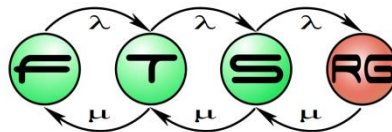


Code-based test generation

**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

Learning outcomes

- Explain the *basic ideas* of different code-based test generation techniques (K2)
- *Demonstrate the workflow* of symbolic execution on a method by *graphically representing the execution* using a symbolic execution tree (K3)
- Use different code-based test generator tools (K3)

Motivation

- **Given a barely tested software to test**
 - Availability: source code or binary
- **Developer testing**
 - Can be expensive, incomplete, etc.
- **Idea: generate tests somehow!**
 - Based on various criteria (e.g., coverage)

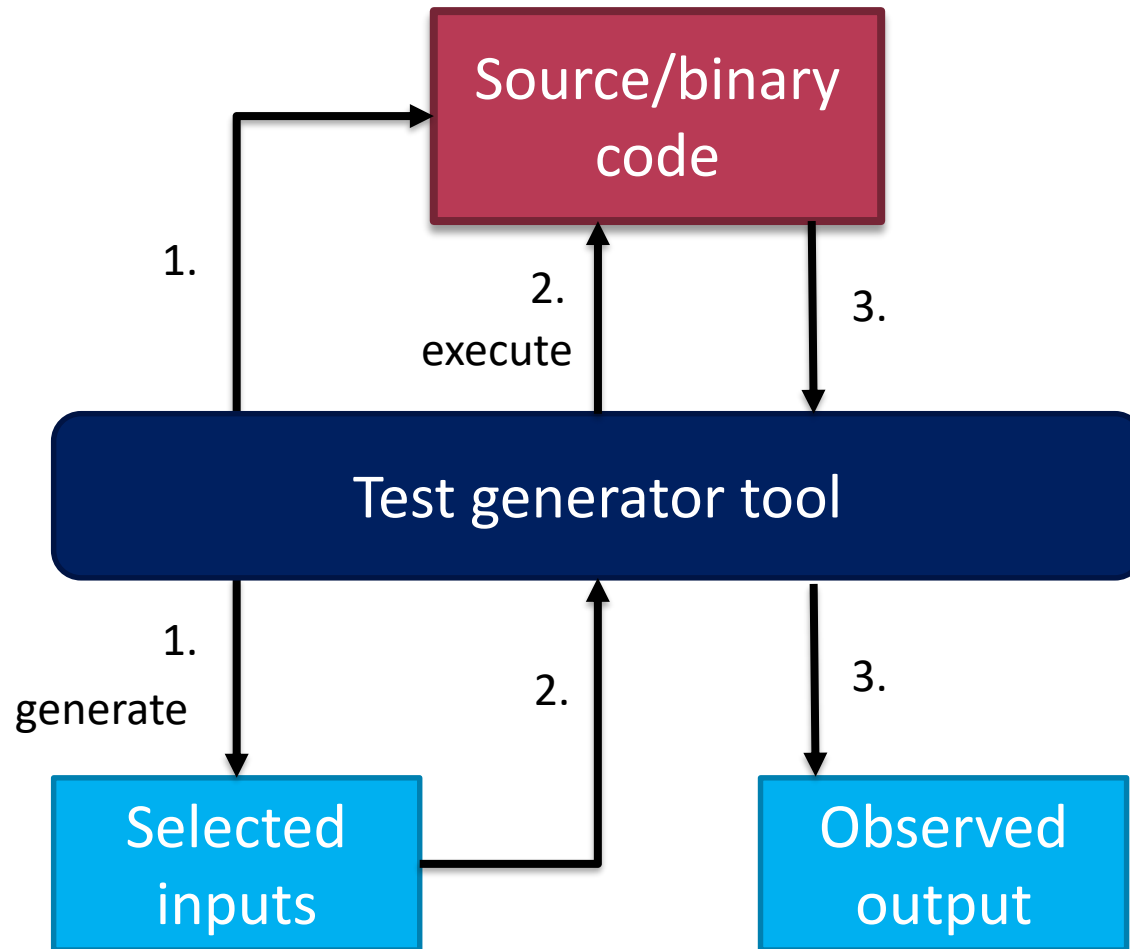
Test selection based on source code

```
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?

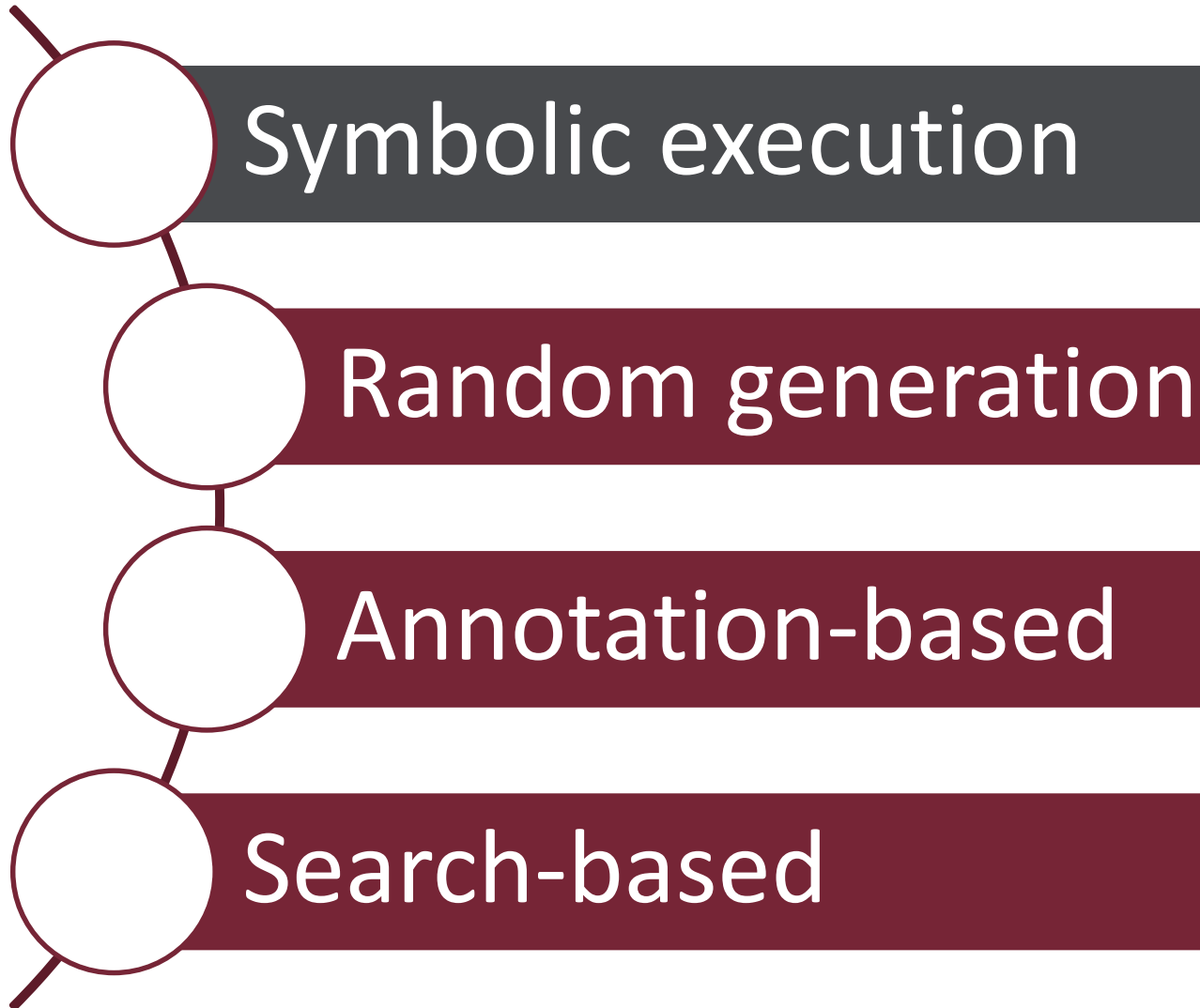
test case = input + *test oracle*

What can be checked without expectations?

- Basic, generic errors (exception, segfault...)
- Failing assert statement for different inputs
- Manually extending assertions can improve this
- Reuse of already existing outputs
 - Regression testing, different implementations

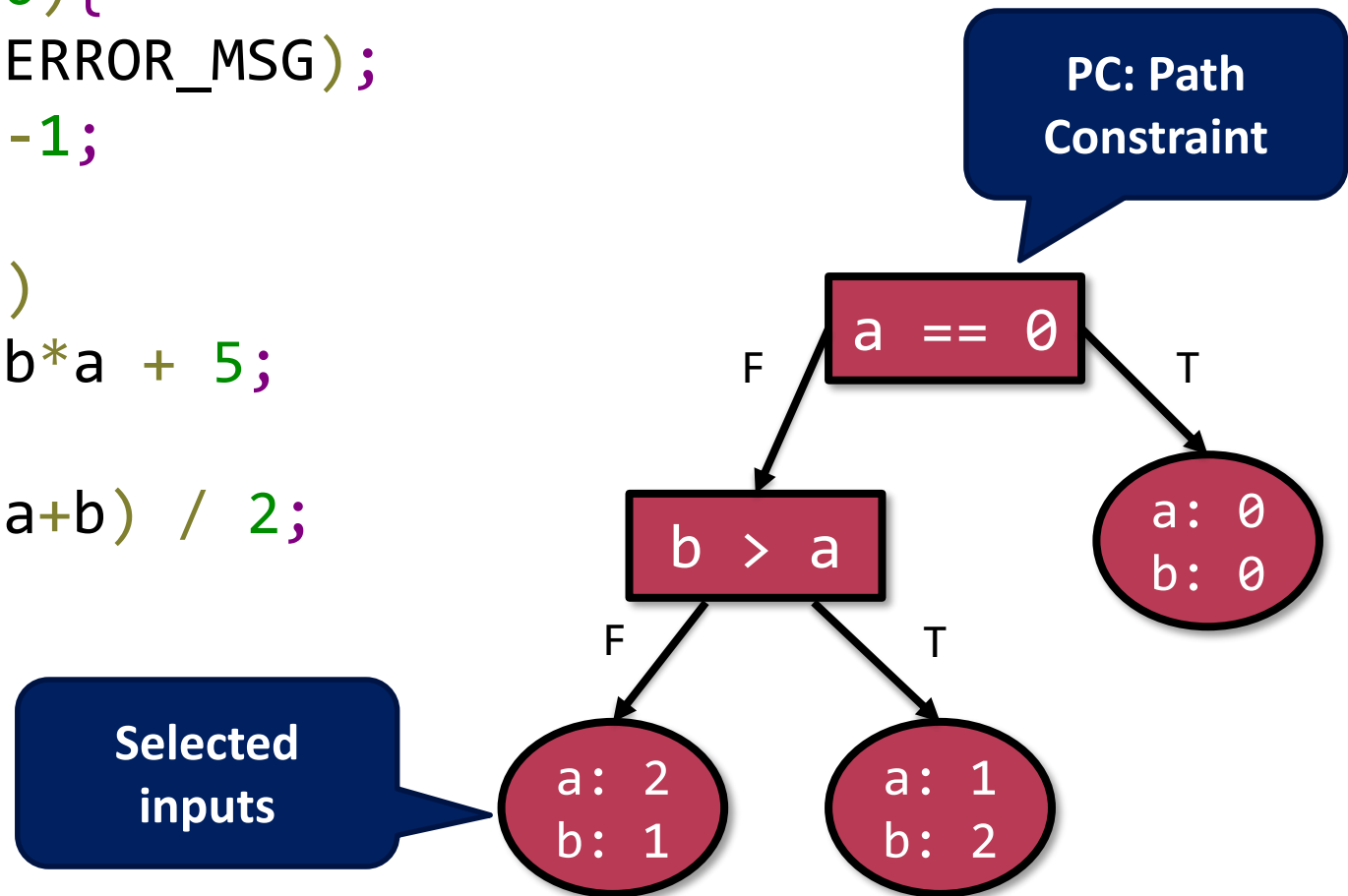
TECHNIQUES

Techniques



Example: Static symbolic execution

```
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;  
}
```



Symbolic execution: the idea

- **Static program analysis technique from the '70s**
- **Application for test generation**
 - Symbolic variables instead of normal ones
 - Constraints forming for each path with symb. variables
 - Constraint solving (e.g., SMT solver)
 - A solution yields an input to execute a given path
- **New century, new progress:**
 - Enough computing power (e.g., for SMT solvers)
 - New ideas, extensions, algorithms and *tools*

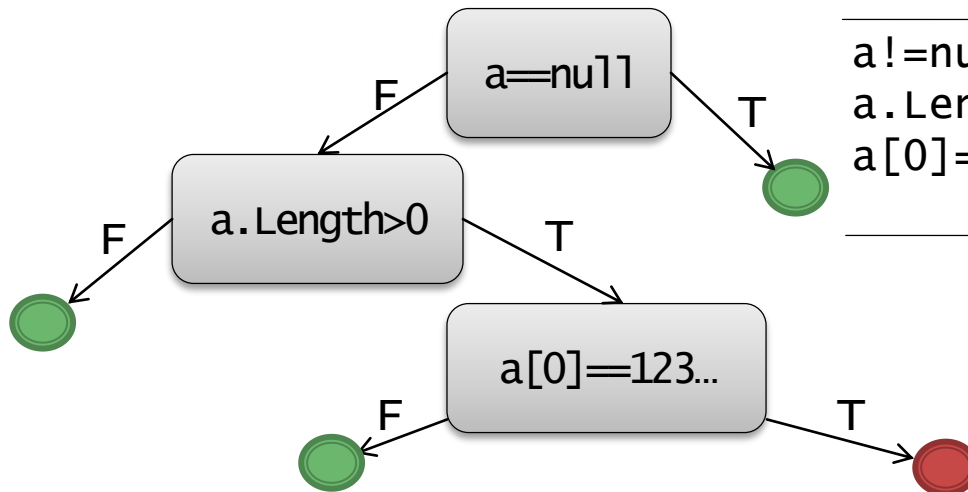
Extending static symbolic execution

- **Static SE fails** in several cases, e.g.
 - Too long paths $\rightarrow$ too many constraints
 - Cannot decide if a path is really feasible or not
- **Idea:** mix symbolic with concrete executions
 - Dynamic Symbolic Execution (DSE) or
 - Concolic Testing

Dynamic symbolic execution

Code to generate inputs for:

```
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		
a!=null && a.Length>0 && a[0]==1234567890	{123..}	a!=null && a.Length>0 && a[0]==1234567890

Negated condition

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
Jalangi	-	JavaScript	
Symbolic PathFinder	-	Java	

Other (discontinued) tools:

CATG, CREST, CUTE, Euclide, EXE, jCUTE, jFuzz, LCT, Palus, PET, etc.

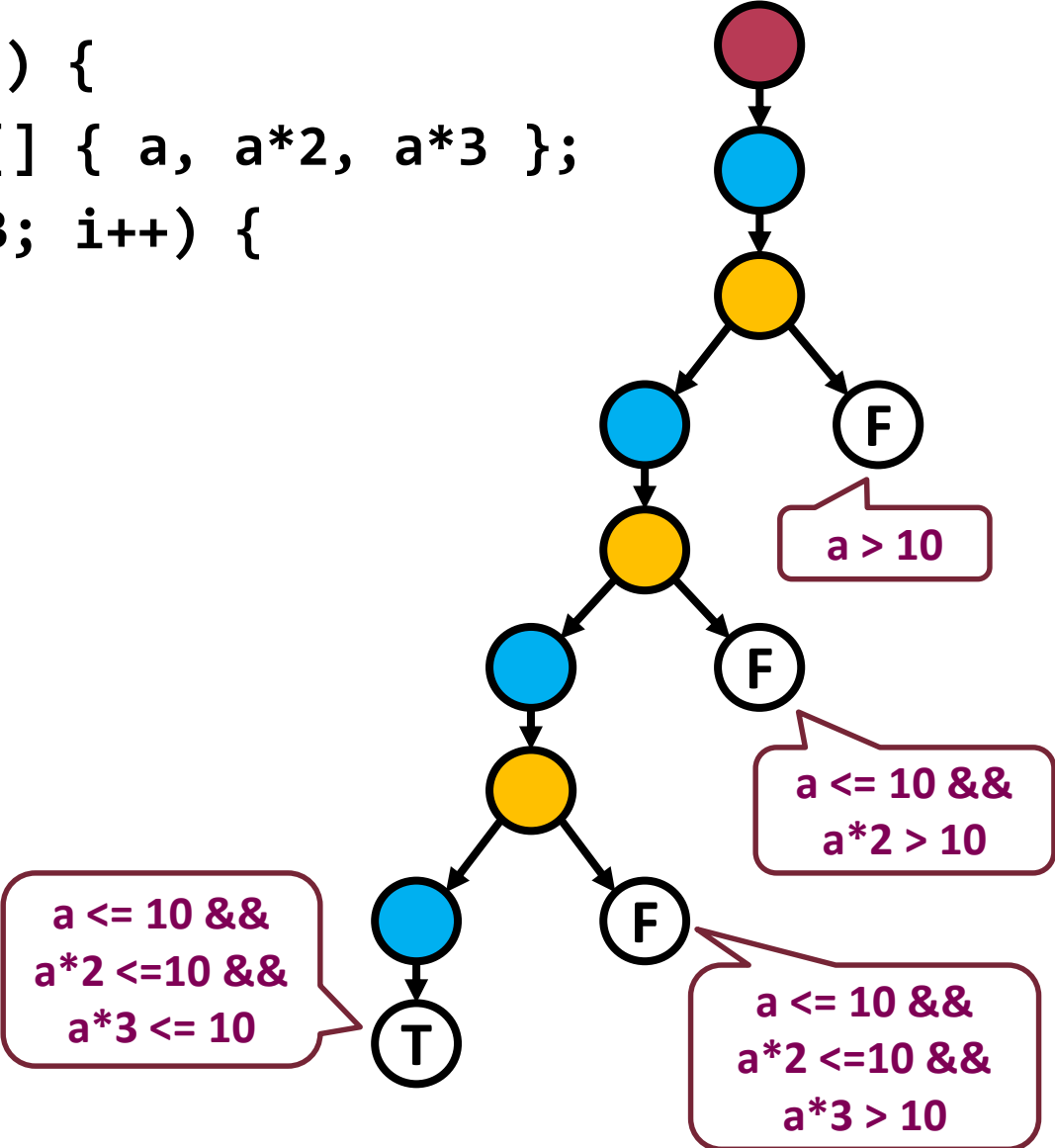
More tools: <http://mit.bme.hu/~micskeiz/pages/cbtg.html>

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



Pex for fun / Code Hunt

<http://pexforfun.com>

Curious?
Learn More! →

Pex  **for fun**

Random Puzzle Learn APCS New

1,469,121 clicked 'Ask Pex!'

The code is a puzzle. Do you understand what the code does? Click Ask Pex! to find out.

```
using System;

public class Program {
    static void B(int x, int y) {
        if (x == 1 && y == 2) throw new Exception("hidden bug!");
    }
    static void A(int x, int y) {
        B(x * y, x / y);
    }
    public static void Puzzle(int x, int y) {
        // What values of x and y can cause the exception? Ask Pex to find out!
        A(x - y, x + y);
    }
}
```

Ask Pex!

<http://codehunt.com>

LEVEL: 00.02 ▶ ATTEMPTS: 1 RATING: ■■■ SCORE: 0

CODE HUNT

Discover the arithmetic operation applied to 'x'.

CAPTURE CODE

RESET LEVEL C#

```
1 using System;
2 public class Program {
3     public static int Puzzle(int x) {
4         return 0;
5     }
6 }
```

	x	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
✗	0	1	0	Mismatch
✓	-1	0	0	

Parameterized Unit Testing

- **Idea: Using tests as specifications**
 - Easy to understand, easy to check, etc.
 - *But:* too specific (used for a code unit), verbose, etc.
- **Parameterized Unit Test (PUT)**
 - Wrapper method for method/unit under test
 - Main elements
 - Inputs of the unit
 - Assumptions for input space restriction
 - Call to the unit
 - Assertions for expected results
 - 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) { ... }
```

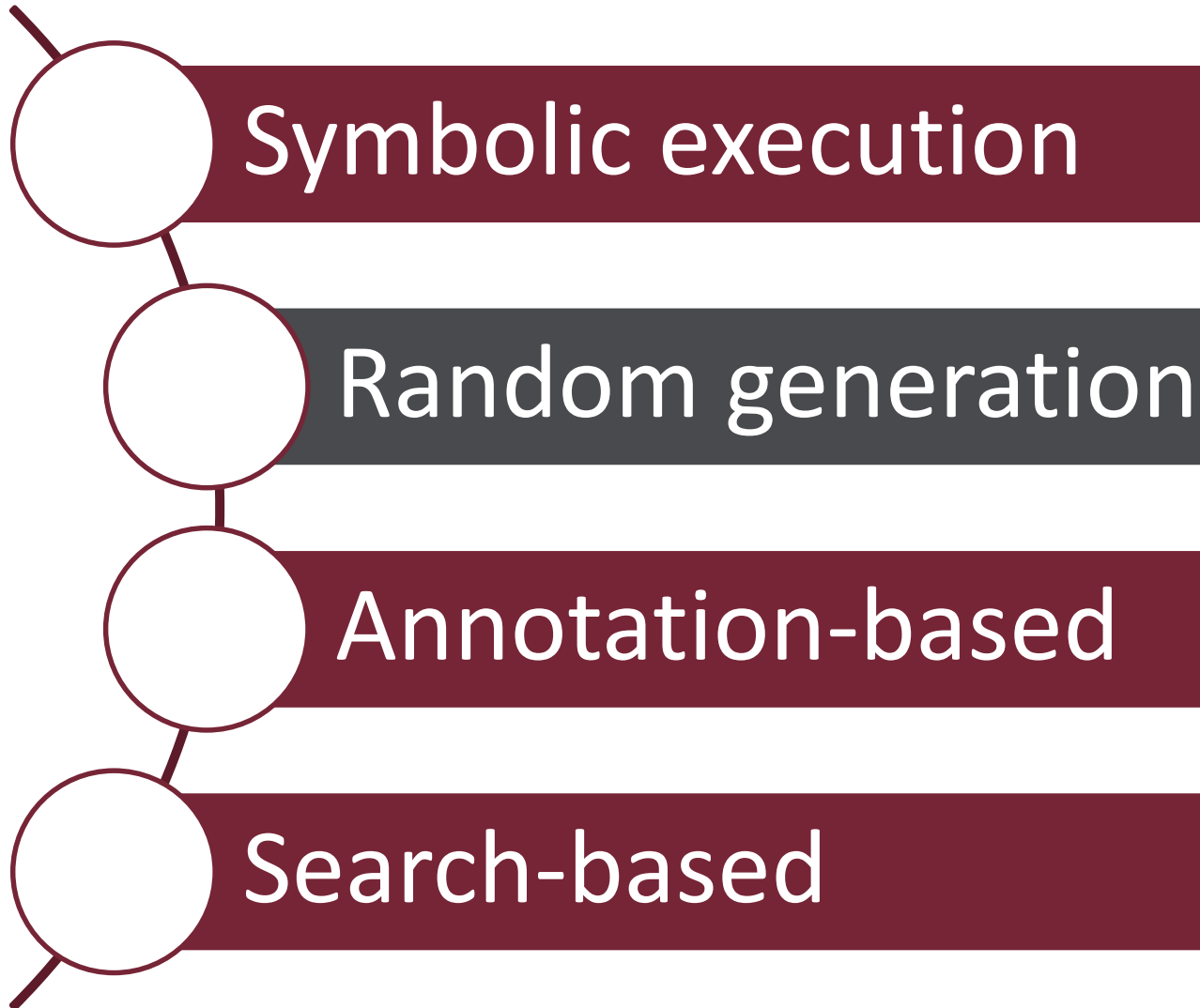
```
void ReduceQuantityPUT(Product prod, int soldCount) {
    // Assumptions
    Assume.IsTrue(prod != null);
    Assume.IsTrue(soldCount > 0);
    // Calling the UUT
    int newQuantity = StorageManager.ReduceQuantity(prod, soldCount);
    // Assertions
    Assert.IsTrue(newQuantity >= 0);
    int oldQuantity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

Techniques



Random test generation

Random selection from input domain

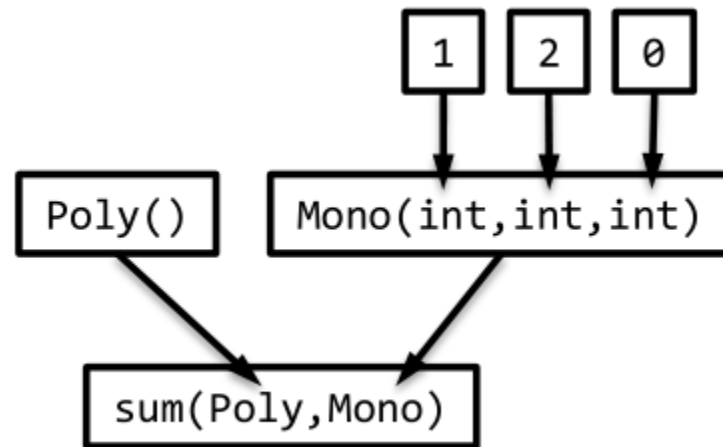
- Advantage:
 - Very fast
 - Very cheap
- Ideas:
 - If no error found: trying different parts of domain
 - Selection based on: "diff", "distance", etc.

- 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 robustness testing

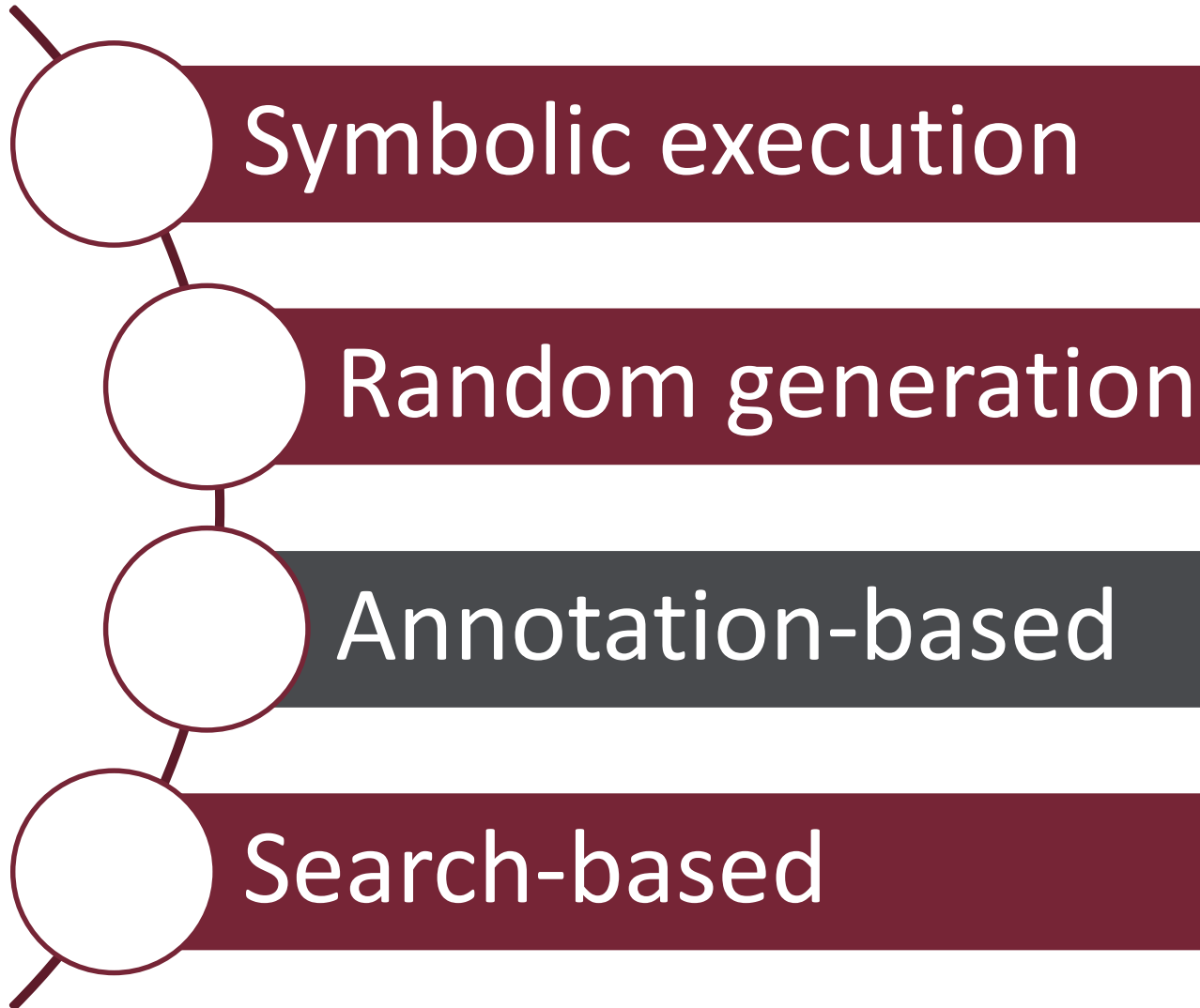
■ Robustness testing

- Fuzz: random inputs for console programs
 - 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

Techniques



Using annotations for test generation

- If the code contains:
 - pre- and post-conditions (e.g.: design by contract)
 - 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);  
  
}
```

Tools for annotation-based test generation

■ AutoTest

- Eiffel language, Design by Contract
- Input: „object pool”, random generation
 - Idea: Include inputs that satisfy preconditions.
- Expected output: contracts

AutoTest: Bertrand Meyer et al., "Programs that Test Themselves", [IEEE Computer](#) 42:9, 2009.

Tools for property-based test generation

■ QuickCheck

- Goal: replace manual values with generated ones
- Tries to cover laws of input domains

Methods to tests:

`byte[] encrypt(byte[] plaintext, Key key)`

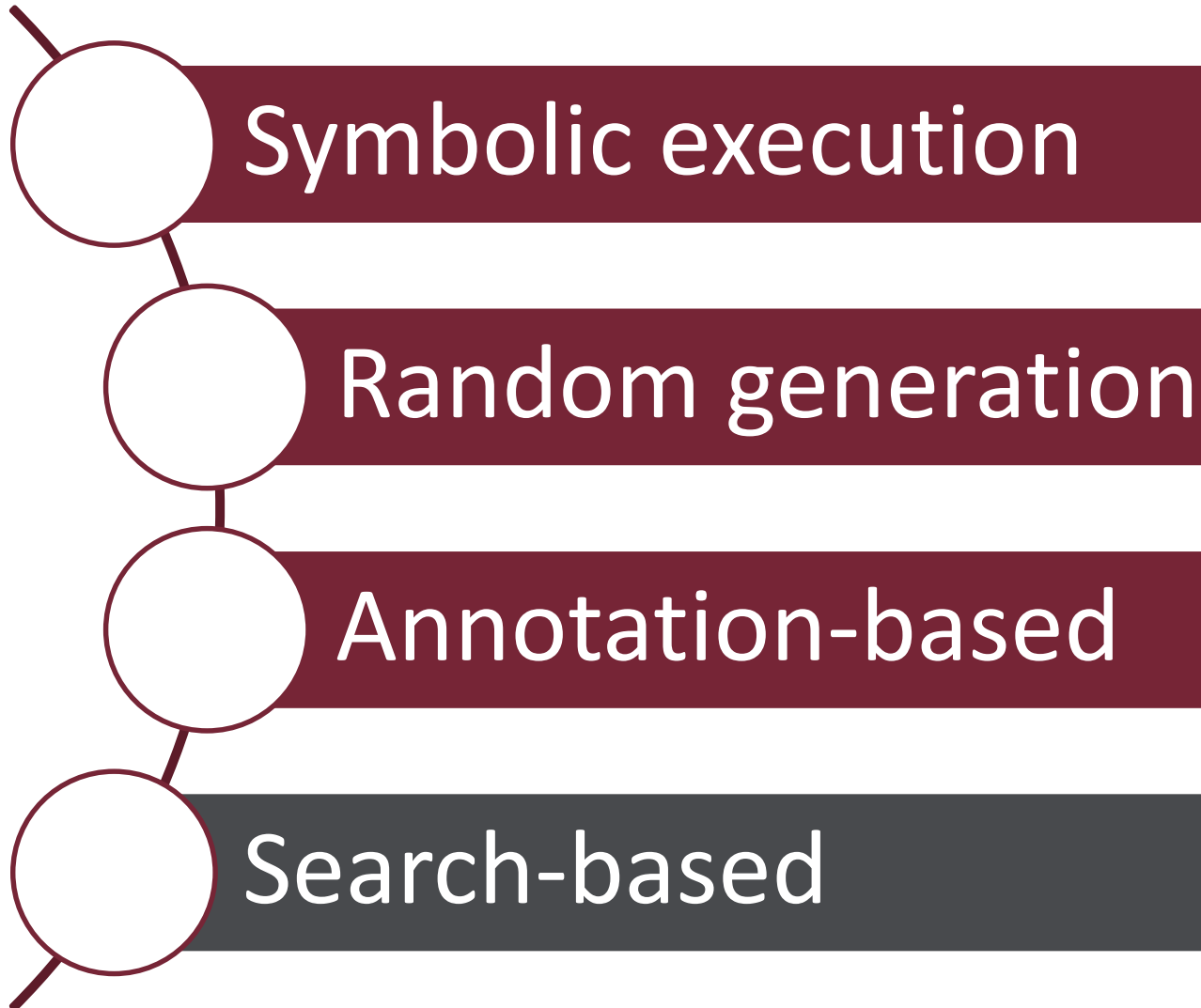
`byte[] decrypt(byte[] ciphertext, Key key)`

Property:

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

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

Techniques



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 decisions of a given condition)

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

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



Current challenges

- Complex arithmetic operations (e.g., logarithms)
- Floating point numbers (e.g., equality)
- Non-trivial string operations
- Environment calls (e.g., files, native, external libs)
- Multithreading
- Compound data structures
- Pointer operations
- ...

Summary

- Tests generation **is possible** based on code
- Various **different techniques** available
- Can **find bugs** in real-world software
- Further challenges (active research topic):
 - Scalability
 - Test oracle production
 - etc.