Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 7

Today’s Topic:
Automated Test Case Generation

How to Test Effectively?

```
public class Factorial {
    /*@ requires n >= 0;
      @ ensures \result > 0;
      @*/
    public static int factorial (int n) {
        int result = n;
        while (--n > 0) result *= n;
        return result;
    }

    public static void main (String[] param) {
        int n = Integer.parseInt(param[0]);
        int fact_n = factorial(n);
        System.out.println("n: " + n + ", n!: " + fact_n);
    }
}
```

Writing a main method for each test case does not scale.

How to Test Effectively?

Faulty implementation of enqueue on binary heap:

```
public void enqueue(Comparable o) {
  if (numElems >= elems.length) grow();
  int pos = numElems++;
  int parent = pos / 2;
  while (pos > 0 && elems[parent].compareTo(o) > 0) {
    elems[pos] = elems[parent];
    pos = parent;
    parent = pos / 2;
  }
  elems[pos] = o;
}
```

Writing all test cases manually does not scale.

Automated Testing

- Unit Testing: write code to automatically test your code.
- A unit test is a test suite for a unit (class/module) of a program and consists of
  - setup code to initialize the tested class; (test fixture/preamble)
  - tear down code to clean up after testing;
  - test cases that call methods of the tested class with appropriate inputs
  - check the result of each call (test oracle)
- Once test suites are written, they are easy to run repeatedly (regression testing).

Unit Testing in Java: JUnit

- A popular framework for unit testing in Java
  - Frameworks are libraries with *gaps*
  - Programmer writes classes following particular conventions to fill in the gaps
  - Result is the complete product
- JUnit automates
  - the execution and analysis of unit tests;
  - generation of tests cases from parameterized test oracles and user-provided test data.

JUnit Example

```
import static org.junit.Assert.*;
import org.junit.*;
...
public class PriorityQueueTest {
  private PriorityQueue pq;

  @Before public void setUp () { pq = new Heap(); }
  @After public void tearDown () { pa = null; }

  @Test public void enqueueTest () {
    Integer value = new Integer(5);
    pq.enqueue(value);
    assertEquals(pq.removeFirst, value);
  }
  ...
}
```

Drawbacks of JUnit

- Low degree of automation
  - Programmer still needs to write all the test cases
- Redundant specification
  - Duplication between checks in test oracles and formal specification
    (e.g. provided as JML annotations)

Automated Test Generation

- Black box testing
  - Implementation is unknown
  - Test data generated from spec (e.g., randomly)
  - Does not require source code
  - Can generate insufficient/irrelevant test data
- White box testing
  - Implementation is analyzed to generate test data for it
  - Requires source or byte code
  - Can use full information from code

Automated Test Generation Methods

- Methods derived from black box testing
  - Generate test cases from analyzing formal specification or formal model of implementation under test (IUT)
- Methods derived from white box testing
  - Code-based test generation that uses symbolic execution of IUT

We will focus on black box testing

Specification-Based Test Generation

- Generate test cases from analyzing formal specification or formal model of implementation under test (IUT)
  - Black box technology with according pros and cons
  - Many tools, commercial as well as academic: JMLUnit, JMLUnitNG, BZ-TT, JML-TT, UniTesK, JTest, TestEra, Korat, Cow Suite, UTJML, . . .
  - Various specification languages: B, Z, Statecharts, JML, ...
  - Detailed formal specification/system model required (here: JML)

Specification-Based Test Generation

- We use design-by-contract and JML as formal specification methodology:
  - View JML method contract as formal description of all anticipated runs

Specification-Based Test Generation

- Approach: Look at one method and its JML contract at a time (unit testing)
  1. Specialize JML contract to representative selection of concrete runs
     - concentrate on precondition (requires clause)
     - assumes that precondition species all anticipated input
     - analysis of implicit and explicit logical disjunctions in precondition
     - choose representative value for each atomic disjunct
  2. Turn these representative program runs into executable test cases
  3. Synthesize test oracle from postcondition of contract

Contracts and Test Cases

```
/*@ public normal_behavior
@ requires Pre;
@ ensures Post;
@*/
public void m() { ... }
```

- All prerequisites for intended behavior contained in requires clause
- Unless doing robustness testing, consider behavior violating preconditions irrelevant
- State at start of IUT execution must make precondition true

Test Case Generation: Example

```
public class Traffic {
  private /*@ spec_public @*/ boolean red, green, yellow;
  private /*@ spec_public @*/ boolean drive, brake, halt;
  /*@ public normal_behavior
    @ requires red || yellow || green;
    @ ensures \old(red) ==> halt &&
    @         \old(yellow) ==> brake;
    @*/
  public boolean setAction() {
    // implementation
  }
}
```

Which test cases should be generated?

Data-Driven Test Case Generation

- Generate a test case for each possible value of each input variable
  - Combinatorial explosion
    (already 2^6 cases for our simple example)
  - Infinitely many test cases for unbounded data structures
  - Some resulting test cases unrelated to specification or IUT
- Restriction to test cases that satisfy precondition?
- Insufficient (still too many), but gives the right clue!

Coverage Criteria for Specification-Based Testing

Example

```
requires red || yellow || green;
```

is true even for `red=yellow=green=true`

How many different test cases to generate?

Create test cases that make parts of precondition true:

- At least one test per spec case (Decision Coverage)
- One for each disjunct in precondition (Disjunctive Coverage)
- All disjunctive combinations (Multiple Condition Coverage)
- Criteria based on making predicates true/false, etc.

Disjunctive Coverage

```
/*@ public normal_behavior
  @ requires red || yellow || green;
  @ ensures \old(red) ==> halt &&
  @         \old(yellow) ==> brake;
  @*/
```

Disjunctive analysis of precondition suggests minimum of three test cases that relate to precondition.

Disjunctive Coverage

- **Definition (Disjunctive Normal Form (DNF))**
  A requires clause of a JML contract is in DNF when it has the form
  $D_1 \;||\; D_2 \;||\; \ldots \;||\; D_n$
  where each D_i does not contain an explicit or implicit disjunction.
- **Disjunctive 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

Disjunctive Coverage

Example:

`@ requires red || yellow || green;`

gives rise to three test cases

- `red=true; yellow=green=false`
- `yellow=true; red=green=false`
- `green=true; red=yellow=false`

Importance of Establishing DNF Syntactically

- Implicit logical disjunctions must be made explicit by computing DNF: e.g. replace `A ==> B` with `!A || B`, etc.

Dealing with Existential Quantification

Example (Square root)

```
/*@ public normal_behavior
  @ requires n>=0 && (\exists int r; r >= 0 && r*r
== n);
  @ ensures ... @*/
public static final int sqrt(int n) { ... }
```

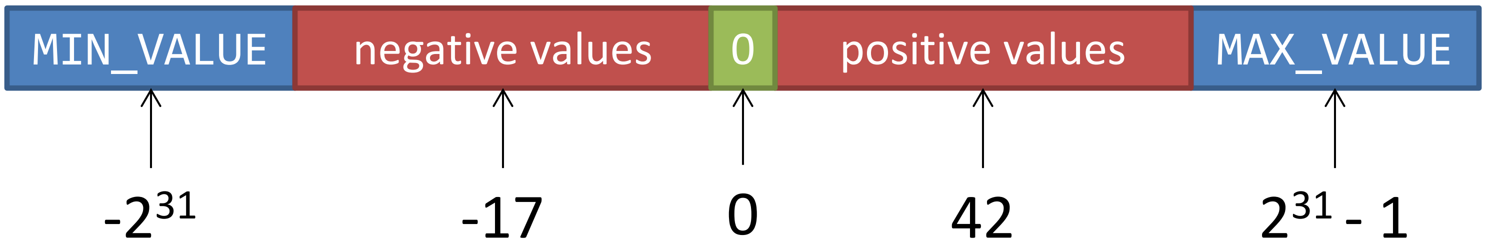
Where is the disjunction in the precondition?

Existential quantifier as disjunction:

- Existentially quantified expression (\exists int r; P(r))
- Rewrite as: P(MIN_VALUE)|| ... || P(0)|| ... || P(MAX_VALUE)
- Get rid of those P(i) that are false: P(0)|| ... || P(46340)
- Still too many cases. . .

Partitioning of Large Input Domains

- Partition large/infinite domains in finitely many equivalence classes



- Partitioning tries to achieve that the same computation path is taken for all input values within a potential equivalence class.
- Then, one value from each class is sufficient to check for defects.
- As we don't know the IUT, correct partitioning is in general unattainable.
- Judicious selection and good heuristics can make it work in practice.

Boundary Values

Example (Square)

```
/*@ public normal_behavior
  @ requires n>=0 && n*n >= 0;
  @ ensures \result >=0 && \result == n*n;
  @*/
public static final int square(int n) { ... }
```

Include boundary values of ordered domains as class representatives.

Which are suitable boundary values for `n` in this example?

Implicit Disjunctions, Part I

Example (Binary search, target not found)

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < array.length;
  @                          array[i-1] <= array[i]);
  @ (\forall int i; 0 <= i && i < array.length;
  @                 array[i] != target);
  @ ensures \result == -1;
  @*/
int search( int array[], int target ) { ... }
```

No disjunction in precondition!?

We can freely choose `array`, `length`, and `target` in precondition!

Free Variables

- Free variables:
  - Values of variables without explicit quantification can be freely chosen
  - Amounts to implicit existential quantification over possible values
- How choose representatives from types of free variables?
  - There are infinitely many different arrays . . .
  - Before defining equivalence classes, need to enumerate all values

Data Generation for Free Variables

Systematic enumeration of values by data generation principle

Assume declaration: `int[] ar;`, then the array `ar` is

1. either the `null` array: `int[] ar = null;`
2. or the empty `int` array: `int[] ar = new int[0];`
3. or an `int` array with one element
   a. `int[] ar = { MIN_VALUE };`
   b. `int[] ar = { MIN_VALUE + 1 };`
   c. ...
4. or an `int` array with two elements . . .
5. . . .

Combining the Heuristics

Example (Binary search, target found)

```
requires (\exists int i; 0 <= i && i < array.length
                         && array[i] == target) &&
         (\forall int i; 0 < i && i < array.length;
                         array[i-1] <= array[i]);
```

Apply test generation principles:

1. Use data generation for unbound `int` array
2. Choose equivalence classes and representatives for:
   - `array: int[]` empty, singleton, two elements (usually, need to stop here)
   - `target: int` (include boundaries)
3. Generate test cases that make precondition true

Combining the Heuristics

Example (Binary search, target found)

```
requires (\exists int i; 0 <= i && i < array.length
                         && array[i] == target) &&
         (\forall int i; 0 < i && i < array.length;
                         array[i-1] <= array[i]);
```

- empty array: precondition cannot be made true, no test case
- singleton array, target must be the only array element
  ```
  array = { 0 }; target = 0;
  array = { 1 }; target = 1;
  ```
- two-element sorted array, target occurs in array
  ```
  array = { 0, 0 }; target = 0;
  array = { 0, 1 }; target = 0;
  array = { 1, 1 }; target = 1;
  ```

Implicit Disjunctions, Part II

Example (List Copy)

```
/*@ public normal_behavior
  @ requires true; // src, dst non-nullable by default
  @ ensures ...
  @*/
static void java.util.Collections.copy(List src, List dst)
```

Aliasing and Exceptions

- In Java object references `src`, `dst` can be aliased, i.e., `src==dst`
  - Aliasing usually unintended - exclusion often forgotten in contract
- Preconditions can be (unintentionally) too weak
  - Exception thrown when `src.length > dst.length`

Generate test cases that enforce/prevent aliasing and throwing exceptions (when not excluded by contract).

The Postcondition as Test Oracle

- Oracle Problem in Automated Testing
  - How to determine automatically whether a test run succeeded?
  - The `ensures` clause of a JML contract provides verdict on success provided that `requires` clause is true for given test case
  - Use `ensures` clauses of contracts (and class invariant) as test oracles

Executable JML Expressions

- How to determine whether a JML expression is true in a program state?
- It is expensive to check whether a JML expression is true in a state
  - Corresponds to first-order model checking, because JML ~ FOL
  - PSPACE-complete problem, efficient solutions exist only for special cases
  - Identify a syntactic fragment of JML that can be mapped into Java

Executable JML Expressions

Example

`\exists int i; 0 <= i && i < ar.length && ar[i] == target`

is of the form

`\exists int i; guard(i) && test(i)`

where

- `guard()` is Java expression with fixed upper/lower bound
- `test()` is executable Java expression

Guarded existential JML quantifiers as Java (Example)

```
for ( int i = 0; 0 <= i && i < ar.length; i++) {
  if (ar[i] == target ) { return true; }
} return false;
```

Tools for JML-based Test Case Generation

JMLUnit: Unit Testing for JML

JMLUnit is a unit testing framework for JML built on top of JUnit

User:

- writes specifications
- supplies test data of each type

JMLUnit automatically:

- constructs test cases from test data
- assembles test cases into test suites
- executes test suites
- decides success or failure
- reports results

Test Cases and Suites

- A test case (o,$\boldsymbol{x}$) consists of:
  - a non-null receiver object o
  - a sequence $\boldsymbol{x}$ of argument objects

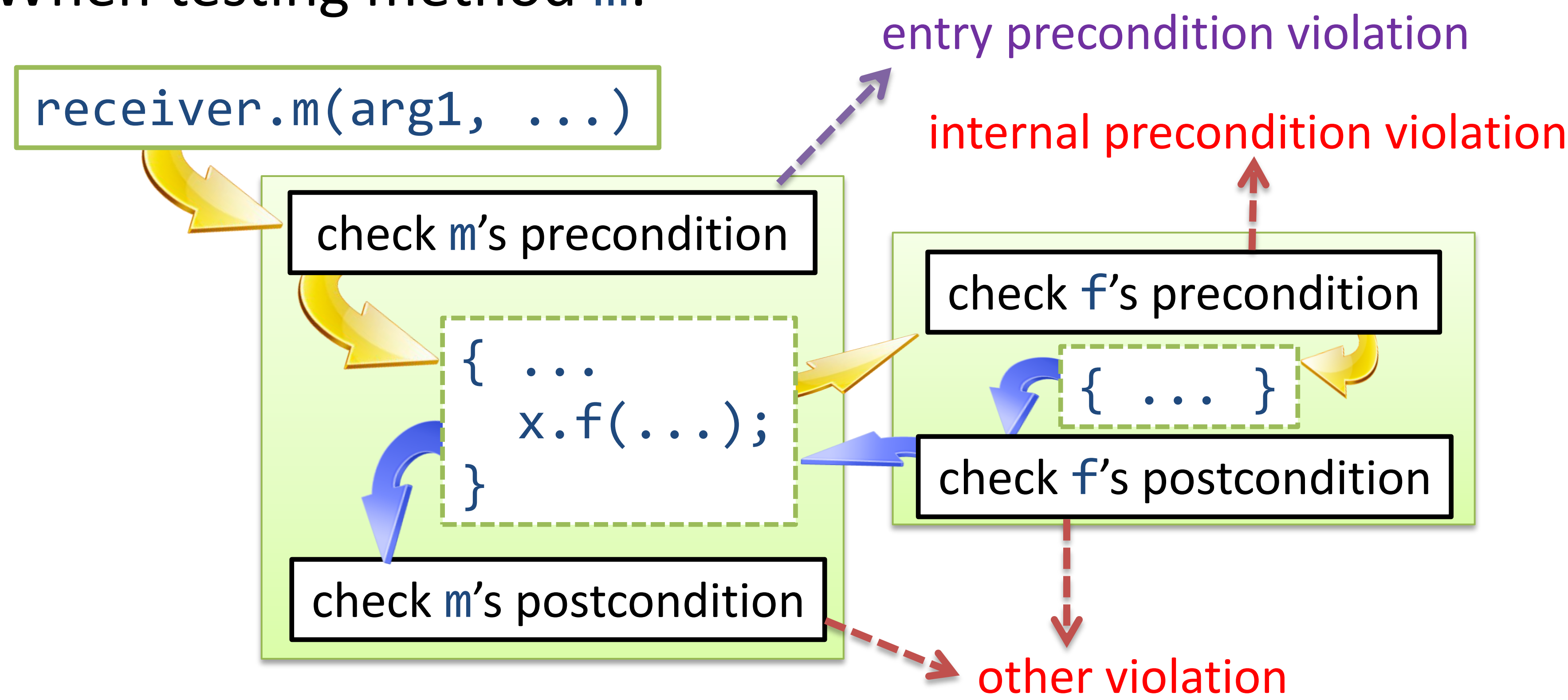
- A test suite for method `m` is a set of test cases with:
  - receiver of `m`’s receiver type
  - arguments of `m`’s argument types

Test Suites are Cross Products

- For method enqueue:
  $\{ (pq, v) \mid pq \in PriorityQueueTestData, v \in IntegerTestData \}$
- Default is to use all data for all methods
  - Filtered automatically by preconditions
  - Users can filter manually if desired
- Factory method allows user control of adding test cases to test suite.

Errors and Meaningless Test Cases

When testing method `m`:



Entry precondition violation ⇒ test case rejected

Internal or other violation ⇒ error reported

Supplying Test Data

- Programmer supplies data in form of strategies
- A strategy for type T:
  - has method that returns iterator yielding T
- Strategies allow reuse of test data
- JMLUnit provides a framework of built-in strategies
  - Strategies for built-in types
  - Allow for easy extension, composition, filtering, etc.

Strategies for Test Data

- Standard strategies:
  - Immutable: iterate over array of values;
  - Cloneable: iterate over array, clone each;
  - Other: create objects each time.
- Cloning and creating from scratch can prevent unwanted interference between tests.
- JMLUnit tries to guess appropriate strategy.

Example Strategies

```
import org.jmlspecs.jmlunit.strategies.*;
import junit.framework.*;

public abstract class Heap_JML_TestData extends TestCase {
  public IntIterator vCompIter(String methodName, int argNum)
  { return vComparableStrategy.ComparableIterator(); }
  private StrategyType vComparableStrategy =
    new ImmutableObjectAbstractStrategy() {
      protected Object[] addData() {
        return new Integer[] {10, -22, 55, 3000};
      }
    };
  ...
```

Example Strategies

```
...
public IndefiniteIterator vHeapIter (String methodName, int argNum)
{ return vPointStrategy.iterator(); }

private StrategyType vHeapStrategy =
  new NewObjectAbstractStrategy() {
    protected Object make(int n) {
      switch (n) {
        case 0: return new Heap();
        case 1: return new Heap(new Integer {1, 2, 3});
        default: break;
      }
      throw new NoSuchElementException();
    }
  };
}
```

Using JMLUnit

- JML-compile the class to be tested
  jmlc Factorial.java
- generate the test suite and test data templates
  jmlunit Factorial.java
- supply the test data
  $EDITOR Factorial_JML_TestData.java
- compile the test suite
  javac Factorial_JML_Test*.java
- execute the test suite
  jmlrac Factorial_JML_Test

Drawbacks of JMLUnit

- Limited degree of automation:
  - only test data for primitive types is generated automatically
- Limited degree of granularity:
  - fine-grained filtering of test data for individual methods is difficult
- Limited coverage:
  - no guarantee that a certain coverage criterion is satisfied
- Limited relevancy of generated test cases
  - black box testing

Some Alternatives to JMLUnit

- JMLUnitNG
  - similar feature set as JMLUnit, better memory footprint, improved filtering of test data, ...
- Korat, TestEra, UDITA
  - automated generation of test data for complex data types (use techniques similar to Alloy)
- KeY Unit Test Generator, Java Pathfinder
  - based on symbolic execution + constraint solving (white box testing)

Automated Test Case Generation with Korat

- Provides test case generation for complex data types.
- Supports checking of JML specifications.
- User provides for each complex data type
  - a Java predicate capturing the representation invariant of the data type;
  - a finitization of the data type.
- Korat generates test cases for all instances that satisfy both the finitization constraints and the representation predicate (similar to Alloy)

Example: Binary Trees

```
import java.util.*;
class BinaryTree {
  private Node root;
  private int size;
  static class Node {
    private Node left;
    private Node right;
  }
  ...
}
```

Representation Predicate for BinaryTree

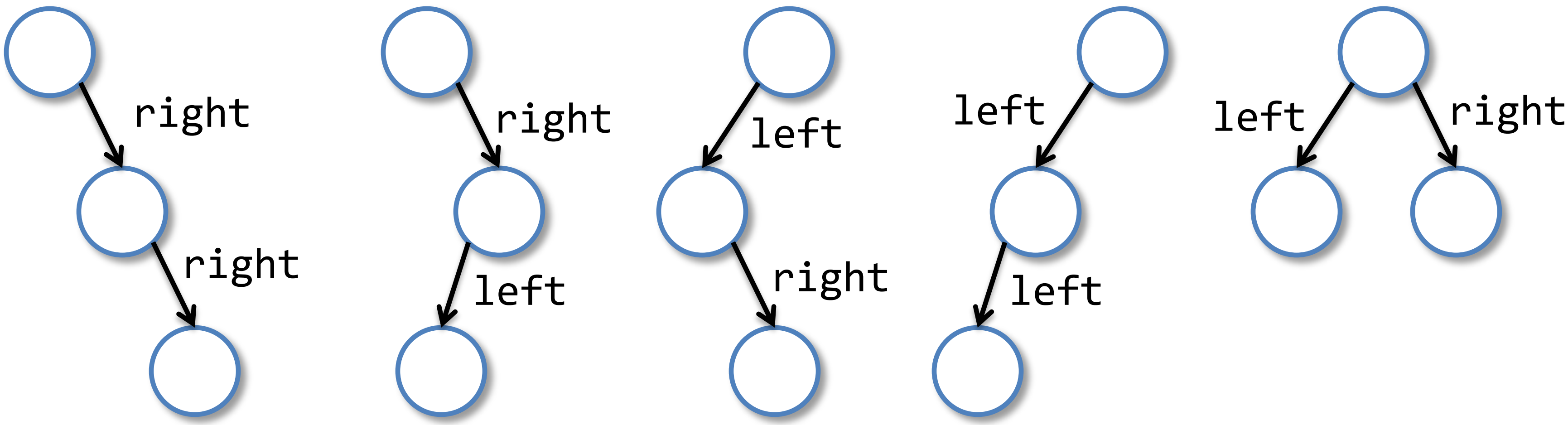
```
public boolean repOK() {
  if (root == null) return size == 0;
  Set visited = new HashSet();
  visited.add(root);
  LinkedList workList = new LinkedList();
  workList.add(root);
  while (!workList.isEmpty()) {
    Node current = (Node) workList.removeFirst();
    if (current.left != null) {
      if (!visited.add(current.left)) return false;
      worklist.add(current.left);
    }
    if (current.right!= null) { ... }
  }
  return visited.size () == size;
}
```

Finitization for BinaryTree

```
public static Finitization finBinaryTree (int NUM_Node) {
  IFinitization f = new Finitization(BinaryTree.class);
  IObjSet nodes = f.createObjSet(Node.class, NUM_Node, true);
                                // #Node = NUM_Node
  f.set(“root”, nodes);         // root in null + Node
  IIntSet sizes = f.createIntSet(Num_Node);
  f.set(“size”, sizes);         // size = NUM_Node
  f.set(“Node.left”, nodes);    // Node.left in null + Node
  f.set(“Node.right”, nodes);   // Node.right in null + Node
  return f;
}
```

Finitization for BinaryTree

Instances generated for finBinaryTree(3)



Summary

- Black box vs. white box testing
- Black box testing ~ specification-based test generation
- Systematic test case generation from JML contracts guided by a few heuristics
  - Only generate test cases that make precondition true
  - Each operation contract and each disjunction in precondition gives rise to a separate test case
  - Choose appropriate coverage criterion, e.g., disjunctive coverage
  - Large/infinite datatypes approximated by class representatives
  - Values of free variables supplied by data generation
  - Create separate test cases for potential aliases and exceptions
- Postconditions of contract and class invariants provide test oracle
- Turn pre- and postconditions into executable Java code