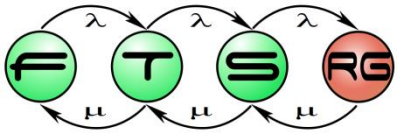
Code-based Test Generation

Dávid Honfi, Zoltán Micskei

honfi@mit.bme.hu

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group



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 – code based**
  - 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

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

Random

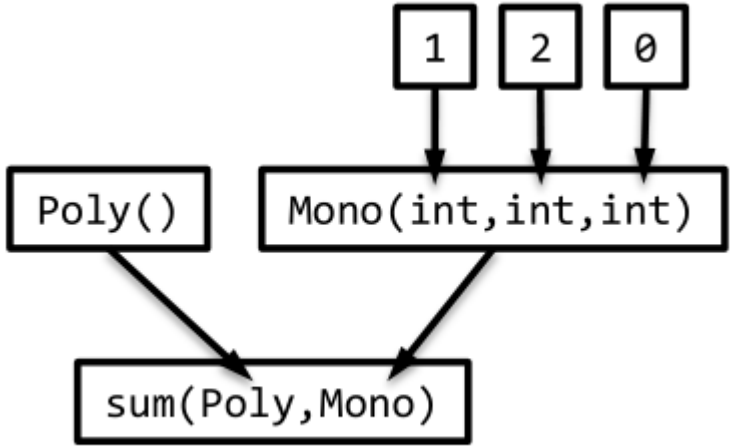
Random test generation

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
- Creating compound objects:



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

Case studies of robustness testing

- **Robustness testing (using invalid inputs)**
  - 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
  - It was able to find bugs in well-tested components

TECHNIQUES

Annotation-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.

```
/*@ 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: checked on the base of contracts

AutoTest: Bertrand Meyer et al., "Program 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

```
@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

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 decision branches)

A tool for search-based test generation

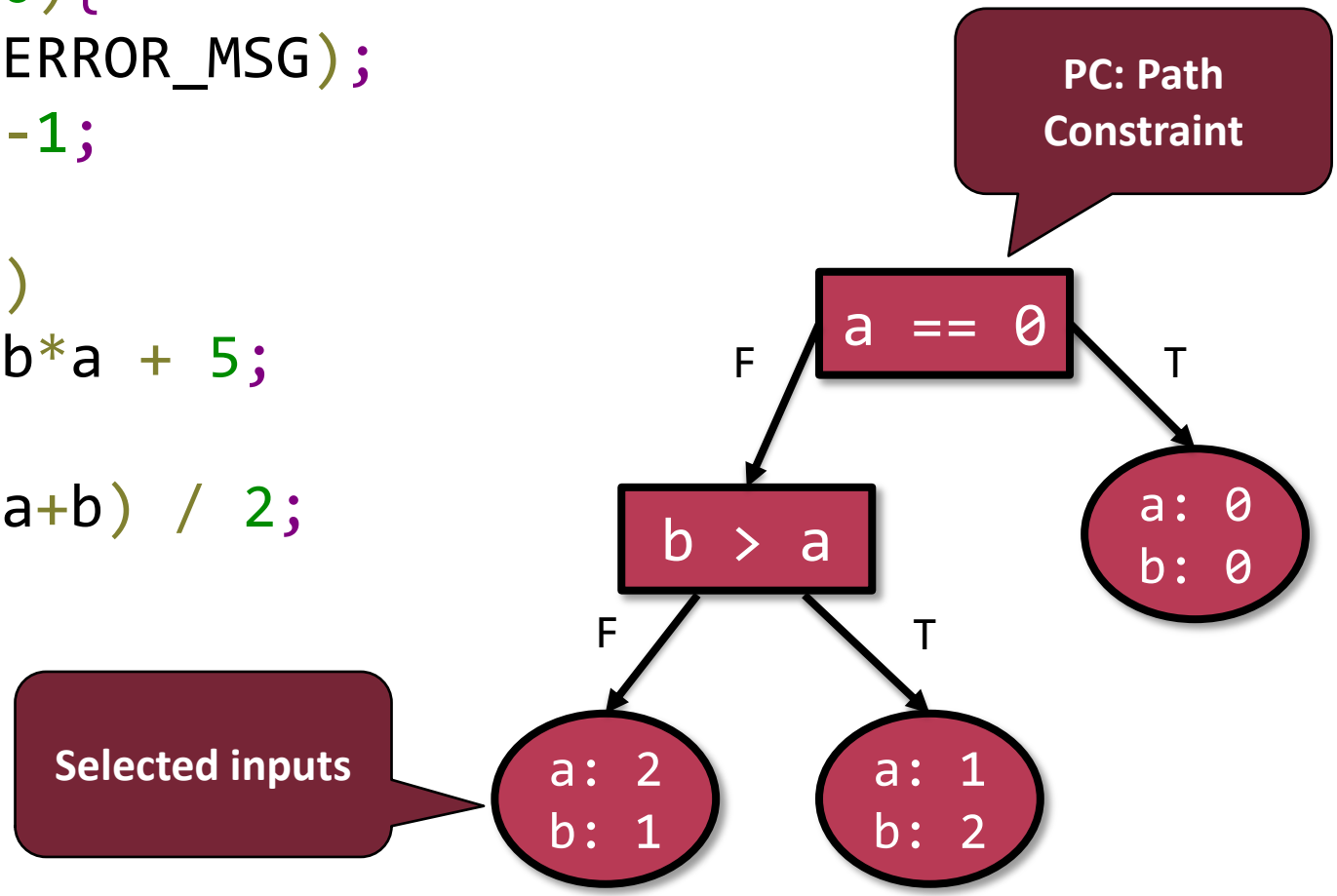
- „Whole test suite generation”
  - All test goals are taken into account
  - Searches based on multiple metrics
    - E.g., high coverage with minimal test suite
- Specialties:
  - Minimizes test code, maintains readability
  - Uses sandbox for environment interaction

TECHNIQUES

Symbolic execution

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

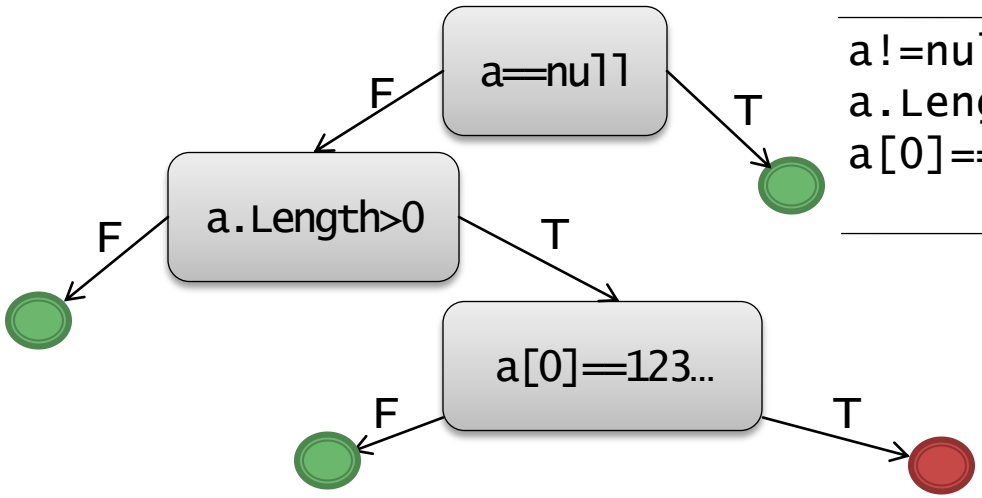
- **Static SE fails** in several cases, e.g.
  - Too long paths → too many constraints
  - Cannot decide if a path is really feasible or not

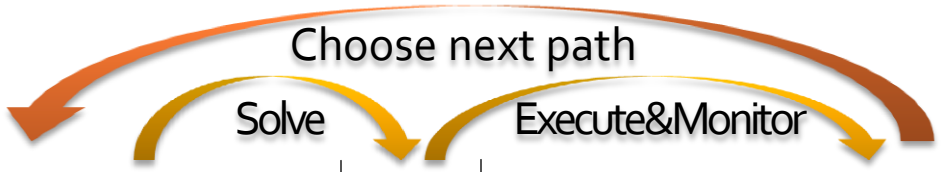
- **Idea:** mix symbolic with concrete executions
  - Dynamic Symbolic Execution (DSE) or
  - Concolic Testing

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");
}
```





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	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

DEMO: Microsoft IntelliTest

Generate unit tests for your code with IntelliTest
https://msdn.microsoft.com/en-us/library/Dn823749.aspx

```
1 reference
public int SwitchBranching(int condition)
{
    var divider = 0;
    switch(condition)
    {
        case 0:
            return 0;
        case 1:
            return -1;
        case 2:
            return -2;
        default:
            return (condition / divider);
    };
}
```

100 %

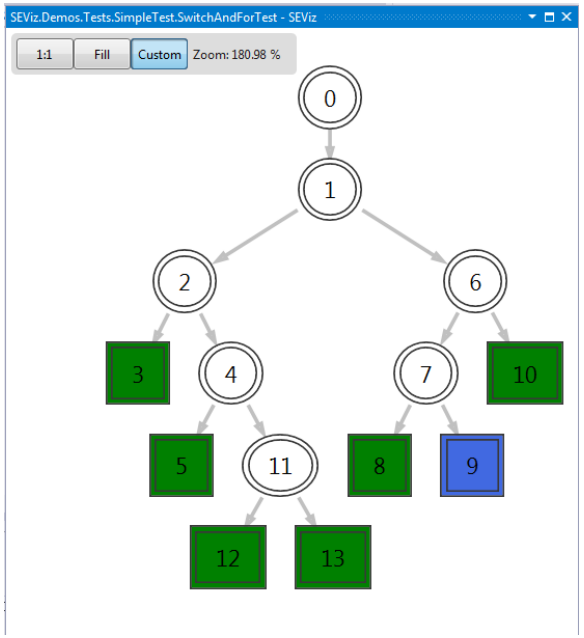
IntelliTest Exploration Results - stopped

SimpleTest.SwitchBranchingTest(S Run 0 Warnings

3 1 5/5 blocks, 0/0 asserts, 4 runs

	target	condition	result(target)	result	Summary / Exception	Error Message
1	new Simple{}	0	new Simple{}	0		
2	new Simple{}	1	new Simple{}	-1		
3	new Simple{}	2	new Simple{}	-2		
4	new Simple{}	3			DivideByZeroException	Attempted to divide by zero.

Error List Output IntelliTest Exploration Results



SEViz (Symbolic Execution VisualIZer)
https://github.com/FTSRG/seviz

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);
    int oldQuantity = StorageManager.GetQuantityFor(prod);
    // Calling the UUT
    int newQuantity = StorageManager.ReduceQuantity(prod,soldCount);
    // Assertions
    Assert.IsTrue(newQuantity >= 0);
    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);
}
```

Challenges of SE-based techniques

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

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

- Ideas: most SMT solvers cannot handle these
  - E.g., CORAL is specially designed for these problems
  - Using different solvers for different cases

Challenges (4)

Compound structures and objects

- Structures, recursive data structures

- Idea: **Lazy initialization**
  - Fields remain uninitialized at start
  - Assigning values only when they are used
    - Values: e.g., null, reference to a new or existing object, etc.

Challenges (6)

Interaction with the environment

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

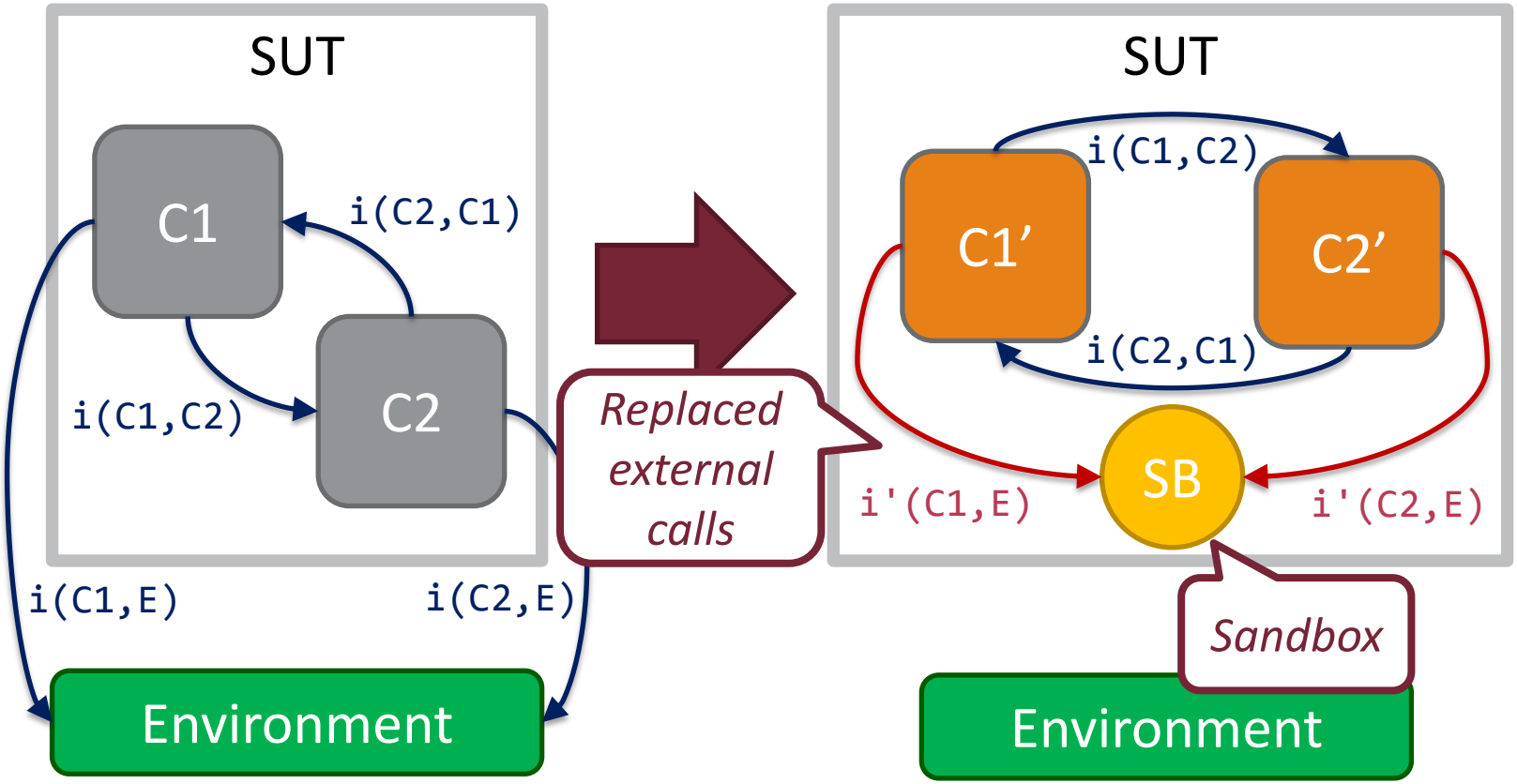
- Calls to platform and external libraries
- Idea:
  - „Environment models” (KLEE): for simple C programs
  - Special Security Manager object (Java)

Existing solutions for environment handling

- Stubbing and mocking (*faking*)
  - Fixed values and checks for all DSE executions
  - Not suitable for test generation
- **Parameterized mocking**
  - **Interaction with DSE is possible**
    - More relevant test cases
    - Custom behavior in mocks: e.g., state change of objects
  - **Introduces complexity for users of DSE**
    - Requires large amount of time and effort
    - Not trivial task in case of complex structures
  - **Fakes cannot be generated under certain conditions**

Our approach for automated isolation

- **Automated isolation on source code level**
  1. Abstract syntax tree transformations in the SUT
  2. Parameterized sandbox synthesization



Example of *AST transformation*

```
public class WeekendNotifier {
  public bool IsWeekendNear() {
    DateTime date = DateTime.GetNow();
    date.AddDays(2);
    if(date.GetDay() == "Saturday") return true;
    return false;
  }
}
```

```
public class WeekendNotifier {
  public bool IsWeekendNear() {
    DateTime date = Fake.DateTimeGetNow();
    Fake.DateTimeAddDays(2,date);
    if(Fake.DateTimeGetDay(date) == "Saturday") return true;
    return false;
  }
}
```

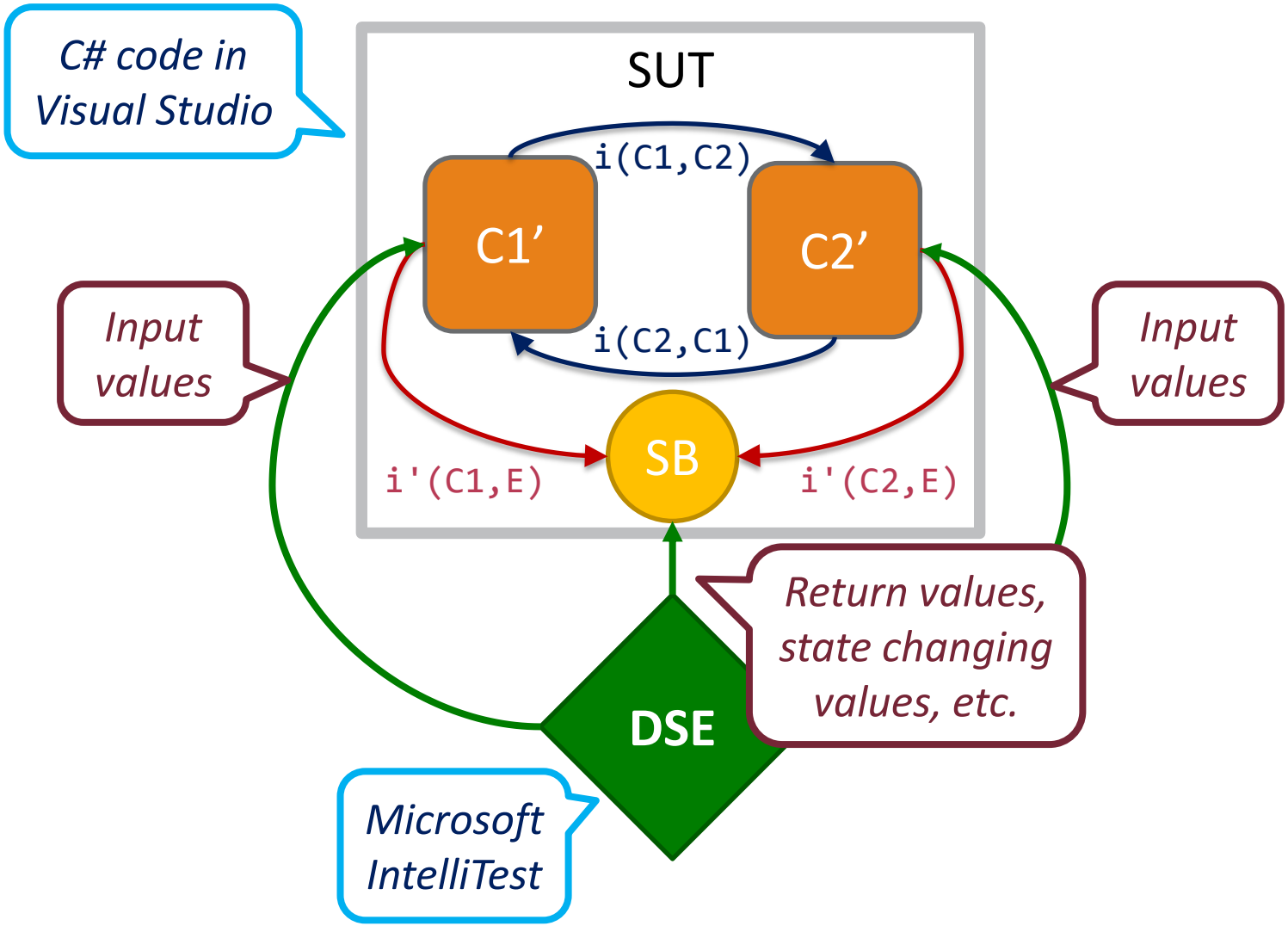
Example of *parameterized sandbox*

```
public static class Fake {

  public DateTime DateTimeGetNow() {
    // Return a state container object instead of the original
    return New<DateTime>.Instance();
  }

  public void DateTimeAddDays(int days, DateTime date) {
    // TODO: State change of date using the memory address
  }

  public int DateTimeGetDay(DateTime date) {
    // Obtaining return value from DSE
    return DSEEngine.ChooseValue<int>();
  }

}
```

A peek in the details

- **Input:** FQNs of classes under test
- **Transformations**
  - **AST traversal** for exploration and rewriting
    - Rewriting of invocations, member accesses: *mandatory*
    - Rewriting of object creations: *centralized state storage*
  - **In-memory partial compilation** for type information
    - Semantic model for AST nodes
- **Parameterized sandbox synthesization**
  - Based on AST transformations: *signature information*
  - Merged into one static class: **Fake**

Workflow of the prototype



Some open questions and ideas

- **How to use the results?**
  - Generated tests cannot be used directly on original
  - Using only test data
    - Feedback for user: in editor, report, etc.
    - Integration testing
  - Compositional dynamic symbolic execution
- **How to obtain basic behavior for the sandbox?**
  - User definition
  - Synthesized from environment models
- **How to restrict behavior to avoid false positives?**

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 – requirements were missing

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

L. Cseppentő, Z. Micskei: „Evaluating Symbolic Execution-based Test Tools," ICST'15