

Advanced fuzzing techniques

- Mutation-based fuzzing
  - Slightly change existing, valid inputs
  - Get past simple input checking to cover more
- Grammar-based fuzzing
  - Specify format of valid inputs
  - CLI parameters, API…
- …



Fuzzing: Breaking Things with Random Inputs

In this chapter, we'll start with one of the simplest test generation techniques. The key idea of random text generation, also known as *fuzzing*, is to feed a *string of random characters* into a program in the hope to uncover failures.

Prerequisites

- You should know fundamentals of software testing; for instance, from the chapter "Introduction to Software Testing".
- You should have a decent understanding of Python; for instance, from the Python tutorial.

We can make these prerequisites explicit. First, we'll import a standard package required for working in notebooks.

```
import bookutils
```

Now, we explicitly import (and thus require) the earlier chapter.

```
import Intro_Testing
```

Synopsis

To use the code provided in this chapter, write

```
>>> from fuzzingbook.Fuzzer import <identifier>
```

and then make use of the following features.

This chapter provides two important classes, introduced in A Fuzzing Architecture:

[The Fuzzing Book (Jupyter+Python)](#)

Killer use case: Automated vulnerability detection (see [security courses](#))

Using annotations in random generation

- If the code contains:
  - pre- and post-conditions (e.g., design by contract)
  - properties or other annotations
- These are able to guide test generation

```
/*@ requires amt > 0 && amt <= acc.bal;
    @ assignable bal, acc.bal;
    @ ensures bal == \old(bal) + amt
    @   && acc.bal == \old(acc.bal - amt); @*/
  public void transfer(int amt, Account acc) {
    acc.withdraw(amt);
    deposit(amt);
  }
```

AutoTest: Bertrand Meyer et al., "Programs that Test Themselves", IEEE Computer 42:9, 2009.

Tools for property-based test generation

- **QuickCheck**
  - Goal: replace manual input values with generated ones
  - Tries to cover laws of input domains
  - Minimizes failing tests (e.g., shortest call sequences)

Methods to tests:

byte[] encrypt(byte[] plaintext, Key key)
byte[] decrypt(byte[] ciphertext, Key key)

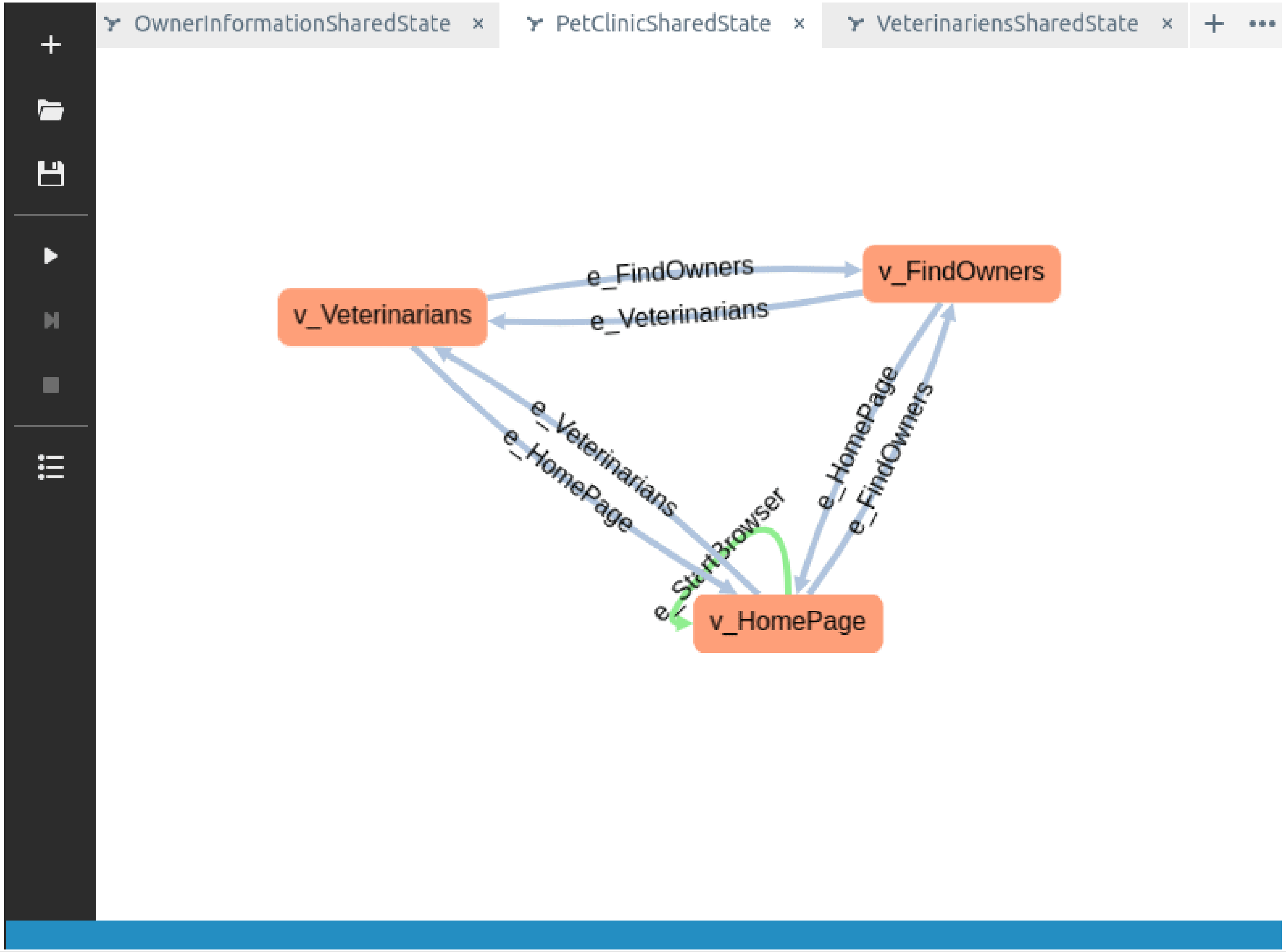
Property:

```
@Property public decryptReversesEncrypt(String plaintext, Key key){
  Crypto.encrypt(plaintext.getBytes("US-ASCII"), key);
  assertEquals(plaintext, new String(Crypto.decrypt(ciphertext, key)));
}
```

More information: video

Claessen et al. "QuickCheck: a lightweight tool for random testing of Haskell programs" ACM Sigplan Notices 46.4 (2011): 53-64

GraphWalker



Techniques

- Random generation
- Search-based
- Symbolic execution

Search-based techniques

Search-based Software Engineering (SBSE)

- Metaheuristic algorithms
  - genetic alg., simulated annealing, hill climbing…
- Representing a problem as a search:
  - Search space: program structure + possible inputs
  - Objective function: reaching a test goal (e.g., covering all branches of decisions)

A tool for search-based test generation

- „Whole test suite generation”
  - All test goals are taken into account
  - Searches based on multiple metrics
    - E.g., high coverage with minimal test suite
- Specialties:
  - Minimizes test code, maintains readability
  - Uses sandbox for environment interaction

Basic techniques

- Random generation
- Search-based
- Symbolic execution

Symbolic execution: the idea

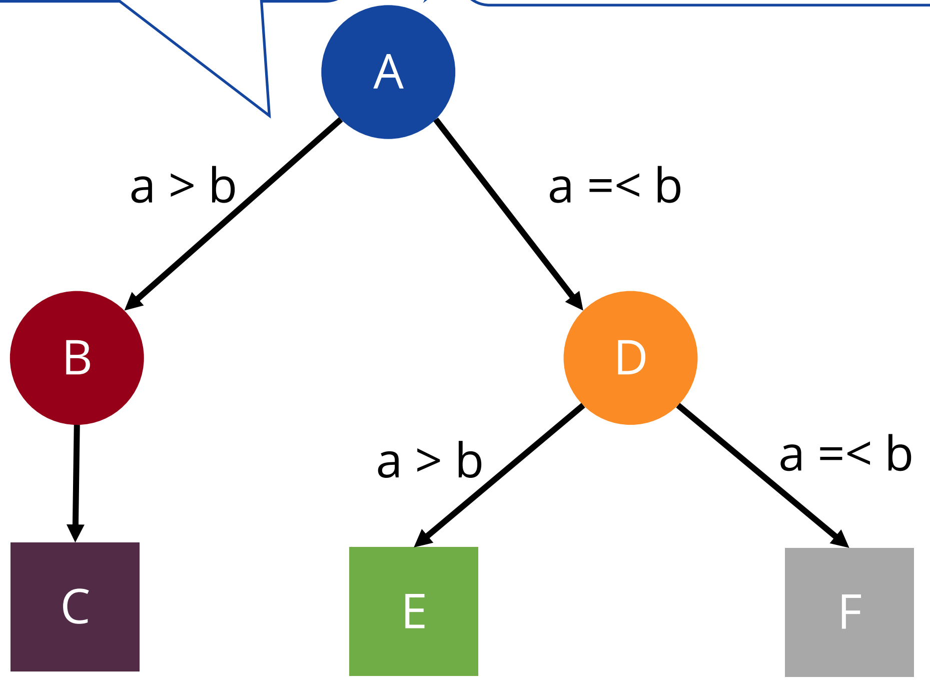
- **Static program analysis technique from the ’70s**
  - Symbolic variables instead of concrete ones
  - Form constraints for each path with symbolic variables
  - Constraint solving (path feasibility)
  - Explore program states and paths (e.g. DFS)
- **Application for test generation**
  - A solution yields an input to execute a given path
- **Note:** SE tries to explore all paths (e.g. unrolls loops)
  - SE tree is not a CFG!
  - SE and abstract interpretation are different techniques!

Example: Static symbolic execution

```
int fun1(int a, int b){
  if (a > b){
    printf(ERROR_MSG);
    return -1;
  }
  if (a > b)
    return b*a + 5;
  else
   return (a+b) / 2;
}
```

- Execute statement
- if: fork execution
- Select a branch

- Symbolic var.: `a, b`
- Statement: `if (a > b)`
- Path constraint (PC): `true`



After reaching a leaf, select a non-explored branch

- PC: `a <= b AND a > b`
- FALSE: not feasible

- PC: `a =< b AND a =< b`
- Solve PC to get test inputs, e.g.
  - `a = 0`, `b = 0`

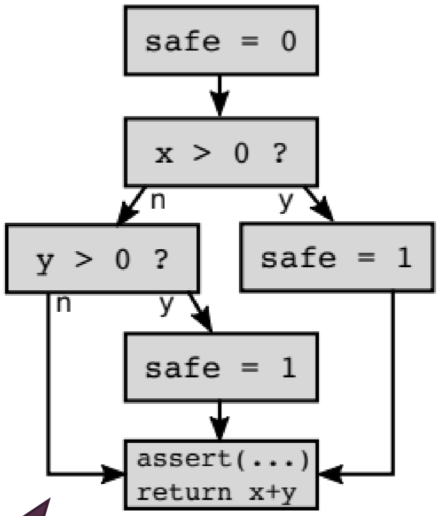
Static symbolic execution: detailed example

```
  uint8_t nzsum(uint8_t x, uint8_t y) {
      int safe = 0;
      if (x > 0)
          safe = 1;
      else if (y > 0)
          safe = 1;
      assert(safe, "Error");
      return x + y;
  }
```

SE tree

Execution state:
- Symbolic values of variables
- PC: path constraint

Executing a statement updates variables symbolically



CFG

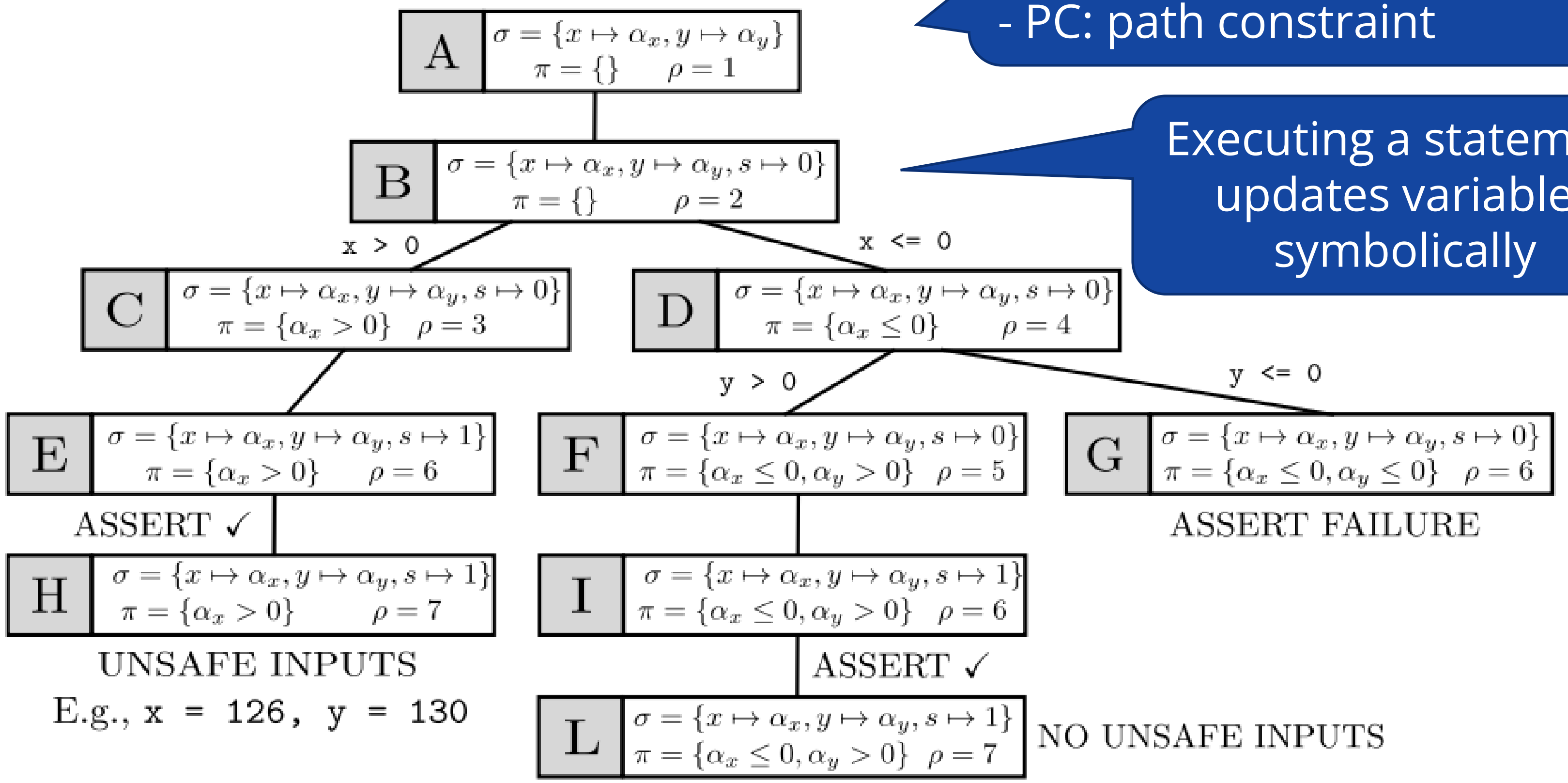
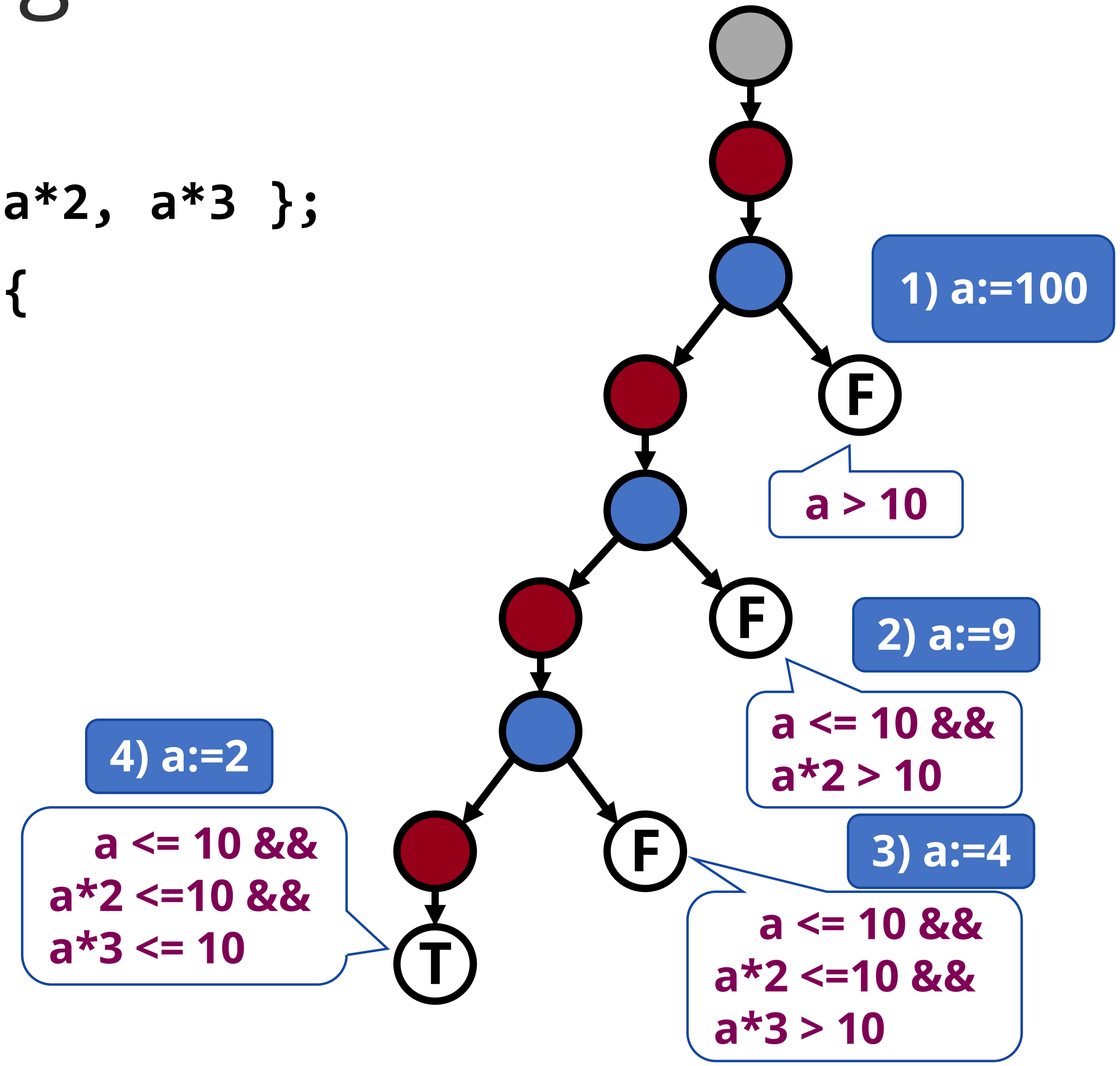


Figure 1: Introductory example: code and control flow graph (left) and symbolic execution tree (right).

Angelini Marco et al. SymNav: Visually Assisting Symbolic Execution. (2019) Published in: 2019 IEEE Symposium on Visualization for Cyber Security (VizSec) pp. 1-11

EXERCISE: Building a SE tree

```
public bool fun2(int a) {
  int[] arr = new int[] { a, a*2, a*3 };
  for(int i = 0; i < 3; i++) {
    if(arr[i] > 10) {
      return false;
    }
  }
  return true;
}
```



Build the SE tree of this method

Extending static symbolic execution

- **Static SE fails** in several cases, e.g.
  - Too long paths → too many constraints
  - Cannot decide if a path is really feasible or not
- **New century, new progress:**
  - Enough computing power (e.g., for SMT solvers)
  - New ideas, extensions, algorithms and *tools*
- **Idea:** mix symbolic with concrete executions
  - Dynamic Symbolic Execution (DSE) or
  - Concolic Testing (concrete + symbolic)

Dynamic symbolic execution

Code to generate inputs for:

Negated condition

```
void CoverMe(int[] a)
{
  if (a == null) return;
  if (a.Length > 0)
    if (a[0] == 1234567890)
      throw new Exception("bug");
}
```

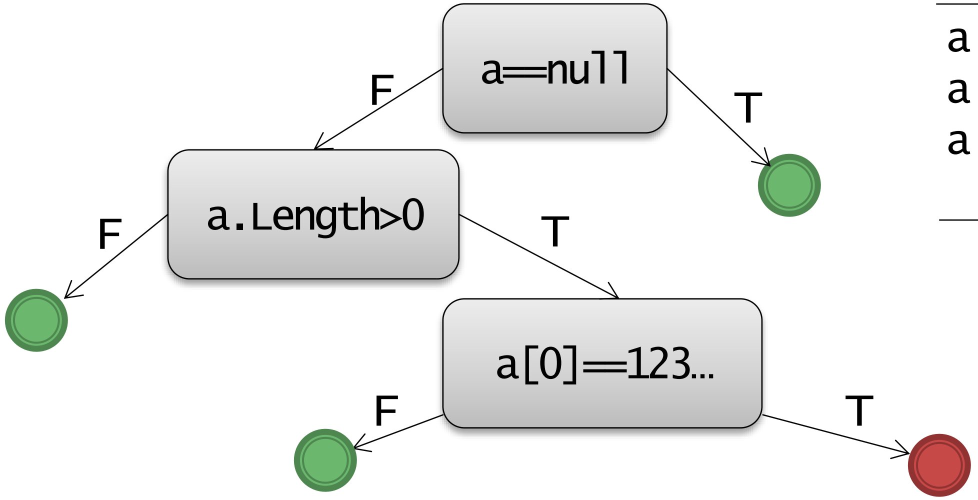
Choose next path

Solve

Execute&Monitor

Constraints to solve	int[] a	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123..}	a!=null && a.Length>0 && a[0]==1234567890

Done: There is no path left.



Source: T. Xie, N. Tillmann, P. Lakshman: Advances in Unit Testing: Theory and Practice

Tools available

Name	Platform	Language	Notes
KLEE	Linux	C (LLVM bitcode)	
Pex	Windows	.NET assembly	Included in Visual Studio (IntelliTest)
SAGE	Windows	x86 binary	Security testing, SaaS model
Jalangi2	*	JavaScript	
Symbolic PathFinder	*	Java	
Sonar Java SE	*	Java	https://github.com/SonarSource/sonar-java/tree/master/java-symbolic-execution

More tools: http://mit.bme.hu/~micskeiz/pages/cbtg.html

Parameterized Unit Testing

- **Idea: Using tests as specifications**
  - Easy to understand, easy to check, etc.
  - *But:* too specific (represents a single execution), oracle needed, etc.
- **Parameterized Unit Test (PUT)**
  - Wrapper method for method/unit under test
    - Partitions the space of concrete test cases
  - Main elements
    - Inputs of the unit
    - Assumptions for input space restriction (partitioning)
    - Call to the unit
    - General assertions for expected results (oracle)
  - Serves as a **specification** → Test generators can use it

Example: Parameterized Unit Testing

```
/// The method reduces the quantity of the specified
/// product. The product is known to be NOT null, also
/// the sold amount is always more than zero. The method
/// has effects on the database, and returns the new
/// quantity of the product. If the quantity would be
/// negative, the method reduces the quantity to zero.
int ReduceQuantity(Product prod, int soldCount) { … }
```

```
void ReduceQuantityPUT(Product prod, int soldCount) {
        // Assumptions
        Assume.IsTrue(prod != null);
        Assume.IsTrue(soldCount > 0);
        // Calling the UUT
        int newQuantity = StorageManager.ReduceQuantity(prod,soldCount);
        // Assertions
        Assert.IsTrue(newQuantity >= 0);
        int oldQuantity = StorageManager.GetQuantityFor(prod);
        Assert.IsTrue(newQuantity < oldQuantity);
}
```

Example: Parameterized Unit Testing

```
/// The method reduces the quantity of the specified
/// product. The product is known to be NOT null, also
/// the sold amount is always more than zero. The method
/// has effects on the database, and returns the new
/// quantity of the product. If the quantity would be
/// negative, the method reduces the quantity to zero.
int ReduceQuantity(Product prod, int soldCount) { … }
```

```
void ReduceQuantityPUT(Product prod, int soldCount) {
        // Assumptions
        Assume.IsTrue(prod != null);
        Assume.IsTrue(soldCount > 0);
        // Calling the UUT
        int newQuantity = StorageManager.ReduceQuantity(prod,soldCount);
        // Assertions
        Assert.IsTrue(newQuantity >= 0);
        int oldQuantity = StorageManager.GetQuantityFor(prod);
        Assert.IsTrue(newQuantity < oldQuantity);
}
```

Basic techniques

- Random generation
- Search-based
- Symbolic execution

Summary of techniques and goals

Two main questions:

- How to search/select test inputs?
- What to cover with selected inputs? How to evaluate?

	Examples	What to cover with search? Evaluation?		
		Code coverage	*Crash, exception…*	*Property, assertion*
How to search?	*Random*	Coverage-based fuzzers	Robustness testing	Property-based testing
	Search-based (Metaheuristics)	SBSE tools (EvoSuite)		
	Symbolic execution	Static SE	SAGE (security)	Parametric Unit Test (Pex)

ftsrg

Evaluations

Applying these techniques on real code?

- SF100 benchmark (Java)
  - 100 projects selected from SourceForge
  - EvoSuite reaches branch coverage of 48%
  - Large deviations among projects

G. Fraser and A. Arcuri, "Sound Empirical Evidence in Software Testing," ICSE 2013

- A large-scale embedded system (C)
  - Execution of CREST and KLEE on a project of ABB
  - ~60% branch coverage reached
  - Fails and issues in several cases

X. Qu, B. Robinson: A Case Study of Concolic Testing Tools and Their Limitations, ESEM 2011

Are these techniques really that good?

- Does it help software developers?
  - 49 participants wrote and generated tests
  - Generated tests with high code coverage did not discover more injected failures

G. Fraser et al., “Does Automated White-Box Test Generation Really Help Software Testers?,” ISSTA 2013

- Finding real faults
  - Defects4J: database of 357 issues from 5 projects
  - Tools evaluated: EvoSuite, Randoop, Agitar
  - Only found 55% of faults

S. Shamshiri et al., „Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges.” ASE 2015

Comparison of test generator tools

- Various source code snippets to execute
  - Covering most important features of languages
- 363 Java/.NET snippets
  - Executed on 6 different tools
- Experience:
  - Huge difference in tools
  - Some snippets challenging for all tools

L. Cseppentő, Z. Micskei: „Evaluating code-based test input generator tools," STVR 2017

Comparison of test generator tools