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.

- **Alternative approaches**
  - Combinatorial, model-based, etc.

- **Idea: generate tests somehow!**
  - Based on various criteria (e.g., coverage)
int fun1(int a, int b) {
    if (a == 0) {
        printf(ERROR_MSG);
        return -1;
    }
    if (b > a)
        return b*a + 5;
    else
        return (a+b) / 2;
}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>1, 2</td>
</tr>
<tr>
<td>a!=0</td>
<td>b &gt; a</td>
<td>3</td>
</tr>
<tr>
<td>a!=0</td>
<td>b &lt;= a</td>
<td>4</td>
</tr>
</tbody>
</table>
What is missing?

test case = input + *expected output*

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

Symbolic execution
Random generation
Annotation-based
Search-based
```c
int fun1(int a, int b){
    if (a == 0){
        printf(ERROR_MSG);
        return -1;
    }
    if (b > a)
        return b*a + 5;
    else
        return (a+b) / 2;
}
```

**Selected inputs**

- **a: 2, b: 1** (T)
- **a: 1, b: 2** (F)

**PC: Path Constraint**

- **a == 0**
  - (F) a: 0, b: 0
  - (T) a: 2, b: 1

**b > a**

- (F) a: 1, b: 2
- (T) a: 2, b: 1

Example: Static symbolic execution
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 → 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:

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

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>int[] a</th>
<th>Observed constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>a==null</td>
<td>null</td>
<td>a==null</td>
</tr>
<tr>
<td>a!=null &amp;&amp; !(a.Length&gt;0)</td>
<td>{}</td>
<td>a!=null &amp;&amp; !(a.Length&gt;0)</td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
<td>{123}</td>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
</tr>
</tbody>
</table>

Done: There is no path left.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Platform</th>
<th>Language</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLEE</td>
<td>Linux</td>
<td>C (LLVM bitcode)</td>
<td></td>
</tr>
<tr>
<td>Pex</td>
<td>Windows</td>
<td>.NET assembly</td>
<td>VS2015: IntelliTest</td>
</tr>
<tr>
<td>SAGE</td>
<td>Windows</td>
<td>x86 binary</td>
<td>Security testing, SaaS model</td>
</tr>
<tr>
<td>Jalangi</td>
<td>-</td>
<td>JavaScript</td>
<td></td>
</tr>
<tr>
<td>Symbolic PathFinder</td>
<td>-</td>
<td>Java</td>
<td></td>
</tr>
</tbody>
</table>

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

DEMO: Microsoft IntelliTest

Generate unit tests for your code with IntelliTest

SEViz (Symbolic Execution Visualizer)
https://github.com/FTSRG/seviz
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;
}
The code is a puzzle. Do you understand what the code does? Click Ask Pex! to find out.

```csharp
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);
    }
}
```

http://codehunt.com
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* \(\rightarrow\) 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.

```csharp
int ReduceQuantity(Product prod, int soldCount) { ... }
```

```csharp
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

- Symbolic execution
- Random generation
- Annotation-based
- Search-based
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:
  
  ![Randooop](image)
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
  - 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

- Symbolic execution
- Random generation
- Annotation-based
- Search-based
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.

```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);
}
```
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

Tools for property-based test generation

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

```java
@Test
public void sortedListCreation() {
    for (List<Integer> any : someLists(integers())) {
        SortedList sortedList = new SortedList(any);
        List<Integer> expected = sort(any);
        assertEquals(expected, sortedList.toList());
    }
}

private List<Integer> sort(List<Integer> any) {
    ArrayList<Integer> sorted = new ArrayList<Integer>(any);
    Collections.sort(sorted);
    return sorted;
}
```

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

Symbolic execution
Random generation
Annotation-based
Search-based
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

**EVSUITe**

- „Whole test suite generation”
  - All test goals are taken into account
  - Searches based on multiple metrics
    - E.g., high coverage with minimal test suite

- Specialities:
  - 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


- **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

- 300 Java/.NET snippets
  - Executed on 6 different tools

- Experience:
  - Huge difference in tools
  - Some snippets challenging for all tools

Comparison of test generator tools

Legend:
- N/A
- EX
- T/M
- NC
- C

[Diagram showing comparisons between various test generator tools with categories such as Basic, Structures, Objects, Generics, Library, and Others, with different colors representing various functionalities.]
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
- ...

42
Summary

- Tests generation is possible based on code

- Various different techniques available

- Further challenges:
  - Scalability
  - Test oracle production
  - etc.

- Active topic of research in software engineering