Program Verification

Automated Test Case Generation, Part II

Reiner Hähnle

30 November 2007

Vous Êtes Ici

Specification-Based Test Case Generation

- Systematic test case generation from JML contracts: Black Box guided by Test Generation Principles
- Make precondition true, consistent with class invariant
- Disjunctive analysis
- Choose representative values from large equivalence classes
- Generation principles for datatypes of unbound variables

Vous Êtes Ici

Specification-Based Test Case Generation

- Systematic test case generation from JML contracts: Black Box guided by Test Generation Principles
- Make precondition true, consistent with class invariant
- Disjunctive analysis
- Choose representative values from large equivalence classes
- Generation principles for datatypes of unbound variables

Remaining Problems of ATCG

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

Vous Êtes Ici

Specification-Based Test Case Generation

- Systematic test case generation from JML contracts: Black Box guided by Test Generation Principles
- Make precondition true, consistent with class invariant
- Disjunctive analysis
- Choose representative values from large equivalence classes
- Generation principles for datatypes of unbound variables

Remaining Problems of ATCG

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

1. Tools `jml-junit` and `jtest` discussed in Exercises
2. Code-based test generation that uses symbolic execution of IUT

Recap

Ideas common to systematic (automated) test generation

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

Code-Based Test Generation

Generate test cases from symbolic execution of code of IUT

- White box technology
- All available tools are academic and more or less experimental: Symstra, Java PathFinder, Korat, PEX, SpecExplorer, Kiasan, KeY
- Very dynamic development, industrial strength in 2–3 years
- Mostly JAVA, but also bytecode
- No formal specification/system model required

What is Symbolic Execution?

Execute a program with symbolic (abstract) initial values

Assume we could write a Java program such as this:

```
int target  = t0;
int[] array = a0;
return search(array, target);
```

where t_0 and a_0 are arbitrary start values.

Can view t_0 and a_0 as first-order terms whose value is fixed by a model

Symbolic Execution by Example

```
int target  = t0;      ⟵ Execute this statement
int[] array = a0;
int low = 0;
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0} ← Symbolic Program State

```
int[] array = a0;   ← First Active Statement (Program Counter)
int low = 0;
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0 | array := a_0}

```
int low = 0;
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0 | array := a_0 | low := 0}

```
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

$\{$`target` := t_0 | `array` := a_0 | `low` := $0\}$

```
int high = a0.length-1;
```

Execution depends on a_0 `!=null`

```
while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

a_0!=null ← Path Condition

{target := t_0 | array := a_0 | low := 0 | high := a_0.length-1}

```
while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

a_0`!=null`

{`target` := t_0 | `array` := a_0 | `low` := 0 | `high` := a_0`.length-1`}

```
while ( low <= high ) {        ← depends on  a0.length>0
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0
{target := t0 | array := a0 | low := 0 | high := a0.length-1}

int mid = (low + high) / 2 ;
if ( target < array[ mid ] ) {
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

a_0!=null && a_0.length > 0
{target := t_0 | array := a_0 | low := 0 | high := a_0.length-1 |
mid := (a_0.length-1)/2}

```
if ( target < array[ mid ] ) {
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0
{target := t0 | array := a0 | low := 0 | high := a0.length-1 |
 mid := (a0.length-1)/2}

if ( t0 < a0[ (a0.length-1)/2 ] ) { No exception thrown!
    high = mid - 1;
} else if ( target > array[ mid ] ) {
    low = mid + 1;
} else {
    return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0
{target := t0 | array := a0 | low := 0 | high := a0.length-1 |
 mid := (a0.length-1)/2}

if ( t0 < a0[ (a0.length-1)/2 ] ) { let t0==a0[(a0.length-1)/2]
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

a_0`!=null && `a_0`.length > 0 && `t_0`==`a_0`[ `a_0`.length-1)/2 ]`
`{target := `t_0` | array := `a_0` | low := 0 | high := `a_0`.length-1 |`
` mid := (`a_0`.length-1)/2}`

```
if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

a_0`!=null && `a_0`.length > 0 && `t_0`==`a_0`[ `a_0`.length-1)/2 ]`
`{target := `t_0` | array := `a_0` | low := 0 | high := `a_0`.length-1 |`
` mid := (`a_0`.length-1)/2}`

```
if ( t0 > a0[ (a0.length-1)/2 ] ) {
    low = mid + 1;
} else {
    return mid;
}
while ( low <= high ) {
  ...
}
return -1;
```

false!

Symbolic Execution by Example

a_0!=null && a_0.length > 0 && t_0==a_0[a_0.length-1)/2]
{target := t_0 | array := a_0 | low := 0 | high := a_0.length-1 |
 mid := (a_0.length-1)/2}

```
return mid;
while ( low <= high ) {
  ...
}
return -1;
```

Symbolic Execution by Example

a_0`!=null && `a_0`.length > 0 && `t_0`==`a_0`[ `a_0`.length-1)/2 ]`
`{target := `t_0` | array := `a_0` | low := 0 | high := `a_0`.length-1 |`
` mid := (`a_0`.length-1)/2}`

return `(`a_0`.length-1)/2;`

Result of Symbolic Execution

Conclusion to be drawn from symbolic execution:

All execution paths for test cases (states) that validate path condition:

`array!=null && array.length>0 && target==array[array.length-1)/2]`

return the result

`(array.length-1)/2`

Result of Symbolic Execution

Conclusion to be drawn from symbolic execution:

All execution paths for test cases (states) that validate path condition:

`array!=null && array.length>0 && target==array[array.length-1)/2]`

return the result

`(array.length-1)/2`

Important Properties

- One symbolic execution path corresponds to ∞ many test runs
- Only one symbolic execution path shown in example
  need to explore others as well!
- Programs with loops or recursion usually have ∞ many symbolic execution paths

Result of Symbolic Execution

Conclusion to be drawn from symbolic execution:

All execution paths for test cases (states) that validate path condition:

`array!=null && array.length>0 && target==array[array.length-1)/2]`

return the result

`(array.length-1)/2`

Main Property of Symbolic Execution

Even symbolic execution cannot cover all execution paths

But symbolic execution covers all execution paths to finite depth

Elements of Symbolic Execution

Components of a State during Symbolic Execution

Path condition — when is this execution path taken?

Symbolic program state — like Variables compartment in Debugger

Program counter — next executable source code statement

Program state and Program counter also present in Debuggers

Elements of Symbolic Execution

Components of a State during Symbolic Execution

Path condition — when is this execution path taken?

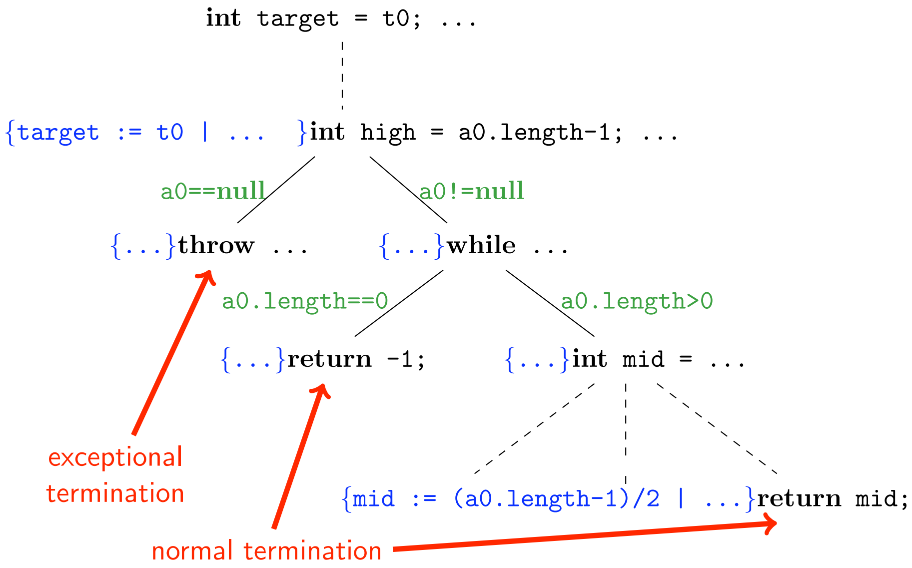
Symbolic program state — like Variables compartment in Debugger

Program counter — next executable source code statement

Program state and Program counter also present in Debuggers

State of Symbolic Execution $\Rightarrow$ node in Symbolic Execution Tree

Symbolic Execution Tree



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 execution branch
   - **2.b** Turn PC into quantifier-free first-order logic formula pc
   - **2.c** Find a model M for pc that validates it
   - **2.d** From M extract concrete values of variables for 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 execution branch
   - **2.b** Turn *PC* into quantifier-free first-order logic formula *pc*
   - **2.c** Find a model *M* for *pc* that validates it
   - **2.d** From *M* extract concrete values of variables for 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 execution branch
   - 2.b Turn *PC* into quantifier-free first-order logic formula *pc*
   - 2.c Find a model *M* for *pc* that validates it
   - 2.d From *M* extract concrete values of variables for 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]`
   - 2.b $pc \equiv \neg a_0 = \mathrm{null} \wedge \mathrm{length}(a_0) > 0 \wedge t_0 = a_0[\mathrm{length}(a_0) - 1] \div 2$

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 execution branch
   - 2.b Turn *PC* into quantifier-free first-order logic formula *pc*
   - 2.c Find a model *M* for *pc* that validates it
   - 2.d From *M* extract concrete values of variables for 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]`
   - 2.b $pc \equiv \neg a_0 = \mathrm{null} \wedge \mathrm{length}(a_0) > 0 \wedge t_0 = a_0[\mathrm{length}(a_0) - 1] \div 2$
   - 2.c $M(\mathrm{length}(a_0)) = 2$, $M(a_0) = \{17, 42\}$, $M(t_0) = M(a_0[0]) = 17$

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 execution branch
   - 2.b Turn *PC* into quantifier-free first-order logic formula *pc*
   - 2.c Find a model *M* for *pc* that validates it
   - 2.d From *M* extract concrete values of variables for 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]`
   - 2.b $pc \equiv \neg a_0 = \mathrm{null} \wedge \mathrm{length}(a_0) > 0 \wedge t_0 = a_0[\mathrm{length}(a_0) - 1] \div 2$
   - 2.c $M(\mathrm{length}(a_0)) = 2$, $M(a_0) = \{17, 42\}$, $M(t_0) = M(a_0[0]) = 17$
   - 2.d `int target = 17; int[] array = {17,42};`

Coverage

Coverage criteria guaranteed by the resulting test suites
depend on which nodes/edges contained in symbolic execution tree

Coverage

Coverage criteria guaranteed by the resulting test suites
depend on which nodes/edges contained in symbolic execution tree

All of finitely many symbolic execution paths

Feasible Path Coverage — Rare to have only finitely many paths!

Coverage

Coverage criteria guaranteed by the resulting test suites
depend on which nodes/edges contained in symbolic execution tree

All of finitely many symbolic execution paths

Feasible Path Coverage — Rare to have only finitely many paths!

As above, but methods approximated by JML contracts

Top Level Feasible Path Coverage

Coverage

Coverage criteria guaranteed by the resulting test suites
depend on which nodes/edges contained in symbolic execution tree

All of finitely many symbolic execution paths

Feasible Path Coverage — Rare to have only finitely many paths!

As above, but methods approximated by JML contracts

Top Level Feasible Path Coverage

Each control-dependency in code occurs on some symbolic path

Feasible Branch Coverage — Achieved by unwinding loops *often enough*

Coverage

Coverage criteria guaranteed by the resulting test suites
depend on which nodes/edges contained in symbolic execution tree

All of finitely many symbolic execution paths

Feasible Path Coverage — Rare to have only finitely many paths!

As above, but methods approximated by JML contracts

Top Level Feasible Path Coverage

Each control-dependency in code occurs on some symbolic path

Feasible Branch Coverage — Achieved by unwinding loops *often enough*

Each statement occurs on some execution path

Feasible 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( 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( 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

Test data must make required precondition true

(Specification-Based) Test Generation Principle 8

Use "ensures" clauses (postconditions) of JML contracts as test oracles

Combining Specification- and Code-Based ATCG

(Specification-Based) Test Generation Principle 1

Test data must make required precondition true

(Specification-Based) Test Generation Principle 8

Use “ensures” clauses (postconditions) of JML contracts as test oracles

(Specification-Based) Test Generation Principle 3

For each disjunct of precondition in DNF create test case making it true

(Code-Based) Test Generation Principle

Create test case for each terminating node in symbolic execution tree

Combined Coverage

(Combined) Test Generation Principle

Create test case for each disjunct of precondition in DNF
AND
Create test case for each terminating node in symbolic execution tree

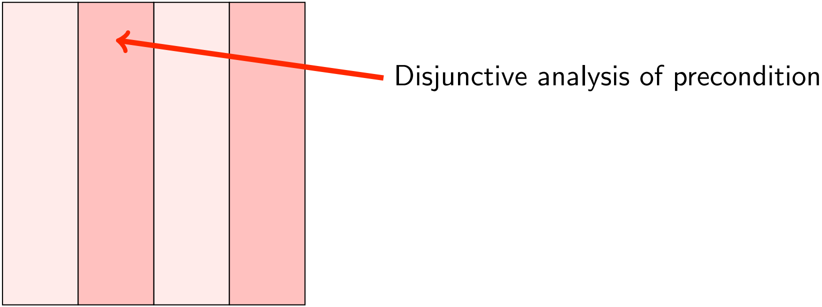
Resulting test cases fulfill both coverage criteria

Combined Coverage

(Combined) Test Generation Principle

Create test case for each disjunct of precondition in DNF
AND
Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill both coverage criteria

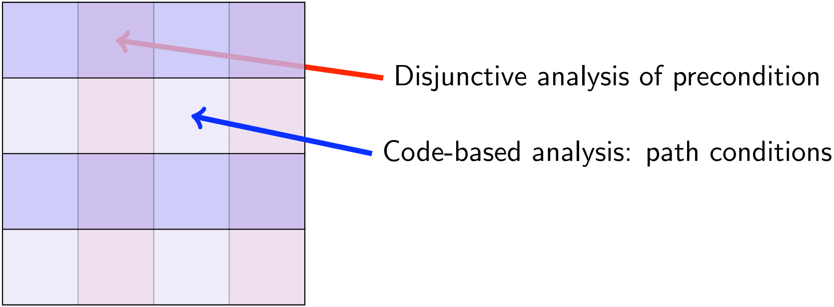


Combined Coverage

(Combined) Test Generation Principle

Create test case for each disjunct of precondition in DNF
AND
Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill both coverage criteria

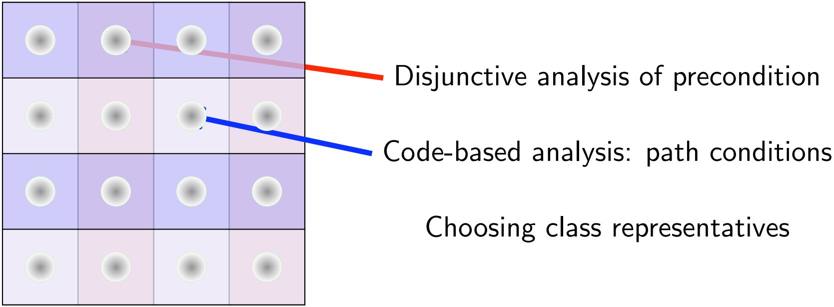


Combined Coverage

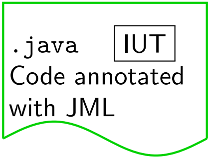
(Combined) Test Generation Principle

Create test case for each disjunct of precondition in DNF
AND
Create test case for each terminating node in symbolic execution tree

Resulting test cases fulfill both coverage criteria

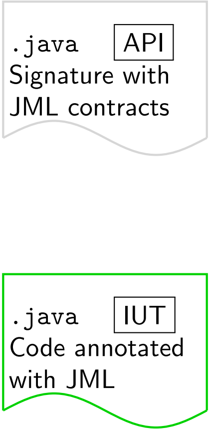


Combined Test Case Generation: Overview



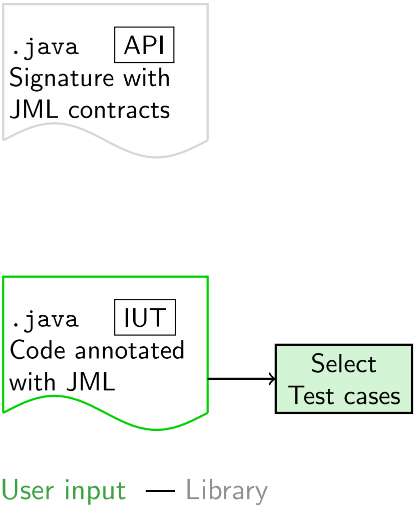
User input

Combined Test Case Generation: Overview

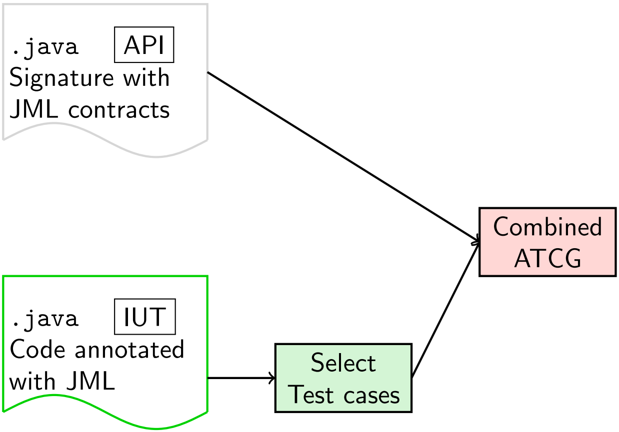


User input — Library

Combined Test Case Generation: Overview

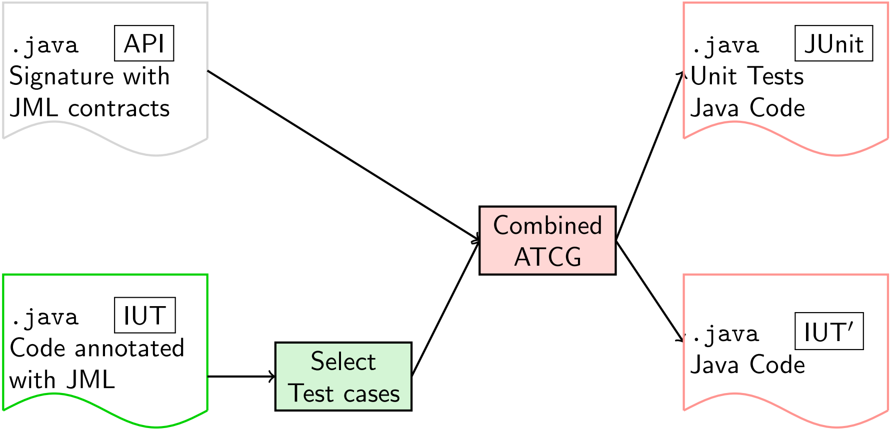


Combined Test Case Generation: Overview



User input — Library

Combined Test Case Generation: Overview



User input — Library — Automatically Generated

Demo: Test Generation

Stand-alone test generation tool KeY Unit Test Generator

Demo: `javaws`

- `export CLASSPATH=/usr/share/java/junit.jar:.`
- `javaws http://www.key-project.org/download/testing/KeYTest.jnlp`
- Load `Examples/NatNumWrap/NaturalNumberWrapper.java`
- Explain class
- Generate tests
- Run created JUnit test cases
- Inspect generated test cases to see failure-inducing test case

Inspect the failed test case file to see initial values

Demo: Test Generation

Stand-alone test generation tool KeY Unit Test Generator

Demo: `javaws`

- `export CLASSPATH=/usr/share/java/junit.jar:.`
- `javaws http://www.key-project.org/download/testing/KeYTest.jnlp`
- Load `Examples/NatNumWrap/NaturalNumberWrapper.java`
- Explain class
- Generate tests
- Run created JUnit test cases
- Inspect generated test cases to see failure-inducing test case

Inspect the failed test case file to see initial values

The bug is found even though it is not covered in the spec!

Summary

- Black box vs White box testing
- Black box testing $\sim$ Specification-based Test Generation
- White box testing $\sim$ 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 conditionsin terminating paths
- Coverage criteria, feasible branch coverage
- Postconditions of contract provide test oracle
- Combine Specification-based and Code-based Test Generation

What Next?

Central Remaining Problem

- When does a program have no more bugs?
  How to prove correctness without executing ∞ many paths?

What Next?

Central Remaining Problem

- When does a program have no more bugs?
  How to prove correctness without executing ∞ many paths?

Final Topic of Course

- Formally Verifying Program Correctness