3-2007

# Random Test Data Generation for Java Classes Annotated with JML Specifications

Yoonsik Cheon
*The University of Texas at El Paso*, ycheon@utep.edu

Carlos E. Rubio-Medrano



Comments:
Technical Report UTEP-CS-07-11

# Random Test Data Generation for Java Classes Annotated with JML Specifications

Yoonsik Cheon and Carlos E. Rubio-Medrano

TR #07-11
March 2007

**Keywords:** random testing, test data generator, runtime assertion checking, pre and postconditions, JML language.

**1998 CR Categories:** D.2.5 [*Software Engineering*] Testing and Debugging — testing tools (e.g., data generators, coverage testing); D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

To appear in *International Conference on Software Engineering Research and Practice, Las Vegas, NV, June 25–28, 2007.*

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

# Random Test Data Generation for Java Classes Annotated with JML Specifications

Yoonsik Cheon and Carlos E. Rubio-Medrano
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968-0518
{ycheon, cerubio}@utep.edu

*Abstract*— **The hidden states of objects create a barrier to designing and generating test data automatically. For example, the state of an object has to be established indirectly through a sequence of method invocations. For a non-trivial class, however, it is extremely difficult for a randomly-chosen sequence of method invocations to construct an object successfully, as each invocation has to satisfy the state invariants. Nonetheless, automated random testing can reduce the cost of testing dramatically and has strong potential for finding errors that are difficult to find in other ways because it eliminates the subjectiveness in constructing test data. We propose a new approach to generating test data automatically for Java classes annotated with JML specifications. The key idea underlying our approach is to construct an object incrementally in that each method call in the sequence is checked before the next call is chosen. We use JML's runtime assertion checker to check the validity of a method invocation. Other ingredients of our approach include object pooling and object equivalence checking. These are to increase the probability of constructing feasible call sequences and to remove redundancy among the successfully-built sequences. We have implemented our approach for JET, a fully-automated testing tool for Java, and our experiment with JET showed a promising result, 10 to 200% increase in the number of generated test cases.**

**Keywords:** random testing, test data generator, runtime assertion checking, pre and postconditions, JML language.

## I. Introduction

Testing is laborious, time consuming, error-prone, and costly. Automation can be a solution to this problem, and the key component of test automation is generating test data automatically. One way to generate test data automatically is to pick an arbitrary input value from the set of all possible input values of the program under test. This random approach can reduce the cost of testing dramatically and has potential for finding faults that are difficult to find in other ways because it eliminates the subjectiveness in constructing test cases and increases the variety of input values.

However, the hidden states of objects in object-oriented programming languages such as Java pose a problem in selecting test data randomly. It is difficult to construct an object of an interesting state, as one can't assign values directly to the hidden state variables. One has to establish the state of an object indirectly through a sequence of method invocations. But, when methods are selected randomly, it is uncertain whether a method call sequence produce an object of a consistent state [1]. Each method invocation in the sequence has to bring the object to a (new) consistent state. The consistency of an object is often specified formally as class invariants in an interface specification language such as JML [2]. In addition, the method invocation has to satisfy the specification of the invoked method, i.e., method pre and postconditions. Even if the sequence produces a consistent object, the object may be redundant in that an object with an equivalent state may already exist in the test suite. According to a recent study, more than 50% of randomly generated test data are redundant [3].

In this paper, we propose a new approach to generating test data randomly for Java classes annotated with JML specifications. In our approach, we construct an object incrementally by ensuring the validity of each randomly-selected method call while constructing the call sequence. For this, we use JML's runtime assertion checker. Each method call should produce no runtime assertion violation errors; otherwise, a new method is selected randomly. To facilitate the random selection of methods, we classify methods into constructors and mutators, based on their signatures and JML specifications. A call sequence consists of a constructor call followed by some number of (mutator) method calls. We also introduce a notion of object equivalence, defined in terms of call sequences and the `equals` method, both to remove redundant test data and to facilitate reuse of successfully-built objects in constructing new objects.

We extended JET [1], an automated unit testing tool for Java, to implement our proposed approach. JET fully automates unit testing of Java classes annotated with JML specifications, from test data generation to test execution and test result determination. As in [4], it uses JML specifications as test oracles to decide test results automatically. The existing JET used a pure random approach to generate test data; it decides the whole call sequence at once without checking the validity of each call in the sequence. We extended JET by adding our approach as a new test data generation strategy (see Section IV). We experimented with the extended JET to evaluate the effectiveness of our proposed approach, and the results were quite promising. For all five classes that we experimented with, our incremental approach outperformed the pure random approach in terms of the number of valid test cases generated. Depending on the characteristics of sample classes, we observed 10 to 200% increase in the number of

generated test cases (refer to Section V for detailed results).

In the remainder of this section we introduce JML briefly with a small illustrative example to be used throughout this paper. In Section II, we explain the hidden state problem of objects in test data generation. In Section III and IV, we first explain our approach in detail and then briefly describe our extension to JET; we explain our algorithm of generating test data incrementally, including topics such as classification of methods, checking feasibility of method calls, object pooling, object equivalence, and redundancy of test data. In Section V, we present our experimental results on evaluating the effectiveness of our approach. In Section VI, we mention a few related work, and we conclude this paper with a concluding remark in Section VII.

### A. JML

The Java Modeling Language (JML) is an interface specification language for Java to formally specify the behavior of Java classes and interfaces [2], [5]. In JML, the behavior of a Java class is specified by writing class invariants and pre and postconditions for the methods exported by the class. The pre and postconditions are viewed as a contract between the client and the implementor of the class. The client must call a method in a state where the method's precondition holds, and the implementor must guarantee that the method's postcondition holds after such a call. The assertions in class invariants and pre and postconditions are usually written in a form that can be compiled, so that violations of the contract can be detected at runtime.

Fig. 1 shows an example JML specification taken from [1]. JML assertions are written as special annotation comments in Java source code, either after `//@` or between `/*@` and `@*/`. The keyword **`spec_public`** in line 2 states that the private field `bal` is treated as public for specification purpose; e.g., it can be used in the specifications of public methods. As shown in lines 5–7, a method (or constructor) specification precedes the declaration of the method (or constructor). The **`requires`** clause specifies the precondition, the **`assignable`** clause specifies the frame condition, and the **`ensures`** clause specifies the postcondition. The keyword **`\old`** in line 14 denotes the pre-state value of its expression; it is most commonly used in the specification of a mutation method such as `transfer` that changes the state of an object. Refer to [1] for a complete specification the `Account` class.

```
1  public class Account {
2    private /*@ spec_public @*/ int bal;
3    //@ public invariant bal >= 0;
4
5    /*@ requires amt >= 0;
6      @ assignable bal;
7      @ ensures bal == amt; @*/
8    public Account(int amt) {
9      bal = amt;
10   }
11
12   /*@ requires amt > 0 && amt <= acc.bal;
13     @ assignable bal, acc.bal;
14     @ ensures bal == \old(bal) + amt
15     @   && acc.bal == \old(acc.bal - amt); @*/
16   public void transfer(int amt, Account acc) {
17     acc.withdraw(amt);
18     deposit(amt);
19   }
20
21   // The rest of the definition including:
22   // Account(Account), deposit(int),
23   // withdraw(int), and int balance().
24 }
```

Fig. 1. Example JML specification

## II. The Problem

It is well known that the hidden state of an object creates difficulties for the programmer of a class in terms of testing [6]. For example, if one wants to check the state of an object before and after an invocation of the method under test, one needs to access the internal state of the object. However, it is hidden to the programmer. The hidden state is also a barrier to designing and creating test data, regardless of whether it is done manually or automatically. It is difficult to construct an object of an interesting state, as values can't be directly assigned to the hidden state variables. The state of an object has to be established indirectly through a sequence of method invocations.

However, there are several problems associated with this indirect approach, especially when the methods to be invoked are selected randomly as in most automated testing tools for Java programs, such as JCrasher [7] and Jtest [8]. It is uncertain whether a particular method call sequence will bring an object to an interesting state for testing. In addition, two different method call sequences may produce objects of the same or equivalent states, thus making one to be redundant. In fact, a recent study has shown that around 50% to 90% of randomly generated test data are redundant [3]. Redundant test data increases the testing time without increasing the ability to detect faults. However, the real problem is that it is even uncertain if a randomly generated sequence of method calls is *feasible*. A sequence of calls is feasible if each call of the sequence maintains the consistency of an object's state in that the state variables of the object satisfy the class invariants. In other words, each method invocation has to satisfy the method's specification (i.e., pre and postconditions) and the class invariants. For a non-trivial class, it is extremely unlikely that all randomly-selected methods in the sequence satisfy their specifications. The problem becomes aggravated by the fact that each method call in the sequence may require to generate a bunch of objects, i.e., the receiver and the arguments, each of which may also require to generate a bunch of other objects, i.e., the receiver and the arguments, and so on. That is, a test case for a method often consists of not just a single object but a large collection of objects. In our recent study [1], we found that up to 99% of randomly-generated call sequences became infeasible.

As an example, let's consider the `transfer` method of the `Account` class (see Fig. 1). The method requires two `Account` objects, one for the receiver and the other for the argument, and an integer value. Here are several test cases for the method.

$T_1$: $r$ = new Account(-10); $T_4$: $r$ = new Account(10);
$a$ = new Account(20); $r$.withdraw(20);
$r$.transfer(10, $a$); $a$ = new Account(20);
$T_2$: $r$ = new Account(10); $r$.transfer(10, $a$);
$a$ = new Account(20); $T_5$: $r$ = new Account(10);
$r$.transfer(10, $a$); $a$ = new Account(10);
$T_3$: $r$ = new Account(10); $a$.deposit(10);
$a$ = new Account(20); $r$.transfer(10, $a$);
$r$.transfer(-10, $a$);

The test cases $T_1$ and $T_4$ are infeasible because it is impossible to construct all the objects required by the test cases; in both cases, the receivers can't be constructed successfully. For $T_1$, the first constructor call doesn't satisfy the constructor's precondition requiring a non-negative initial balance. For $T_4$, the `withdraw` method call violates the method's precondition (or the class invariant) because the receiver ($r$) doesn't have enough balance left for the withdrawal transaction. The test case $T_3$ is feasible, but it is meaningless in that it is outside the domain of the `transfer` method; i.e., the requested transfer amount is negative. The test cases $T_2$ and $T_5$ are both feasible, but one is redundant because the elements of the two test cases are pairwise equivalent; i.e., both receivers have the same balance (20).

## III. Our Approach

The key components of our approach are an incremental construction of object states, object pooling, and checking equivalence of object states. We construct an object incrementally by assuring the feasibility of each method invocation whenever a new method is randomly selected to mutate the object. That is, instead of deciding the whole sequence of method calls at once, we build a sequence in such a way that its feasibility is guaranteed by selecting one feasible method at a time. The feasibility of a method invocation is checked by using the JML's runtime assertion checker. We also reuse objects by pooling successfully-built sequences; e.g., to create a new object, we may pick a pooled sequence and mutate the represented object by appending an additional sequence of method invocations at the end. The first two components of our approach increase the probability of constructing feasible sequences, and the last component helps to remove redundancy among constructed sequences.

### A. Notation

We assume each method of a class is tested separately. A test case for a method, $m$, of a class, $C$, consists of a receiver object and arguments. We denote a test case as a tuple of objects and values, $\langle r, a_1, \ldots, a_n \rangle$, where $r$ is the receiver and $a_i$'s are the arguments. The receiver is an object of the class $C$ and arguments can be either primitive values or objects. For a constructor and a static method, a test case consists of only arguments, $a_i$'s. We denotes an object as a sequence of constructor and method invocations, $\langle s_1; \ldots; s_n \rangle$, where each $s_i$ is of the form $m(a_1, \ldots, a_k)$, where $m$ is the method or constructor to invoke. For a method invocation, the receiver is implicit; it's the object being constructed and is given by the first element of the sequence, $s_1$. For example, the five test cases of Section II are represented as follows.

$T_1$: ⟨Account(-10), 10, Account(20)⟩
$T_2$: ⟨Account(10), 10, Account(20)⟩
$T_3$: ⟨Account(10), -10, Account(20)⟩
$T_4$: ⟨⟨Account(10); withdraw(20)⟩, 10, Account(20)⟩
$T_5$: ⟨Account(10), 10, ⟨Account(10); deposit(10)⟩⟩

As the arguments of a method or constructor can be objects, the sequence representing an object can have nested subsequences, e.g., ⟨Account(⟨Account(10); deposit(10)⟩); withdraw(10); transfer(20, ⟨new Account(30)⟩)⟩.

```
Object generate(Method m) {
  do {
    c = declaring_class(m);
    r = generate(c);
    foreach (p_i of m's parameter)
      a_i = generate(T_i);
  } while (is_redundant(⟨r, a_1, ..., a_n⟩));
  return ⟨r, a_1, ..., a_n⟩;
}
```

Fig. 2. Test data generation

### B. Detailed Approach

Fig. 2 shows a pseudo-code algorithm of generating test data for instance methods. It first generates a receiver object and then each argument of an appropriate type. For this, the algorithm uses a sub-algorithm *generate* that takes a type as an argument and returns an arbitrary value of that type. For a primitive type (e.g., `int`), the *generate* sub-algorithm selects an arbitrary value randomly. For an array type, it first chooses the dimension randomly and then generates the elements in accordance with the element type. For an interface and an abstract class, one has to specify at least one concrete implementation class or subclass; otherwise, null will be the only possible value. For a concrete class, it generates a random object by using the method described in Section III-C below. The generated test data is checked for redundancy (see Section III-D). If it is redundant, the whole process is repeated until non-redundant test data is found or a pre-defined maximum number of attempts is reached.

### C. Generating Objects

The key requirement of our algorithm is to generate an object of an arbitrary state. Remember that we represent an object as a sequence of method and constructor invocations. Thus, the algorithm's main task is to select a method or constructor to invoke, and to ensure that an invocation of the selected method or constructor is possible. Note that the selected constructor should create a new instance of a consistent state, and the selected method should mutate the object state to bring it to a new consistent state. For the successful construction of call sequences, we classify methods and constructors into different categories.

```
Object generate(Class T) {
  if (reuse?) {
    r = pick_from_pool();
  } else {
    do {
      c = pick_constructor(T);
      foreach (p_i of c's parameter)
        a_i = generate(T_i);
      r = invoke "new T(a_1, a_2, ..., a_n)";
    } while (invocation fails?);
  }
  r = mutate(r);
  add_to_pool(r);
  return r;
}

Object mutate(Object o) {
  for (several times) {
    do {
      m = pick_mutator(o);
      foreach (p_i of m's parameter)
        a_i = generate(T_i);
      invoke "o.m(a_1, ..., a_n)";
    } while (invocation fails?);
  }
  return o;
}
```

Fig. 3. Algorithm for generating objects of arbitrary states

Let $C = \langle C_b, C_e, M, O \rangle$ be a concrete class, where $C_b$, $C_e$, $M$, $O$ are the sets of basic constructors, extended constructors, mutators, and observers of $C$, respectively.[1] A *basic constructor* is a constructor or static method that can create an object of $C$ without needing another object of $C$. An *extended constructor* is a constructor or static method that can create an object of $C$ but needs one or more other objects of $C$ to do that. A *mutator* is an instance (non-static) method of $C$ that may mutate the state of an object. All other methods are *observers* and ignored by the algorithm. We will shortly explain below the criteria that we use to classify methods and constructors.

Fig. 3 shows our algorithm for generating objects. The algorithm consists of three steps:

*Step a*: *Create an instance.* This is done by either reusing a previously-built object or instantiating a fresh new object by selecting a constructor, $c \in C_b \cup C_e$, randomly and generating its arguments. If any of the arguments are of object types, the algorithm calls itself recursively to generate the argument objects. If it is decided to create a new instance, the selected constructor $c$ is invoked to ensure the feasibility of the call (see below for details of the feasibility check).

*Step b*: *Mutate the state.* The state of the created object is mutated by making a sequence of mutation method calls. For this, a mutator $m \in M$ is selected randomly and, as in the constructor call in the previous step, is actually invoked to check the feasibility of the call.

*Step c*: *Add to the object pool.* The constructed object is optionally added to the object pool so that it can be reused in building new objects as in the first step.

In the remainder of this section we briefly explain how we classify methods and constructors, how we ensure the feasibility of method or constructor invocations, and how we pool objects for future reuse.

*1) Method Classification:* As in our earlier work [1], we classify methods and constructors based on their signatures and, if available, specifications. We use the following criteria.

- *Basic constructor*. A constructor $c$ of $C$ is a basic constructor if it doesn't require an argument of type $C$. A static method $m$ of $C$ is a basic constructor if its return type is $C$ and it doesn't require an argument of type $C$.
- *Extended constructor*. A constructor $c$ of $C$ is an extended constructor if it requires at least one argument of type $C$. A static method $m$ of $C$ is an extended constructor if its return type is $C$ and it requires at least one argument of type $C$. If $m$ is annotated with a JML specification, the postcondition should include an assertion `\fresh(\result)` stating that the returned object should be freshly created.
- *Mutator*. An instance method $m$ is a mutator if its return type is **`void`**. If $m$ is annotated with a JML specification, it should include an *assignable clause* stating that at least one of the instance variables of the receiver may be modified. The assignable clause can be used to include static methods and non-**`void`** instance methods as mutators.
- *Observer*. All other methods of $C$ are observers. In JML, an observer is annotated with the **`pure`** modifier.

For example, the `Account` class specified in Section I-A has one basic constructor (`Account(int)`), one extended constructor (`Account(Account)`), three mutators (`deposit`, `withdraw`, and `transfer`), and one observer (`balance`).

*2) Checking Feasibility of Calls:* Whenever the algorithm selects a constructor or method to create a new instance or mutate an existing object, it ensures that the constructor or method call is indeed feasible. A call is *feasible* if the call terminates normally without throwing an exception, including assertion violation errors. This means that the call conforms to the specification and successfully produces a new instance or mutates an existing object. The feasibility check is done by dynamically calling the method or constructor and checking its termination status, as shown by the pseudo-code below.

```
feasible = false;
try {
  invoke ``r.m(a1, ..., an)'';
  feasible = true;
} catch (Exception e) {
```

[1] We consider all the methods of $C$ regardless of whether they are declared in $C$ or inherited from $C$'s superclasses. However, we only consider public methods, as we build objects dynamically by using Java's reflection facility.

```
} catch (JMLAssertionError e) {
}
```

For example, the algorithm will detect that the test cases $T_1$ and $T_4$ are infeasible and $T_3$ are meaningless (see Section II). The decision will be made as soon as the violating methods are selected and their arguments are constructed, i.e., without waiting the whole sequences to be determined. When the algorithm detects an infeasible method or constructor call, it selects a new method or constructor randomly (see Fig. 3); an alternative approach would be to try different arguments for the same infeasible method or constructor.

*3) Object Pooling:* A successfully-constructed object is pooled for later reuse if it is not a redundant object. An object is *redundant* if there already exists an object in the pool with the same or an equivalent state (refer to Section III-D for the definition of object equivalence). Object pooling increases the chance of successfully constructing a new object. It also increases the average length of the call sequences, and it is hoped that longer sequences bring a diversity on the states of generated objects.

For example, while building the test cases $T_1$-$T_5$ of Section II, the following sets of objects are inserted to the object pool:

$T_1$: $\phi$
$T_2$: {⟨Account(10)⟩, ⟨Account(20)⟩}
$T_3$: $\phi$
$T_4$: $\phi$
$T_5$: $\phi$ or {⟨Account(10); deposit(10)⟩}

Note that some test cases can't produce any (non-redundant) objects at all. The elements of a test case—the receiver and the arguments—are built from left to right, and that's why the object ⟨Account(20)⟩ from $T_1$ is not constructed and added to the pool. The account object of balance 20 produced by $T_5$ may be redundant depending on the context (see Section III-D below).

### D. Checking Redundancy

Redundant test data increases the testing time without increasing the ability to detect faults. Our algorithm removes redundant test data, and the redundancy of test data is defined in terms of the equivalence of objects and values. A test case is *redundant* if there exists an equivalent test case in the test suite. A test case is *equivalent* to another test case if the elements (i.e., the receiver and the arguments) of both test cases are pairwise equivalent. For primitive values, two values are equivalent if they are equal (==). For array values, two values are equivalent if they have the same dimension and their elements are pairwise equivalent. For objects, we define the equivalence in terms of the `equals` method and the call sequences. An instance $o_1$ of a class $C$ is said to be *equivalent* to another instance $o_2$ of $C$ if the following conditions hold:

- If $C$ overrides the `equals` method, $o_1$`.equals(`$o_2$`)` returns true.
- If $C$ doesn't override the `equals` method, the call sequences of $o_1$ and $o_1$ are pairwise equivalent; two calls are equivalent if they invokes the same method or constructor and their arguments are pairwise equivalent.

The rationale behind this definition is to leverage user-defined `equals` methods, if exist. If there is no user-defined `equals` method, we compare the sequence of method and constructor calls that construct the object. An added benefit of our use of the `equals` method is that we also uses, if exists, an equality specification, as the `equals` method is invoked with the runtime assertion checking enabled.

As an example, let's consider the test cases $T_1$-$T_5$ of Section II again. If the `Account` class doesn't override the inherited `equals` method, the object ⟨Account(20)⟩ from $T_2$ and the object ⟨Account(10); deposit(10)⟩ from $T_5$ are not equivalent to each other, as they have different call sequences. Thus, the test case $T_5$ isn't redundant (with respect to $T_2$). However, if `Account` overrides the `equals` method to introduce a new notion of equality based only on the available balance, the two objects are equivalent to each other and $T_5$ is redundant (with respect to $T_2$).

## IV. Implementation

We have implemented our algorithm for a testing tool called JET [1]. JET is a fully-automated unit testing tool for Java classes annotated with JML specifications. It generates test data dynamically, performs test execution on-the-fly, and reports test results. It can also export generated test data as JUnit test classes. As in [4], JET uses JML's runtime assertion checker as a test oracle; i.e., if an execution of the method under test results in a certain type of assertion violations (e.g., postcondition) when the generated test data is supplied, it is considered as a test failure. JET is effective in finding inconsistencies between code and its specification and helps to perform regression testing of inherited methods [1].

The earlier version of JET supported only what we call a *pure random approach* where the whole method call sequences are determined without attempting to build the represented objects. The feasibility of the sequences is checked as a side product of a test execution, as it is done as a part of constructing the objects for the test execution. We extended JET to add our algorithm as a new testing strategy. We also added several new testing options for the parameters of our algorithm, e.g., the maximum number of incremental attempts, optional use of object pool, the maximum size of the pool, and optional redundancy check. Some of these parameters are also applicable to the pure random approach.

JET has two limitations. It doesn't use public fields in the sense of directly assigning values to them, and it doesn't support object sharing. However, these are the limitations of the current implementation, not the approach itself.

## V. Evaluation

In order to evaluate the effectiveness of our approach, we used JET to generate test cases for several Java classes annotated with JML specifications. We took most of our case study classes from the JML distribution available from `www.jmlspecs.org`, and the classes include:

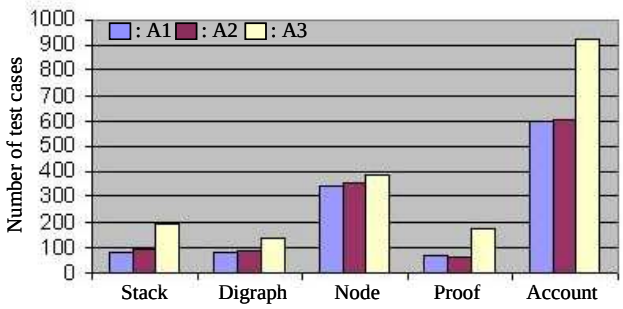


Fig. 4. Numbers of non-redundant meaningful test cases generated by different test data generation strategies: pure random (A1), pure random with object pooling (A2), and incremental random with object pooling (A3).

- BoundedStackImplementation: This is an array-based implementation of stacks and declares nine methods.
- SearchableDigraph: This class represents a directed graph and declares three methods while inheriting ten methods from its two superclasses.
- SearchableNode: This is the node class for the SearchableDigraph class. It declares nine methods and inherits two methods from its superclass.
- Proof: This is a small utility class demonstrating Floyd-Hoare-style proofs using JML. It includes four methods to find an element from an array of integer values.
- Account: This is the running example of this paper and consists of six declared methods (refer to [1] for the complete specification).

These classes have different characteristics, e.g., containers implemented as an array (BoundedStackImplementation) or a linked list (SearchableDigraph), a getter-and-setter class (SearchableNode), an integer array with non-trivial assertions (Proof), and a combination of arithmetic and getter-and-setter methods (Account). For classes with superclasses, we tested only the methods explicitly declared in the target classes However, all the inherited methods were used to generate test data, and the specifications inherited from the superclasses and the interfaces were checked at runtime.

In this experiment, we tested each class three times with different test data generation strategies.

- A1: Pure random without object pooling
- A2: Pure random with object pooling
- A3: Incremental random with object pooling

We enabled redundancy checking for all the tests. However, only the SearchableNode class overrode the `equals` method; other classes used the structural equivalence based on the method call sequences (see Section III-D).

For each test, we measured both the number of generated test cases and the elapsed time. The results of our experiment are shown in Fig. 4 and 5. As shown in Fig. 4, for all the tested classes the incremental approach outperformed the pure random approach in terms of the number of non-redundant meaningful test cases generated. The improvements were from 10 to 200%. The improvement was marginal for a simple getter-setter class (i.e., SearchableNode), but a considerable

| | *MF* | *R* | *ML* | *T* | *Time* | *SR* | *RR* |
|---|---|---|---|---|---|---|---|
| Stack | 78 | 285 | 637 | 1000 | 78 | 0.08 | 0.79 |
| Digraph | 81 | 929 | 1590 | 2600 | 452 | 0.03 | 0.92 |
| Node | 344 | 299 | 457 | 1100 | 110 | 0.31 | 0.47 |
| Proof | 69 | 128 | 503 | 700 | 421 | 0.09 | 0.65 |
| Account | 601 | 124 | 475 | 1200 | 436 | 0.50 | 0.17 |

(a) A1: Pure random

| | *MF* | *R* | *ML* | *T* | *Time* | *SR* | *RR* |
|---|---|---|---|---|---|---|---|
| Stack | 89 | 286 | 625 | 1000 | 110 | 0.08 | 0.76 |
| Digraph | 84 | 943 | 1573 | 2600 | 452 | 0.03 | 0.92 |
| Node | 355 | 298 | 447 | 1100 | 110 | 0.32 | 0.46 |
| Proof | 59 | 121 | 520 | 700 | 359 | 0.08 | 0.67 |
| Account | 604 | 112 | 484 | 1200 | 374 | 0.50 | 0.15 |

(b) A2: Pure random with object pooling

| | *MF* | *R* | *ML* | *T* | *Time* | *SR* | *RR* |
|---|---|---|---|---|---|---|---|
| Stack | 192 | 561 | 247 | 1000 | 688 | 0.19 | 0.74 |
| Digraph | 135 | 1780 | 685 | 2600 | 3086 | 0.05 | 0.93 |
| Node | 388 | 251 | 461 | 1100 | 500 | 0.35 | 0.39 |
| Proof | 178 | 136 | 386 | 700 | 4551 | 0.25 | 0.43 |
| Account | 923 | 142 | 135 | 1200 | 1839 | 0.76 | 0.13 |

(c) A3: Incremental random with object pooling

Fig. 5. Experimental results. In the tables, *MF* stands for meaningful test cases, *R* for redundant (meaningful) test cases, *ML* for meaningless test cases, *T* for total attempts, *Time* for the elapsed time in milliseconds (measured on Pentium 4 3.0 GHz with 1 GB of RAM), *SR* for success ratio ($MF/T$), and *RR* for redundancy ratio ($R/(MF+R)$).

improvement was obtained for classes with non-trivial assertions (i.e., Proof). For a typical class such as Account, we observed an increase of 48%. For container classes, the increase was 146% for the array-based stacks and 67% for the linked-list-based graphs. As expected, the redundancy rates were similar for all strategies; however, they varied dramatically depending on the characteristics of the tested classes, e.g., a lower ratio for Account and higher ratios for container classes. The experiment also showed that, contrary to our expectation, object pooling doesn't give much improvement in terms of the number of generated test cases, but it produced objects with longer calling sequences. We hope longer calling sequences make better test data, though we didn't measure the quality of test data in this experiment.

The tables in Fig. 5 also shows a downside of the incremental approach. The incremental approach is 3 to 7 times slower than the pure random approach. It is understandable, as the incremental approach executes every candidate method call with the runtime assertion checks enabled, which is known to be very slow; however, it may become a barrier to the practical use of our approach for larger complex classes.

## VI. Related Work

The most related work is our own recent work [1] that explored automated random testing to detect inconsistencies between Java classes and their JML specifications. In that work, it was shown that automated random testing could be a cost-effective alternative to ensure the correctness of JML specifications. However, test cases are generated in a pure random way in the sense that the whole method call sequence is determined at once without actually constructing the represented object. The current work extends this pure

random approach by building the sequence incrementally, i.e., by selecting one method call at a time and checking the validity of the selected method call. This incremental approach can find more meaningful test cases and produce longer call sequences (see Section V).

JCrasher [7] is an automated, random testing tool to test the robustness of Java classes. It generates a sequence of method calls to cause the class under test to crash, i.e., to throw an uncaught exception. There are two important differences between JCrasher and our approach. First, JCrasher tests the whole class as a single unit while our approach tests one method at a time. Thus, JCrasher doesn't distinguish between mutation methods and observer methods, and the generated call sequences may consist of any methods of the class under test. Second, JCrasher doesn't use formal specifications to check the validity of each call in the sequence. In fact, the whole purpose of JCrasher is to find such an invalid call that results in an uncaught exception.

Jtest [8] is a commercial tool from Parasoft that supports automatic white box testing of Java classes. It generates a collection of test cases based on code analysis, e.g., a set of test cases that execute every possible branch of the method under test. However, it's not known how the test cases are actually generated, though it doesn't look like that formal specifications are used to guide the construction of test data.

One novel feature of our approach is using formal specifications in constructing valid test data. The origin of this idea can be traced back to the use of formal specification as test oracles. Peters and Parnas proposed a tool that generates a test oracle from formal program documentation written in tabular expressions [9]. The test oracle procedure, generated in C++, checks if an input and output pair satisfies the relation described by the specification. Cheon and Leavens proposed a novel approach of employing a runtime assertion checker as a test oracle engine. [4]. In our approach, we promoted this idea further by applying it to the incremental construction of valid test data.

The redundancy of test cases is one of the main problems in generating test data randomly for object-oriented program. As the state of an object is indirectly represented as a sequence of method invocations (see Section II), it is difficult to decide whether two method invocation sequences lead to an equivalent object state or not. Xie, Marinov, and Notkin proposes a framework for detecting redundant test cases [3]. The framework supports several different techniques for detecting equivalent object states, e.g., comparing the whole or parts of call sequences, comparing concrete states, and using the `equals` method. Pacheco and Ernst compare a program's behavior on a given test case against an operational model of correct operation, derived from an example program execution [10]. The purpose is to select, from a large set of test cases, a small subset of test cases that are likely to reveal faults. As in Xie, Marinov, and Notkin's work, our approach uses the `equals` method if the class under test overrides it by proving its own definition; if not, our approach compares the structures of call sequences including arguments of method calls. However, note that our approach also uses an equality specification, if exists, as the `equals` method is invoked with the runtime assertion check enabled.

## VII. Conclusion

We explained the hidden state problem of objects on generating test data randomly for Java classes. The main problem is that it is unlikely that a randomly-chosen sequence of method calls can construct an object of a consistent state. We addressed this problem by proposing an *incremental approach* where the validity of each method call is checked as the call is selected. We implemented our approach by adding it as a new test data generation strategy to JET, an automated unit testing tool for Java. An experiment with the extended JET showed a promising result of 10 to 200% increase in the number of successfully generated test cases.

In addition to the incremental construction of objects, the contributions of our work include (1) employing a runtime assertion checker in constructing an object to check the consistency of an object's state, (2) object pooling to reuse successfully-built objects in constructing a new object, and (3) checking equivalence of objects by using the `equals` method and its specification.

## Acknowledgment

This work was supported in part by the National Science Foundation under grant CNS-0509299. We thank Myoung Kim for reading a draft of this paper.

## References

[1] Y. Cheon, "Automated random testing to detect specification-code inconsistencies," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 07-07, Feb. 2007.

[2] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[3] T. Xie, D. Marinov, and D. Notkin¡, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering (ASE'04)*. IEEE, 2004, pp. 196–205.

[4] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, ser. Lecture Notes in Computer Science, B. Magnusson, Ed., vol. 2374. Berlin: Springer-Verlag, June 2002, pp. 231–255.

[5] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Boston: Kluwer Academic Publishers, 1999, pp. 175–188.

[6] J. D. McGregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

[7] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software—Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, Sept. 2004.

[8] Parasoft Corporation, "Automatic Java software and component testing: Using Jtest to automate unit testing and coding standard enforcement," available from urlhttp://www.parasoft.com, as of Jan. 2007.

[9] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, Mar. 1998.

[10] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 — European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, A. Black, Ed., vol. 3586. Berlin: Springer-Verlag, 2005, pp. 504–527.