Code-based Test Generation

Dávid Honfi, Zoltán Micskei

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

Motivation

- Goal: Developer testing of modules (units, components)
  - At this level, specification (for specification based testing) may be missing
- Idea: Generate test inputs based on the code
  - Source code coverage criteria can be satisfied (to execute all parts of the code)
- How test outputs are checked?
  - Based on overall expectations (on the basis of higher level specifications) given by the tester
  - Using generic criteria: avoiding crash, OS level error signal, exception, timeout, violated assertion
  - Re-using outputs of previous test (regression testing)

Random test generation

Random selection from the input domain

- Advantage:
  - Very fast
  - Very cheap
- Ideas:
  - If no error found: trying different parts of the domain
  - Selection based on: ”difference”, ”distance”, etc.
- Tool for Java:

Annotation-based test generation

- If the code contains:
  - Pre- and post-conditions (e.g.: design by contract)
  - Other annotations (e.g., loop invariants)
- These are able to guide test generation

```java
/*@ 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);

 }
```

Annotation-based test generation: Tools

- AutoTest
  - Eiffel language, with Design by Contract
  - Input: object pool
    - Random generation of inputs that satisfy the preconditions
  - Expected output: checked on the base of the contracts
  - Ref: Bertrand Meyer et al., "Program that Test Themselves", IEEE Computer 42:9, 2009.
- QuickCheck: Property based test generation
  - Goal: Generate test values that take into account the types and laws of the input domains
  - Ref: Claessen et al. "QuickCheck: a lightweight tool for random testing of Haskell programs“ ACM Sigplan 46.4 (2011): 53-64

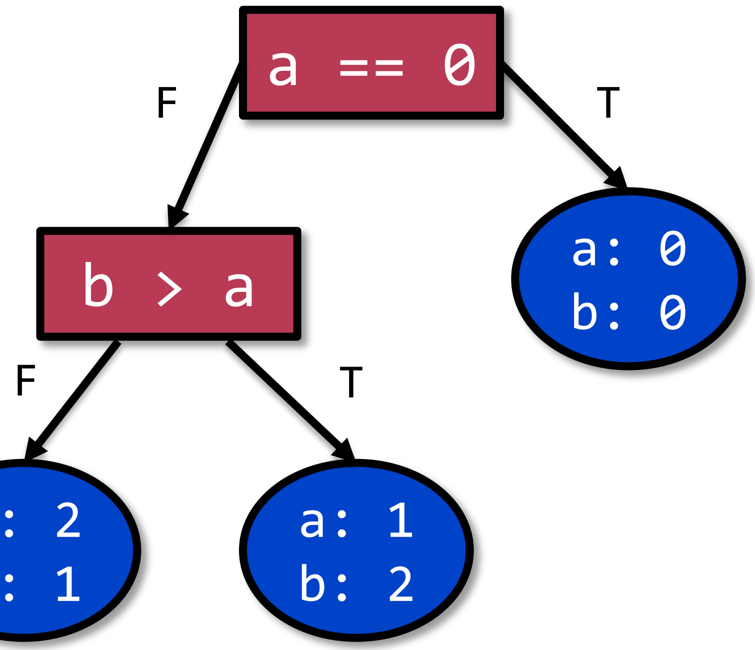
Symbolic execution

- Static program analysis technique
- Basic idea
  - Following computation of program paths with symbolic variables
  - Deriving reachability conditions as path constraints
  - Constraint solving (e.g., SMT solver):
    A solution yields an input to execute a given path
- Popular nowadays:
  - Efficient SMT solvers exist
  - Used to generate test inputs for covering given paths
  - Mixing symbolic and concrete execution: “Concolic”

Program paths and related inputs

```
  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;
  }
```

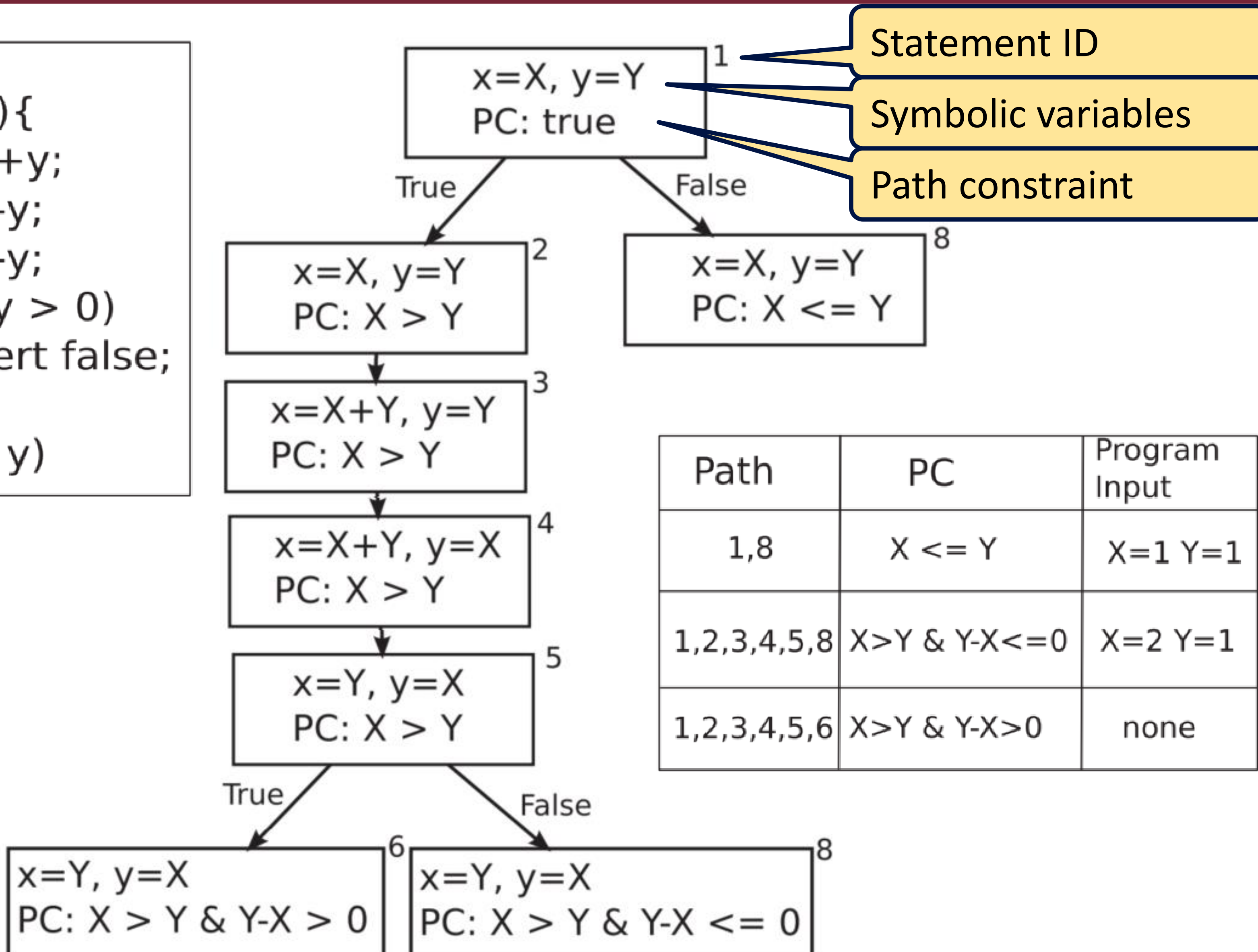
Exploring program paths, together with branch conditions
→ Deriving path constraint (PC) for each path



For each path: Selecting inputs that satisfy path constraints

Example for deriving path constraints

```
  int x, y;
1 if(x > y){
2    x = x+y;
3    y = x-y;
4    x = x-y;
5    if(x - y > 0)
6       assert false;
7 }
8 print(x, y)
```



Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

Tools available

Name	Platform	Language	Notes
KLEE	Linux	C (LLVM bitcode)	
Pex	Windows	.NET assembly	VS2015: IntelliTest
SAGE	Windows	x86 binary	Security testing, SaaS model
Jalangi	-	JavaScript	
Symbolic PathFinder	-	Java	

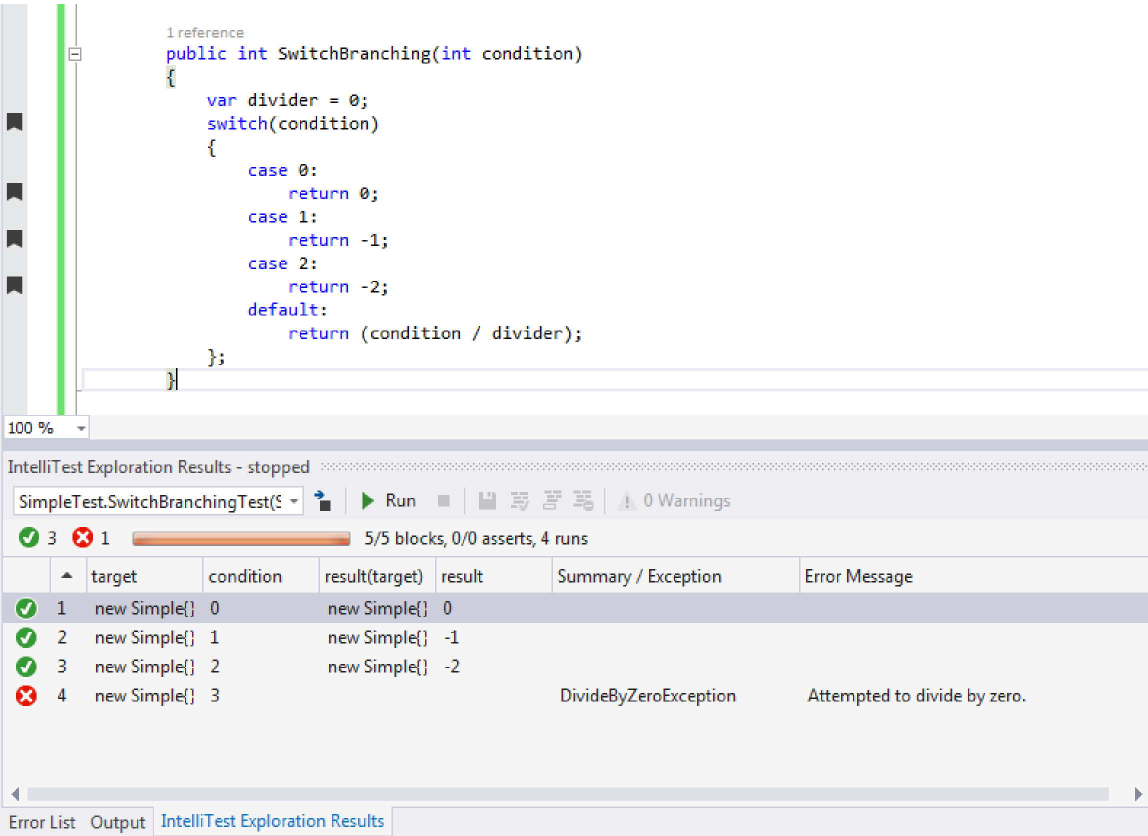
Other (discontinued) tools:
CATG, CREST, CUTE, Euclide, EXE, jCUTE, jFuzz, LCT, Palus, PET, etc.

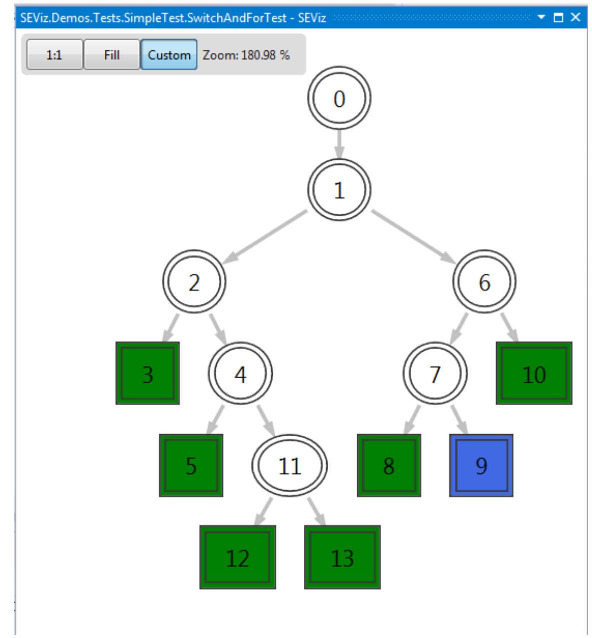
More tools: http://mit.bme.hu/~micskeiz/pages/cbtg.html

Microsoft IntelliTest

Generate unit tests for your code with IntelliTest

https://msdn.microsoft.com/en-us/library/Dn823749.aspx





SEViz

SEViz (Symbolic Execution VisualIZer)

https://github.com/FTSRG/seviz

Challenges for symbolic execution

1. Exponential growth of execution paths
2. Complex arithmetic expressions
3. Floating point operations
4. Compound structures and objects
5. Pointer operations
6. Interaction with the environment
7. Multithreading
8. …

T. Chen et al. „State of the art: Dynamic symbolic execution for automated test generation”. Future Generation Computer Systems, 29(7), 2013

Challenges (1)

Exponential growth of execution paths

```
int hardToTest(int x){
    for (int i=0; i<100; i++){
        int j = complexMathCalc(i,x);
        if (j > 0) break;
    }

    return i;
}
```

- Ideas:
  - Various traversal algorithms instead of DFS
  - Method summary: simple representation of methods

Challenges (2)

Complex arithmetic expressions

- Most SMT solvers cannot handle these

```
int hardToTest2(int x){
    if (log(x) > 10)
        return x
    else
        return -x;
}
```

- Ideas: Using specific solvers for different cases
  - E.g., CORAL is specially designed for these problems

Challenges (6)

Interaction with the environment

- Calls to platform and external libraries

```
int hardToTest3(string s){
  FileStream fs = File.Open(s, FileMode.Open);
  if (fs.Lenth > 1024){
    return 1;
  } else
    return 0;
  }
}
```

- Idea:
  - „Environment models” (KLEE): for simple C programs
  - Special objects representing the environment (Java)

EVALUATIONS

Applying these techniques on real code?

- 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

- Does it help software developers?
  - 49 participants wrote and also 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 – requirements were missing

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