

Testing, Debugging, Program Verification

Automated Test Case Generation, Part II

Wolfgang Ahrendt & Vladimir Klebanov & Moa Johansson

12 December 2012

Specification-/Model-Based Test Case Generation

Systematic test case generation from JML contracts:

Black Box guided by several **Test Generation Principles**

- ▶ Test states make precondition true, consistency with class invariant
- ▶ Disjunctive analysis of each clause in precondition
- ▶ Choose representative values from larger sets
- ▶ Generation principle for datatypes of unbound locations

Specification-/Model-Based Test Case Generation

Systematic test case generation from JML contracts:

Black Box guided by several **Test Generation Principles**

- ▶ Test states make precondition true, consistency with class invariant
- ▶ Disjunctive analysis of each clause in precondition
- ▶ Choose representative values from larger sets
- ▶ Generation principle for datatypes of unbound locations

Remaining Problems of ATCG

1. How to **automate** specification-based test generation?
2. Generated test cases bear no relation to **implementation**

Recap: Ideas Behind ATCG

Ideas common to systematic (automated) test case generation

- ▶ **Formal** analysis of specification and/or code yields enough information to produce test cases
- ▶ Systematic algorithms give certain **coverage** guarantees
- ▶ Post conditions and invariants can be turned into test **oracles**
- ▶ **Mechanic reasoning** technologies achieve automation:
 - ▶ constraint solving
 - ▶ logic-based deduction
 - ▶ **symbolic execution**
 - ▶ model finding

Automated Test Generation Methods

Methods derived from **black box** testing

The implementation is unknown. Test data generated from spec, randomly, etc.

Generate test cases by analyzing **formal specification** or **formal model** of IUT (Implementation Under Test)

Methods derived from **white box** testing

The implementation is analyzed to generate test data.

Code-based test generation that uses **symbolic execution** of IUT.

Code-Based Test Generation

Generate test cases from **symbolic execution** of **code** of IUT

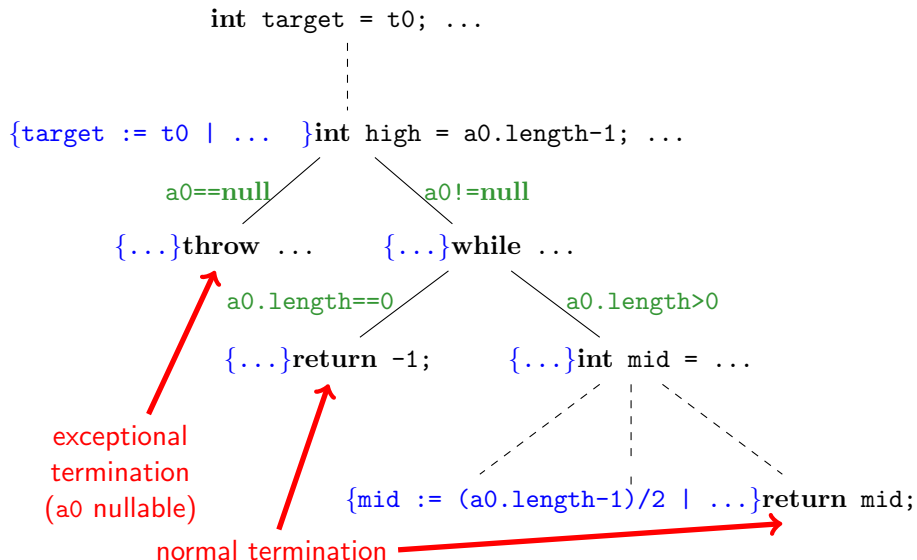
- ▶ **White/glass box** technology
- ▶ Current tools are research prototypes:
Symstra, Java PathFinder, Korat, PEX, SpecExplorer, Kiasan, KeY
- ▶ Very dynamic development, nearly industrial strength
- ▶ JAVA, bytecode, .NET, C (via abstraction)
- ▶ **No formal specification/system model required** (but can help)

Recap: What is Symbolic Execution?

Symbolic Execution

- ▶ State: a “symbolic” valuation of all variables (stack) and fields (heap)
 - ▶ Introduce new symbols (terms) to denote initial value of variables etc.
 - ▶ Each term represents a set of possible concrete values
- ▶ Execution tree: finite OR infinite tree of states
- ▶ States in the tree are annotated with next (sub-)statement to be executed (program counter)
- ▶ Branching state transitions are annotated with path conditions
- ▶ Each concrete execution path is an instance of some symbolic path through the tree
- ▶ Initial state given explicitly or by a symbolic precondition

Symbolic Execution Tree



Properties of Symbolic Execution

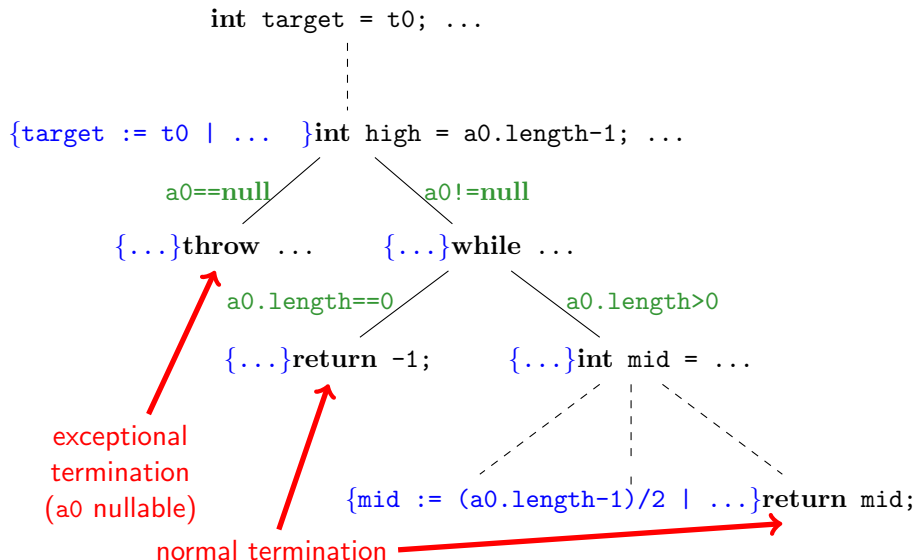
Important Conclusions

- ▶ One **symbolic** execution path corresponds to ∞ many test runs
- ▶ Programs with loops or recursion usually have ∞ many symbolic execution paths

Main Properties of Symbolic Execution

1. Even **symbolic** execution cannot cover **all** execution paths
2. But symbolic execution covers **all** execution paths up to **finite** depth

Symbolic Execution Tree: Binary Search



From Symbolic Execution to Test Cases

Code-Based Test Case Generation

1. Create symbolic execution tree for IUT **until finite depth**
2. For each **terminating** node (normal/exceptional) create test case:
 - 2.a Let PC be path condition of executed branch
 - 2.b Turn PC into quantifier-free first-order logic formula pc
 - 2.c Find a f.o. model M for pc that validates it (use theorem prover)
 - 2.d From M extract assignments for preamble of unit test case

From Symbolic Execution to Test Cases

Code-Based Test Case Generation

1. Create symbolic execution tree for IUT **until finite depth**
2. For each **terminating** node (normal/exceptional) create test case:
 - 2.a Let PC be path condition of executed branch
 - 2.b Turn PC into quantifier-free first-order logic formula pc
 - 2.c Find a f.o. model M for pc that validates it (use theorem prover)
 - 2.d From M extract assignments for preamble of unit test case

Example (Code-Based Test Case Generation)

1. See previous slide
2. Choose right-most terminating path
 - 2.a PC: `a0!=null && a0.length>0 && t0==a0[a0.length-1)/2]`

From Symbolic Execution to Test Cases

Code-Based Test Case Generation

1. Create symbolic execution tree for IUT **until finite depth**
2. For each **terminating** node (normal/exceptional) create test case:
 - 2.a Let PC be path condition of executed branch
 - 2.b Turn PC into quantifier-free first-order logic formula pc
 - 2.c Find a f.o. model M for pc that validates it (use theorem prover)
 - 2.d From M extract assignments for preamble of unit test case

Example (Code-Based Test Case Generation)

1. See previous slide
2. Choose right-most terminating path
 - 2.a PC: $a_0 \neq \text{null} \ \&\& \ a_0.\text{length} > 0 \ \&\& \ t_0 == a_0[a_0.\text{length}-1]/2]$
 - 2.b $pc \equiv \neg a_0 = \text{null} \wedge \text{length}(a_0) > 0 \wedge t_0 = a_0[(\text{length}(a_0) - 1) \div 2]$

From Symbolic Execution to Test Cases

Code-Based Test Case Generation

1. Create symbolic execution tree for IUT **until finite depth**
2. For each **terminating** node (normal/exceptional) create test case:
 - 2.a Let PC be path condition of executed branch
 - 2.b Turn PC into quantifier-free first-order logic formula pc
 - 2.c Find a f.o. model M for pc that validates it (use theorem prover)
 - 2.d From M extract assignments for preamble of unit test case

Example (Code-Based Test Case Generation)

1. See previous slide
2. Choose right-most terminating path
 - 2.a PC: $a_0 \neq \text{null} \ \&\& \ a_0.\text{length} > 0 \ \&\& \ t_0 == a_0[a_0.\text{length}-1]/2]$
 - 2.b $pc \equiv \neg a_0 = \text{null} \wedge \text{length}(a_0) > 0 \wedge t_0 = a_0[(\text{length}(a_0) - 1) \div 2]$
 - 2.c $M(\text{length}(a_0)) = 2, M(a_0) = \{17, 42\}, M(t_0) = M(a_0[0]) = 17$

From Symbolic Execution to Test Cases

Code-Based Test Case Generation

1. Create symbolic execution tree for IUT **until finite depth**
2. For each **terminating** node (normal/exceptional) create test case:
 - 2.a Let PC be path condition of executed branch
 - 2.b Turn PC into quantifier-free first-order logic formula pc
 - 2.c Find a f.o. model M for pc that validates it (use theorem prover)
 - 2.d From M extract assignments for preamble of unit test case

Example (Code-Based Test Case Generation)

1. See previous slide
2. Choose right-most terminating path
 - 2.a PC: $a_0 \neq \text{null} \ \&\& \ a_0.\text{length} > 0 \ \&\& \ t_0 == a_0[a_0.\text{length}-1]/2]$
 - 2.b $pc \equiv \neg a_0 = \text{null} \wedge \text{length}(a_0) > 0 \wedge t_0 = a_0[(\text{length}(a_0) - 1) \div 2]$
 - 2.c $M(\text{length}(a_0)) = 2, M(a_0) = \{17, 42\}, M(t_0) = M(a_0[0]) = 17$
 - 2.d `int target = 17; int[] array = {17, 42};`

Coverage, Code-Based

Code-based coverage criteria guaranteed by the resulting test suites depend on nodes/edges/paths covered in symbolic execution tree

All of finitely many feasible symbolic execution paths

Feasible Path Coverage — Rare to have (small) finite # of paths!

Each control-dependency in code occurs on some symbolic path

Feasible Branch Coverage — Achieved by unwinding loops often enough

Each reachable statement occurs on some symbolic path

Reachable Statement Coverage — Achieved by unwinding each loop once

Preconditions: Pruning Infeasible Execution Paths

Example (Binary search with precondition (requires clause))

```
/*@ public normal_behavior
  @ requires array != null && ... ;
  @*/
int search(/*@ nullable @*/ int array[], int target) {...}
```

```
    int target = t0; ...
    |
    |
{array := a0 | ... }int high = a0.length-1; ...
    |               |
    |               | a0!=null
a0==null /          \
{...}throw ...      {...}while ...
                    |
                    |
```

Preconditions: Pruning Infeasible Execution Paths

Example (Binary search with precondition (requires clause))

```
/*@ public normal_behavior
   @ requires array != null && ... ;
   @*/
int search(/*@ nullable @*/ int array[], int target) {...}
```

int target = t0; ...

|

{array := a0 | ... } int high = a0.length-1; ...

a0==null

a0!=null

{...} throw ...

{...} while ...



execution branch
contradicts precondition

|

Postconditions: Synthesizing Test Oracle Code

Oracle Problem in Automated Testing

How to determine automatically whether a test run succeeded?

The “ensures” clause of a JML contract tells exactly that provided that “requires” clause is true for given test case

Guarded JML quantifiers as executable Java code

JML:

```
\forall int i; guard(i) ==> test(i)
```

Equivalent executable JAVA code:

```
for (int i = lowerBound; guard(i); i++) {  
    if (!test(i)) { return false; }  
} return true;
```

Combining Specification- and Code-Based ATCG

(Specification-Based) Test Generation Principle 1 (Relevance)

State at start of IUT execution must make required precondition true

(Specification-Based) Test Generation Principle 8 (Oracle)

Use “ensures” clauses of contracts (and class invariant) as test oracles

Combining Specification- and Code-Based ATCG

(Specification-Based) Test Generation Principle 1 (Relevance)

State at start of IUT execution must make required precondition true

(Specification-Based) Test Generation Principle 8 (Oracle)

Use “ensures” clauses of contracts (and class invariant) as test oracles

(Specification-Based) Test Generation Principle 3 (Clause Coverage)

For each disjunct D of precondition in DNF create a test case whose initial state makes D true and as many other disjuncts as possible false

(Code-Based) Test Generation Principle (Statement Coverage)

Create test case for each terminating node in symbolic execution tree —
Unwind each loop at least once

Hybrid Coverage

(Hybrid) Test Generation Principle

Create true/false test cases for each disjunct of precondition in DNF

AND

Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill logic- and code-based coverage criteria

Hybrid Coverage

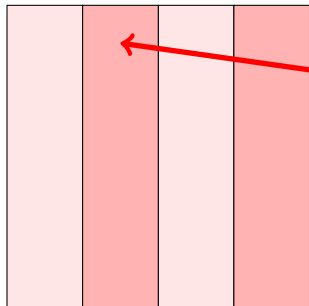
(Hybrid) Test Generation Principle

Create true/false test cases for each disjunct of precondition in DNF

AND

Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill logic- and code-based coverage criteria



Disjunctive analysis of precondition

Hybrid Coverage

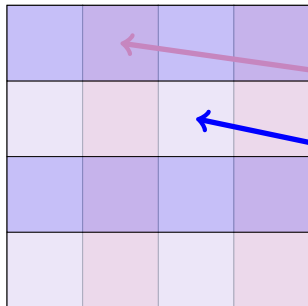
(Hybrid) Test Generation Principle

Create true/false test cases for each disjunct of precondition in DNF

AND

Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill logic- and code-based coverage criteria



Disjunctive analysis of precondition

Code-based analysis: path conditions

Hybrid Coverage

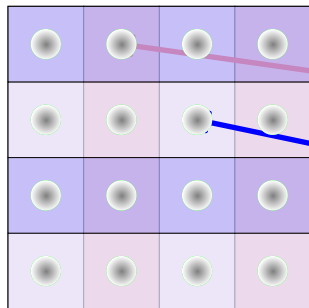
(Hybrid) Test Generation Principle

Create true/false test cases for each disjunct of precondition in DNF

AND

Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill logic- and code-based coverage criteria

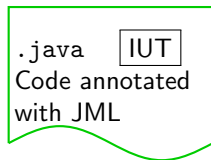


Disjunctive analysis of precondition

Code-based analysis: path conditions

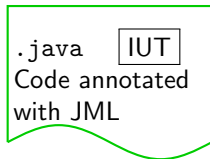
Choosing class representatives

Hybrid Test Case Generation: Overview



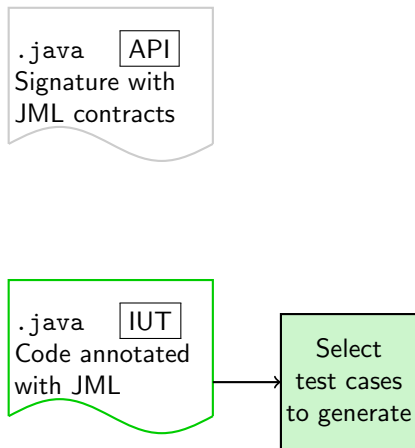
User input

Hybrid Test Case Generation: Overview



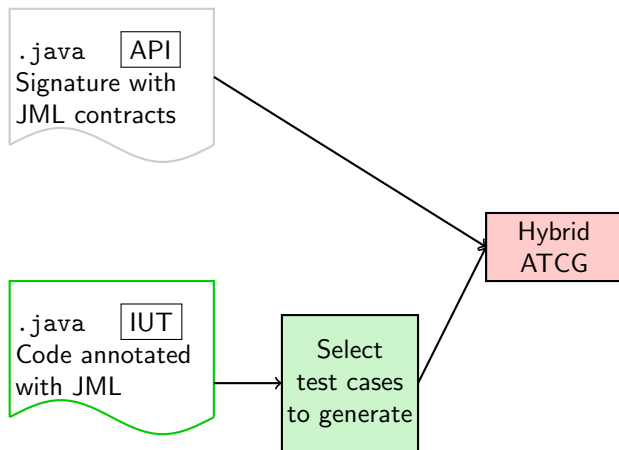
User input — Library

Hybrid Test Case Generation: Overview



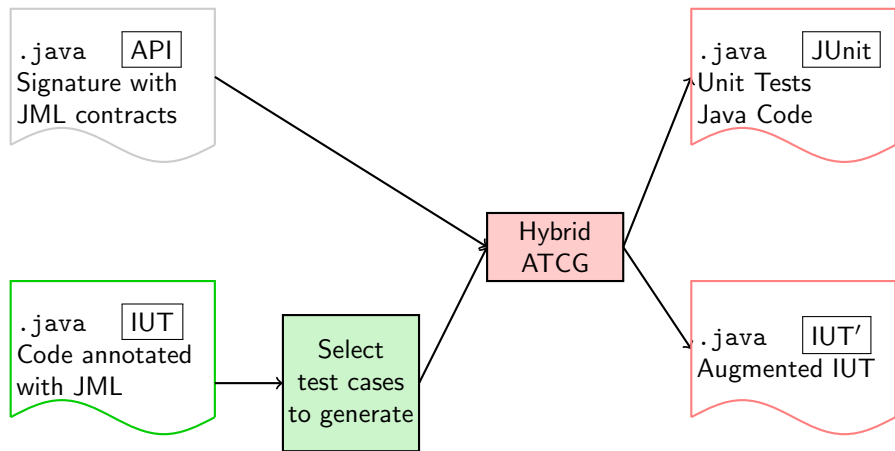
User input — Library

Hybrid Test Case Generation: Overview



User input — Library

Hybrid Test Case Generation: Overview



User input — Library — Automatically Generated

Summary

- ▶ Black box vs White box testing
- ▶ Black box testing ~ Specification-/Model-based Test Generation
- ▶ White box testing ~ Code-based Test Generation
- ▶ Systematic test case generation from JAVA code guided by Symbolic Execution
- ▶ Symbolic Execution:
Path Condition + Symbolic State + Program Counter
- ▶ Test cases are models of path conditions in terminating paths
- ▶ Coverage criteria, feasible branch coverage
- ▶ Postconditions of contract and invariants provide test oracle
- ▶ Combine Specification-based and Code-based Test Generation