Nighthawk: A Two-Level Genetic-Random Unit Test Data Generator

James H. Andrews and Felix C. H. Li
Department of Computer Science
University of Western Ontario
London, Ontario, CANADA N6A 5B7
andrews,cli9@csd.uwo.ca

Tim Menzies
Lane Department of Computer Science
West Virginia University
PO Box 6109, Morgantown, WV, USA 26506
tim@menzies.us

ABSTRACT

Randomized testing has been shown to be an eœective method for testing software units. However, the thoroughness of randomized unit testing varies widely according to the settings of certain parameters, such as the relative frequencies with which methods are called. In this paper, we describe a system which uses a genetic algorithm to ønd parameters for randomized unit testing that optimize test coverage. We compare our coverage results to previous work, and report on case studies and experiments on system options.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging¯ Testing tools; I.2.8 [Artiøcial Intelligence]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Measurement

Keywords

Randomized testing, genetic algorithms, test coverage

1. INTRODUCTION

Software testing involves running a piece of software (the software under test, or SUT) on selected input data, and checking the outputs for correctness. The goals of software testing are to force failures of the SUT, and to be thorough. The more thoroughly we have tested an SUT without forcing failures, the more sure we are of the reliability of the SUT.

Randomized testing is the practice of using randomization for some aspects of test input data selection. Several independent studies [27, 1, 29, 18] have found that randomized testing of software units is eœective at forcing failures in even well-tested units. However, there remains a question of whether randomized testing can be thorough enough. Using various code coverage measures to measure thoroughness, researchers have come to varying conclusions about the ability of randomized testing to be thorough [26, 34, 1].

The thoroughness of randomized unit testing is highly dependent on parameters that control when and how randomization is applied. These parameters include the number of method calls to make, the relative frequency with which diœerent methods are called, the ranges from which integer arguments are chosen, and the manner in which previously-used arguments or previously-returned values are used in new method calls. It is often diŒcult to work out the optimal values of these parameters by hand.

In this paper, we describe Nighthawk, a system for generating unit test data. The system can be viewed as consisting of two levels. The lower level is a randomized unit testing engine which tests a set of methods according to parameter values speciøed in a chromosome. The upper level is a genetic algorithm (GA) which uses øtness evaluation, selection, mutation and recombination to ønd good values for the randomized unit testing parameters, including parameters that encode a value reuse policy. Goodness is evaluated on the basis of test coverage and number of method calls performed.

Users can use Nighthawk to ønd good parameters, and then perform randomized unit testing based on those parameters. The randomized testing can quickly generate many new test cases that achieve high coverage, and can continue to do so for as long as users wish to run it.

1.1 Randomized Testing

ıRandomȷ or ırandomizedȷ testing has a long history, being mentioned as far back as [22]. For randomized testing, an automated oracle is needed. However, studies have found that even with simple, high-pass oracles, randomized testing is eœective at forcing failures [27, 11, 29].

Randomized testing has not, however, always been found to be suŒciently thorough. For instance, Michael et al. [26] performed randomized testing on the well-known Triangle program; this program accepts three integers as arguments, interprets them as sides of a triangle, and reports whether the triangle is equilateral, isosceles, scalene, or not a triangle at all. They concluded that randomized testing could not achieve 50% condition/decision coverage of the code, even after 1000 runs. Visser et al. [34] compared randomized testing with various model-checking approaches and found that while randomized testing was good at achieving block coverage, it failed to achieve optimal coverage for stronger

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

coverage measures, such as a measure derived from Ball's predicate coverage [3].

In contrast, Doong and Frankl [14] tested several units using randomized sequences of method calls, and found that by varying some parameters of the randomized testing, they could greatly increase or decrease the likelihood of finding injected faults. The parameters included number of operations performed, ranges of integer arguments, and the relative frequencies of some of the methods in the call sequence.

1.2 Randomized Unit Testing

Unit testing is variously defined as the testing of a single method, a group of methods, a module or a class. We will use it in this paper to mean the testing of a group M of methods, called the target methods. A unit test is a sequence of calls to the target methods, with each call possibly preceded by code that sets up the arguments, and with each call possibly followed by code that checks results.

Randomized unit testing is unit testing where there is some randomization in the selection of the target method call sequence and/or arguments to the method calls. Many researchers [14, 2, 6, 31, 34] have performed randomized unit testing, sometimes combined with other tools such as model checkers.

A key concept in randomized unit testing is that of value reuse. We use this term to refer to how the testing engine reuses the receiver, arguments or return values of past method calls when making new method calls. In previous research, value reuse has mostly taken the form of making a sequence of method calls all on the same receiver object.

In our previous research, we developed a GUI-based randomized unit testing engine called RUTE-J [1]. To use RUTE-J, users write their own customized test wrapper classes, hand-coding such parameters as relative frequencies of method calls. Users also hand-code a value reuse policy by drawing receiver and argument values from value pools, and placing return values back in value pools. Finding good parameters quickly, however, requires experience with the tool.

The system Nighthawk described in this paper significantly builds on this work by automatically determining good parameters. The lower, randomized-testing, level of Nighthawk initializes and maintains one or more value pools for all relevant types, and draws and replaces values in the pools according to a policy specified in a chromosome. The chromosome also specifies relative frequencies of methods, method parameter ranges, and other testing parameters. The upper, genetic-algorithm, level performs a search for the parameter setting that causes the lower level to achieve a high value of a coverage-related measure. Nighthawk uses only the Java reflection facility to gather information about the SUT, making its general approach robust and adaptable to other languages.

1.3 Contributions and Paper Organization

The main contributions of this paper are as follows.

1. We describe the implementation of a novel two-level genetic-random testing system, Nighthawk. In particular, we describe how we encode a value reuse policy in a manner amenable to meta-heuristic search.
2. We compare Nighthawk to other systems described in previous research, showing that it can achieve the same coverage levels.
3. We describe the results of a case study carried out on real-world units (the Java 1.5.0 Collection and Map classes) to determine the effects of different option settings on the basic algorithm.

We discuss related work in section 2. In section 3, we describe the results of an exploratory study that suggested that a genetic-random approach was feasible and could find useful parameter settings. In section 4, we describe the design and use of Nighthawk. Section 5 contains our comparison to previous work, and section 6 our case study; section 7 contains a discussion of the threats to validity of the empirical work in the paper.

2. RELATED WORK

2.1 Genetic Algorithms for Testing

Meta-heuristic search methods such as GAs have often been applied to the problem of test suite generation. In Rela's review of 122 applications of meta-heuristic search in software engineering [30], 44% of the applications related to testing. Notable examples are Michael et al.'s evolutionary approach for generating code-level test data [26], Tonella's approach to class testing [32] and Guo et al.'s GA approach for generating UIO sequences for protocol testing [19].

GAs use a modified hill-climbing strategy; such strategies do not perform well when the search space is mostly flat, with steep jumps in score. Consider the problem of generating two input values x and y that will cover the true direction of the decision "x==y". If we cast the problem as a search for the two values themselves, and the score as whether we have found two equal values, the search space is shaped as in the left-hand side of Figure 1: a flat plain of zero score with a narrow ridge along the diagonal. Most approaches to GA white-box test data generation address this problem by proposing other measures that detect how close the target decision is to being true.

Our approach is essentially to instead recast the problem as a search for the best values of two variables lo and hi that will be used as the lower and upper bound for random generation of x and y, and the score as the probability of generating two equal values. Seen in this way, the search space landscape still contains a steep "cliff", as seen in the right-hand side of Figure 1, but the cliff is approached on one side by a gentle slope. We further consider not only numeric data, but data of any type.

2.2 Other Test Data Generation Approaches

Approaches to test data generation via symbolic execution have existed as far back as [7]. Other source code analysis-based approaches have used such methods as iterative relaxation of a set of constraints on input data [20] and generation of call sequences using goal-directed reasoning [24]. Some recent approaches use model checkers [33], sometimes augmented with randomized search [34, 16, 31, 28]. Groce et al. [18] have concluded that randomized testing is a good first step, before model checking, in achieving high quality software.

Our approach does not require source code or bytecode analysis, instead depending only on the robust Java reflec-

$x\|y$	1	2	3	4	5
1	1.0	.00	.00	.00	.00
2	.00	1.0	.00	.00	.00
3	.00	.00	1.0	.00	.00
4	.00	.00	.00	1.0	.00
5	.00	.00	.00	.00	1.0

$lo\|hi$	1	2	3	4	5
1	1.0	.50	.33	.25	.20
2	.00	1.0	.50	.33	.25
3	.00	.00	1.0	.50	.33
4	.00	.00	.00	1.0	.50
5	.00	.00	.00	.00	1.0

Figure 1: Chance of selecting two identical integers x and y. (left) Search space as space of (x, y) pairs. (right) Search space as space of lower and upper bounds for random generation of x and y.

tion mechanism. For instance, our code was initially written with Java 1.4 in mind, but worked seamlessly on the Java 1.5 versions of the java.util classes, despite the fact that the source code of many of the units had been heavily modiøed to introduce templates. However, model-checking approaches have other strengths, such as the ability to analyze multithreaded code [21], further supporting the conclusion that the two approaches are complementary.

2.3 Randomized Testing

As noted above, randomized testing has a long history, but faces two main problems: the oracle problem and the question of thoroughness.

There are two main approaches to the oracle problem. The ørst is to use general-purpose, ıhigh-passȷ oracles that pass many executions but check properties that should be true of most software. For instance, Miller et al. [27] fail only executions that crash or hang; Csallner and Smaragdakis [11] judge a randomly-generated unit test case as failing if it throws an exception; and Pacheco et al. [29] check general-purpose contracts for units.

The second approach is to write unit-speciøc oracles in order to check unit-speciøc properties [1]. These oracles, like all formal unit speciøcations, are non-trivial to write; tools such as Daikon for automatically deriving likely invariants [15] could help here.

Previous studies also show that randomized testing is effective in forcing failures of the SUT. Here, our research focus is not on measuring eœectiveness, but rather on optimizing commonly-used objective measures (i.e. code coverage) that can measure thoroughness even in the absence of failures; for instance, when all bugs found have been eliminated. We also generalize the problem domain from the heuristic to the meta-heuristic by adding the GA level.

3. EXPLORATORY STUDY

To ønd out whether there was any merit in the idea of a genetic-random system, we conducted an exploratory experiment. In this section, we describe the prototype software we developed, the design of the experiment and its results.

3.1 Software Developed

Using code from RUTE-J (see above) and the open-source genetic algorithm package JDEAL [10], we constructed a prototype two-level genetic-random unit testing system that took Java classes as its testing units. For each unit under test (UUT) with n methods to call, the GA level constructed a chromosome with $n + 2$ integer genes: the number of method calls to make in each test case, the number of test cases to generate, and the relative weights (calling frequencies) of the n methods. All other randomized testing parameters were hard-coded in the test wrappers.

The evaluation of the øtness of each chromosome c proceeded as follows. We got the random testing level to generate the number of test cases of the length speciøed in c, using the method weights speciøed in c. We then measured the number of coverage points covered using Cobertura [8], which measures line coverage. If we had based the øtness function only on coverage, however, then any chromosome would have beneøtted from having a larger number of method calls and test cases, since every new method call has the potential of covering more code. We therefore built in a brake to prevent these values from getting unfeasibly high. We calculated the øtness function as:

$$\text{(number of coverage points covered)} * \text{(coverage factor)} \\ - \text{(number of method calls performed)}$$

We set the coverage factor to 1000, meaning that we were willing to make 1000 more method calls (but not more) if that meant covering one more coverage point.

3.2 Experiment Design

We chose as our subject programs three units taken from the Java 1.4.2 edition of java.util: BitSet, HashMap and TreeMap. These units were clearly in wide use, and TreeMap had been used as the basis of earlier experiments [33]. For each UUT, we wrote a test wrapper class containing methods that called selected target methods of the UUT (16 methods for BitSet, 8 for HashMap and 9 for TreeMap). Each wrapper contained a simple oracle for checking correctness. We instrumented each UUT using Cobertura.

We ran the two-level algorithm 30 times on each of the three test wrappers, and recorded the amount of time taken and the parameters in the ønal chromosome. To test whether the weights in the chromosomes were useful given the length and number of method calls, for each ønal chromosome c we created a variant chromosome c' with the same length and number of method calls but with all weights equal. We then compared the coverage achieved by c and c' on 30 paired trials. Full results from the experiment are available in [25].

3.3 Results

We performed two statistical tests to evaluate whether the system was converging on a reasonable solution. First, we ordered the average weights discovered for each method in each class, and performed a t test with Bonferroni correction between each pair of adjacent columns. We found that for the HashMap and TreeMap units, the clear method (which removes all data from the map) had a statistically signiøcantly lower weight than the other methods, indicating that the algorithm was consistently converging on a solution in which it had a lower weight. This is because much of the code in these units can be executed only when there is a large amount of data in the container objects. Since the clear method clears out all the data, executing it infrequently ensured that the objects would get large enough.

We also found that for the TreeMap unit, the remove and put methods had a statistically significantly higher weight than the other methods. This is explainable by the large amount of complex code in these methods and the private methods that they call; it takes more calls to cover this code than it does for the simpler code of the other methods. Another reason is that sequences of put and remove were needed to create data structures via which code in some of the other methods was accessible.

The second statistical test we performed tested whether the weights found by the GA were efficient. For this, we used the 30 trials comparing the discovered chromosome c and the equal-weight variant c'. We found that for all three units, the equal-weight chromosome covered less code than the original, to a statistically significant level (as measured by a t test with $\alpha = 0.05$). This can be interpreted as meaning that the GA was correctly choosing a good combination of parameters.

In the course of the experiment, we found a bug in the Java 1.4.2 version of BitSet: when a call to set() is performed on a range of bits of length 0, the unit could return an incorrect length. We found that a bug report for this bug had already been submitted to Sun's bug database. It has been corrected in the Java 1.5.0 version of the library.

In summary, the experiment indicated that the two-level algorithm was potentially useful, and was consistently converging on similar solutions that were more optimal than calling all methods equally often.

4. NIGHTHAWK: SYSTEM DESCRIPTION

The results of our exploratory study encouraged us to expand the scope of the GA to include method parameter ranges, value reuse policy and other randomized testing parameters. The result was the Nighthawk system.

In this section, we first outline the lower, randomized-testing, level of Nighthawk, and then describe the chromosome that controls its operation. We then describe the genetic-algorithm level and the end user interface. Finally, we describe the use of automatically-generated test wrappers for precondition checking, result evaluation and coverage enhancement.

4.1 Randomized Testing Level

Here we present a simplified description of the algorithm that the lower, randomized-testing, level of Nighthawk uses to construct and run a test case. The algorithm takes two parameters: a set M of Java methods, and a GA chromosome c appropriate to M. The chromosome controls aspects of the algorithm's behaviour, such as the number of method calls to be made, and will be described in more detail in the next subsection.

We refer to M as the set of "target methods". We define the set I_M of types of interest corresponding to M as the union of the following sets of types[1]:

- All types of receivers, parameters and return values of methods in M.
- All primitive types that are the types of parameters to constructors of other types of interest.

Each type $t \in I_M$ is associated with an array of value pools, and each value pool for t contains an array of values of type t. Each value pool for a range primitive type (a primitive type other than boolean and void) has bounds on the values that can appear in it. The number of value pools, number of values in each value pool, and the range primitive type bounds are specified by the chromosome c.

The algorithm first chooses initial values for primitive type pools, and then moves on to non-primitive type pools. We define a constructor method to be an initializer if it has no parameters, or if all its parameters are of primitive types. We define a constructor to be a reinitializer if it has no parameters, or if all its parameters are of types in I_M. We define the set C_M of callable methods to be the methods in M plus the reinitializers of the types of I_M. The callable methods are the ones that Nighthawk calls directly.

A call description is an object representing one method call that has been constructed and run. It consists of the method name, an indication of whether the method call succeeded, failed or threw an exception, and one object description for each of the receiver, the parameters and the result (if any). A test case is a sequence of call descriptions, together with an indication of whether the test case succeeded or failed.

Nighthawk's randomized testing algorithm is referred to as constructRunTestCase, and is described in Figure 2. It

Input: a set M of target methods; a chromosome c.
Output: a test case.
Steps:

1. For each element of each value pool of each primitive type in I_M, choose an initial value that is within the bounds for that value pool.
2. For each element of each value pool of each other type t in I_M:
   (a) If t has no initializers, then set the element to null.
   (b) Otherwise, choose an initializer method i of t, call tryRunMethod(i,c), and place the result in the destination element.
3. Initialize test case k to the empty test case.
4. Repeat n times, where n is the number of method calls to perform:
   (a) Choose a target method $m \in C_M$.
   (b) Run algorithm tryRunMethod(m,c), and add the call description returned to k.
   (c) If tryRunMethod returns a method call failure indication, return k with a failure indication.
5. Return k with a success indication.

Figure 2: Algorithm constructRunTestCase.

[1]In this paper, the word "type" refers to any primitive type, interface, or abstract or concrete class.

Input: a method m; a chromosome c.
Output: a call description.
Steps:

1. If m is non-static and not a constructor:
   (a) Choose a type $t \in I$ which is a subtype of the receiver of m.
   (b) Choose a value pool p for t.
   (c) Choose one value $recv$ from p to act as a receiver for the method call.
2. For each argument position to m:
   (a) Choose a type $t \in I$ which is a subtype of the argument type.
   (b) Choose a value pool p for t.
   (c) Choose one value v from p to act as the argument.
3. If the method is a constructor or is static, call it with the chosen arguments. Otherwise, call it on $recv$ with the chosen arguments.
4. If the method call threw an AssertionError, return a call description with a failure indication.
5. Otherwise, if the method call threw some other exception, return a call description with an exception indication.
6. Otherwise, if the method return type is not void, and the return value ret is non-null:
   (a) Choose a type $t \in I$ which is a supertype of the type of the return value.
   (b) Choose a value pool p for t.
   (c) If t is not a primitive type, or if t is a primitive type and ret does not violate the bounds on p, then choose an element of p and replace it by ret.
   (d) Return a call description with a success indication.

Figure 3: Algorithm tryRunMethod.

takes a set M of target methods and a chromosome c as inputs. It begins by initializing value pools, and then constructs and runs a test case, and returns the test case. It uses an auxiliary method called tryRunMethod, described in Figure 3, which takes a method as input, calls the method and returns a call description. In the algorithm descriptions, the word ıchooseȷ is always used to mean speciøcally a random choice which may partly depend on the chromosome c.

tryRunMethod considers a method call to fail if and only if it throws an AssertionError. It does not consider other exceptions to be failures, since they might be correct responses to bad input parameters. A separate mechanism is used for detecting precondition violations and checking correctness of return values and exceptions; see Section 4.5.

For conciseness, the algorithm descriptions omit some details which we now øll in. These concern the treatment of nulls, the treatment of String, and the treatment of Object.

The receiver of a method call cannot be null, and no parameter can be null unless tryRunMethod chooses it to be. If tryRunMethod fails to ønd a non-null value when it is looking for one, it reports failure of the attempt to call the method. constructRunTestCase tolerates a certain number of these attempt failures before terminating the test case generation process.

java.lang.String is treated as if it is a primitive type, the values in the value pools being initialized with ıseed stringsȷ. Some default seed strings are supplied by the system, and the user can supply more.

Formal parameters of type java.lang.Object stand for some arbitrary object, but it is usually suŒcient to use a small number of speciøc types as actual parameters; Nighthawk uses only int and String by default. A notable exception to this rule is the parameter to the equals() method, which can be treated specially by test wrapper objects (see Section 4.5).

4.2 Chromosomes

Aspects of the test case execution algorithms are controlled by the genetic algorithm chromosome given as an argument. A chromosome is composed of a ønite number of genes. Each gene is a pair consisting of a name and an integer, Æoating-point or boolean value.

Every Nighthawk chromosome contains a gene specifying the number n of method calls that constructRunTestCase is to run. In addition, a chromosome appropriate to a set M of target methods contains the following genes:

- For each method in C_M, the relative weight of the method, i.e. the likelihood that it will be chosen at step 4(a) of constructRunTestCase.
- For each type of interest in I_M, the number of value pools for that type.
- For each value pool, the number of values in the pool.
- For each value pool of a range primitive type, the upper and lower bounds on values in the pool. Initial values are drawn from this range with a uniform distribution.
- For each method in C_M and every argument position, the chance that null will be chosen as an argument.
- For each method in C_M, the types of interest that will be chosen from to ønd a receiver, and, for each of those types, the value pools that will be chosen from.
- For each method in C_M and every argument position, the types of interest that will be chosen from to ønd an argument, and, for each of those types, the value pools that will be chosen from.
- For each method in C_M, the types of interest that will be chosen from to ønd an element that will be replaced by the return value, and, for each of those types, the value pools that will be chosen from.

The last three kinds of genes are expressed as bit vectors; each bit stands for one of the types of interest that is a subtype of the declared type (resp. one of the value pools). These bit vectors thus encode the value reuse policy expressed by the chromosome.

It is clear that diœerent gene values in the chromosome may cause dramatically diœerent behaviour of the algorithm on the methods. We illustrate this point with two concrete examples.

Consider the “triangle” unit from [26]. If the chromosome specifies that all three parameter values are to be taken from a value pool of 65536 values in the range -32768 to 32767, then the chance that the algorithm will ever choose two or three identical values for the parameters (needed for the “isosceles” and “equilateral” cases) is very low. If, on the other hand, the value pool contains only 30 integers each chosen from the range 2 to 5, then the chance rises dramatically due to reuse of previously-used values. The amount of additional coverage this would give would vary depending on the UUT, but is probably nonzero.

Consider further a container class with put and remove methods, each taking an integer key as its only parameter. If the parameters to the two methods are taken from two different value pools of 30 values in the range 0 to 1000, there is little chance that a key that has been put into the container will be successfully removed. If, however, the parameters are taken from a single value pool of 30 values in the range 0 to 1000, then the chance is very good that added values are removed, again due to value reuse. A remove method for a typical data structure executes different code for a successful removal than it does for a failing one.

4.3 Genetic Algorithm Level

We take the space of possible chromosomes as a solution space to search, and apply the GA approach to search it for a good solution. GAs have been found to be superior to purely random search in finding solutions to complex problems. Goldberg [17] argues that their power stems from being able to engage in “discovery and recombination of building blocks” for solutions in a solution space.

The parameter to Nighthawk's GA is the set M of target methods. The GA first derives an initial template chromosome appropriate to M, constructs an initial population of size p as clones of this chromosome, and mutates the population. It then performs a loop, for the desired number g of generations, of evaluating each chromosome's fitness, retaining the fittest chromosomes, discarding the rest, cloning the fit chromosomes, and mutating the genes of the clones with probability $m\%$ using point mutations and crossover (exchange of genes between chromosomes). The fitness function for a chromosome is calculated in a manner identical to the exploratory study (Section 3).

It is recognized that the design of genetic algorithms is a “black art” [23], and that very little is known about why GAs work when they do work and why they do not work when they do not. Nighthawk uses default settings of $p = 20, g = 50, m = 20$. These settings are different from those taken as standard in GA literature [12], and are motivated by a need to do as few chromosome evaluations as possible (the primary cost driver of the system). The settings of other variables, such as the retention percentage, are consistent with the literature.

To enhance availability of the software, Nighthawk uses the popular open-source coverage tool Cobertura [8] to measure coverage. Cobertura can measure only line coverage (each coverage point corresponds to a source code line, and is covered if any code on the line is executed) [2]. However, Nighthawk's algorithm is not specific to this measure; indeed, our empirical studies (see below) show that Nighthawk performs well when using other coverage measures.

[2]Cobertura (v. 1.8) also reports what it calls “decision coverage”, but this is coverage of lines containing decisions.

4.4 Top-Level Application

The Nighthawk application takes several switches and a set of class names as command-line parameters. The default behaviour is to consider the command-line class names as a set of “target classes”. If, however, the “--deep” switch is given to Nighthawk, the public declared methods of the command-line classes are explored, and all non-primitive types of parameters and return values of those methods are added to the set of target classes. The set M of target methods is computed as all public declared methods of the target classes. Intuitively, therefore, the --deep switch performs a “deep target analysis” by getting Nighthawk to call methods in the layer of classes surrounding the command-line classes.

Nighthawk runs the GA, monitoring the chromosomes and retaining the most fit chromosome ever encountered. This most fit chromosome is the final output of the program.

After finding the most fit chromosome, a test engineer can perform the randomized testing that it specifies. To do this, they run a separate program, RunChromosome, which takes the chromosome description as input and runs test cases based on it for a user-specified number of times. Randomized unit testing generates new test cases with new data every time it is run, so if Nighthawk finds a parameter setting that achieves high coverage, a test engineer can automatically generate a large number of distinct, useful test cases with RunChromosome.

4.5 Test Wrappers

We provide a utility program that, given a class name, generates the Java source file of a “test wrapper” class for the class. Running Nighthawk on an unmodified test wrapper is the same as running it on the target class; however, test wrappers can be customized for precondition checking, result checking or coverage enhancement.

A test wrapper for class X is a class that contains one private field of class X (the “wrapped object”), and one public method with an identical declaration for each public declared method of class X. Each wrapper method simply passes the call on to the wrapped object.

To customize a test wrapper for precondition checking, the user can insert a check in the wrapper method before the target method call. If the precondition is violated, the wrapper method can simply return. To customize a test wrapper for test result checking, the user can insert any result-checking code after the target method call; examples include normal Java assertions and JML [5] contracts. We provide switches to the test wrapper generation program that cause the wrapper to check commonly-desired properties, such as that a method throws no NullPointerException unless one of its arguments is null. The switch --pleb generates a wrapper that checks all the Java Exception and Object contracts from Pacheco et al. [29].

To customize a test wrapper for coverage enhancement, the user can insert extra methods that cause extra code to be executed. We provide two switches for commonly-desired enhancements. The switch --checkTypedEquals adds a method to the test wrapper for class X that takes one argument of type X and passes it to the equals method of the wrapped object. This is distinct from the normal wrapper method that calls equals, which has an argument of type

Object and would therefore by default receive arguments only of type int or String (see Section 4.1). For classes X that implement their own equals method, the typed-equals method is likely to execute more code.

Tailored serialization is accomplished in Java via specially-named private methods that are inaccessible to Nighthawk. The test wrapper generation program switch --checkSerialization adds a method to the test wrapper that writes the object to a byte array and reads it again from the byte array. This causes Nighthawk to be able to execute the code in the private serialization methods.

5. COMPARISON WITH PREVIOUS RESULTS

We compared Nighthawk with two well-documented systems in the literature by running it on the same software and measuring the results.

5.1 Pure GA Approach

To compare the results of our genetic-random approach with those of the purely genetic approach of Michael et al. [26], we adapted their published C code for the Triangle program to Java, transforming each decision so that each condition and decision direction corresponded to an executable line of code measurable by Cobertura. We then ran Nighthawk 10 separate times on the resulting class.

We found that Nighthawk reached 100% of feasible condition/decision coverage on average after 8.5 generations, in an average of 6.2 seconds of clock time [3]. Michael et al. had found that a purely random approach could not achieve even 50% condition/decision coverage. The discrepancy between the results may be due to Nighthawk being able to ønd a setting of the randomized testing parameters that is more optimal than the one Michael et al. were using. Inspection revealed that the chromosomes encoded value reuse policies that guaranteed frequent selection of the same values.

5.2 Model-Checking and Feedback-Directed Randomization

To compare our results with those of the model-checking approach of Visser et al. [34] and the feedback-directed random testing of Pacheco et al. [29], we downloaded the four data structure units used in those studies. The units had been hand-instrumented to record coverage of the deepest basic blocks in the code.

We ørst wrote restricted test wrapper classes that called only the methods called by the previous researchers. We ran Nighthawk giving these test wrapper classes as command-line classes, and observed the number of instrumented basic blocks covered, and the number of lines covered as measured by Cobertura.

Figure 4 shows the results of the comparison. We show the block coverage ratio achieved by the best Java-Pathønder-based technique from Visser et al. (JPF), by Pacheco et al.'s tool Randoop (RP), and by Nighthawk using the restricted test wrappers. Nighthawk was able to achieve the same coverage as the previous tools. The Time column shows the clock time in seconds needed by Nighthawk to achieve its greatest coverage. For BHeap and FibHeap, Nighthawk runs faster than JPF, but for the other two units it runs slower

[3]All empirical studies in this paper were performed on a Sun UltraSPARC-IIIi running SunOS 5.10 and Java 1.5.0˙11.

	Instr Blk Cov			Time	Line Cov	
UUT	JPF	RP	NH	(sec)	Restr	Full
BinTree	.78	.78	.78	.58	.84	1
BHeap	.95	.95	.95	4.1	.88	.92
FibHeap	1	1	1	5.1	.74	.92
TreeMap	.72	.72	.72	5.4	.76	.90

Figure 4: Comparison of results on the JPF subject units.

	Number of cond value combinations		
UUT	Total	Reachable	Covered
BinTree	34	28	28 (.82, 1.0)
BHeap	75	75	70 (.93, .93)
FibHeap	57	47	44 (.77, .94)
TreeMap	157	126	107 (.68, .85)

Figure 5: Multiple condition coverage of the subject units.

than both JPF and Randoop (modulo the fact that our experiments were run on a dioeerent machine architecture than that of Pacheco et al). This is not surprising, since for coverage information Nighthawk relies on general-purpose Cobertura instrumentation, which slows down programs, rather than the eŒcient, but application-speciøc, hand instrumentation that the other methods used.

We then ran Nighthawk giving the target classes themselves as command-line classes (bypassing the test wrappers), and observed the number of lines covered. The ıLine Covȷ columns show the line coverage ratio achieved when using the restricted wrappers and on the full target classes. When using the full target classes, Nighthawk was able to cover signiøcantly more lines of code, including all the blocks covered by the previous studies.

Visser et al. and Pacheco et al. also studied a form of predicate coverage [3] whose implementation is linked to the underlying Java Pathønder code, and is diŒcult for Nighthawk to access. For comparison, we therefore studied multiple condition coverage (MCC), a standard coverage metric which is, like predicate coverage, intermediate in strength between decision/condition and path coverage. We instrumented the source code so that every combination of values of conditions in every decision caused a separate line of code to be executed. We then ran Nighthawk on the test wrappers, thus eoeectively causing it to optimize MCC rather than just line coverage.

The results are in Figure 5. We list the total number of valid condition value combinations in all the code, and the number that were in decisions reachable by calling only the methods called by the other research groups. We also list the number of combinations covered by Nighthawk, both as a raw total and as a fraction of the total combinations and the reachable combinations. Nighthawk achieved between 68% and 93% of MCC, or between 85% and 100% when only reachable condition combinations were considered. These results are very good, since ıcode coverage of 70-80% is a reasonable goal for system test of most projects with most coverage metricsȷ [9].

In summary, the comparison suggests that Nighthawk was achieving good coverage with respect to the results achieved

Source øle	SLOC	PN	EN	PD	ED
ArrayList	150	111	140	109	140 (.93)
EnumMap	239	7	9	10	7 (.03)
HashMap	360	238	265	305	347 (.96)
HashSet	46	24	40	26	44 (.96)
Hashtable	355	205	253	252	325 (.92)
IHashMap	392	156	196	283	333 (.85)
LHashMap	103	27	37	28	96 (.93)
LHashSet	9	6	6	7	9 (1.0)
LinkedList	227	156	173	196	225 (.99)
PQueue	203	98	123	140	155 (.76)
Properties	249	101	102	102	102 (.41)
Stack	17	17	17	17	17 (1.0)
TreeMap	562	392	415	510	526 (.94)
TreeSet	62	44	59	41	59 (.95)
Vector	200	183	184	187	195 (.98)
WHashMap	338	149	175	274	300 (.89)
Total	3512	1914	2194	2487	2880
Ratio		.54	.62	.71	.82

Figure 6: Coverage achieved by conøgurations of Nighthawk on the java.util Collection and Map classes.

by previous researchers, even when strong coverage measures such as decision/condition and MCC were taken into consideration.

6. CASE STUDY

In order to study the eœects of diœerent test wrapper generation and command-line switches to Nighthawk, we studied the Java 1.5.0 Collection and Map classes; these are the 16 concrete classes with public constructors in java.util that inherit from the Collection or Map interface. The source øles total 12137 LOC, and Cobertura reports that 3512 of those LOC contain executable code. These units are ideal subjects because they are heavily used and contain complex code, including templates and inner classes.

For each unit, we generated test wrappers of two kinds: plain test wrappers (P), and enriched wrappers (E) generated with the ˘checkTypedEquals and ˘checkSerializable switches (see Section 4.5). We studied two diœerent option sets for Nighthawk: with no command-line switches (N), and with the ˘deep switch (see Section 4.4) turned on (D). For each ⟨UUT, test wrapper, option set⟩ triple, we ran Nighthawk and saved the best chromosome it found. For each triple, we then executed RunChromosome (see Section 4.4) specifying that it generate 10 test cases with the given chromosome, and we measured the coverage achieved.

Figure 6 shows the results of this study. The column labelled SLOC shows the total number of source lines of code reported by Cobertura (including inner classes) in the source øle associated with the class. Column PN shows the SLOC covered by Nighthawk with the plain test wrappers and no Nighthawk switches; columns EN, PD and ED show the other combinations, and column ED also shows the coverage ratio with respect to total SLOC. The second last line shows the totals for each column, and the last line shows the coverage ratio attained.

With enriched test wrappers and deep target class analysis, Nighthawk performs well, achieving over 90% coverage on 11 out of the 16 classes, and 82% coverage overall. Paired t tests with Bonferroni correction (corrected $\alpha = .00833$) on each pair of columns in the table indicate that there are statistically signiøcant diœerences between every pair except the (EN, PD) pair.

Source øle	PN	EN	PD	ED	RC
ArrayList	75	91	29	48	15
EnumMap	3	9	6	5	8
HashMap	63	37	136	176	30
HashSet	25	29	27	39	22
Hashtable	8	110	110	157	25
IHashMap	31	41	59	134	34
LHashMap	1	5	4	129	25
LHashSet	1	4	6	24	16
LinkedList	32	61	41	53	17
PQueue	23	40	242	103	13
Properties	104	19	49	47	18
Stack	5	10	5	26	8
TreeMap	80	131	231	106	26
TreeSet	110	93	98	186	26
Vector	106	83	156	176	20
WHashMap	37	35	92	110	21
Total	704	798	1291	1519	324

Figure 7: Time in seconds taken by conøgurations of Nighthawk to achieve highest coverage on the java.util Collection and Map classes.

Nighthawk performed poorly on the EnumMap class because the main constructor to EnumMap expects an enumerated type as one of the parameters. Nighthawk had no facility for supplying such a type, and so only a few lines of error code in constructors were executed. When we customized the test wrapper class so that it used a øxed enumerated type, Nighthawk covered 204 lines of code (coverage ratio .85), raising the total coverage ratio for all classes to .88.

Table 7 shows the amount of time taken by Nighthawk on the various conøgurations. In columns PN-ED, we report the number of seconds of clock time taken for Nighthawk to ørst achieve its best coverage. t tests showed that the only pairs of columns that were diœerent to a statistically signiøcant level were (PN, ED) and (EN, ED). This suggests that generating the enriched wrappers allowed Nighthawk to cover signiøcantly more code without running signiøcantly longer; the deep target class analysis also caused Nighthawk to cover signiøcantly more code, but took signiøcantly longer (though still less than 100 seconds per unit on average).

In column RC of Table 7, we report the number of CPU seconds needed for the RunChromosome program to create and run the 10 new test cases with the parameters chosen by Nighthawk in the ED conøguration. This time includes JVM startup time. The results show that with the parameters chosen by Nighthawk, RunChromosome can automatically generate many new test cases that achieve high coverage, in an average of approximately 2 seconds per test case.

7. THREATS TO VALIDITY

Here we discuss the threats to validity of the empirical results in this paper.

The representativeness of the units under test is the major threat to external validity. We studied Java collection classes because these are complex, heavily-used units that have high quality requirements. However, other units might have characteristics that cause Nighthawk to perform poorly. Randomized unit testing schemes in general require many test cases to be executed, so they perform poorly on methods that do a signiøcant amount of disk I/O or thread generation.

Nighthawk uses Cobertura, which measures line coverage, a weak coverage measure. The results that we obtained may not extend to stronger coverage measures. However, the Nighthawk algorithm does not perform special checks particular to line coverage. The comparison studies suggest that it still performs well when decision/condition coverage and MCC are simulated. The question of whether code coverage measures are a good indication of the thoroughness of testing is still, however, an area of active debate in the software testing community.

Time measurement is a threat to construct validity. We measured time using Java's systemTimeInMillis, which gives total wall clock time rather than CPU time. This may give run time numbers that do not reÆect the actual cost to the user of the testing.

8. CONCLUSIONS AND FUTURE WORK

Randomized unit testing is a promising technology that has been shown to be eœective, but whose thoroughness depends on the settings of test algorithm parameters. In this paper, we have described Nighthawk, a system in which an upper-level genetic algorithm automatically derives good parameter values for a lower-level randomized unit test algorithm. We have shown that Nighthawk is able to achieve high coverage of complex Java units. The code is available by writing to the ørst author.

Future work includes the integration into Nighthawk of useful facilities from past systems, such as failure-preserving or coverage-preserving test case minimization, and further experiments on the eœect of program options on coverage and eŒciency.

Finally, we comment that once a large community starts comparatively evaluating some technique, then evaluation methods for diœerent methods become just as important as the generation of new methods. To place this comment in an historical perspective, we note that evaluation bias is an active research area in the øeld of data mining [4, 13]. Much of our future work should hence focus on a meta-level analysis of the advantages and disadvantages of diœerent assessment criteria. Currently, there are no clear conventions on how this type of work should be assessed. For example:

- While the predicate coverage proposed by Visser et al. is an interesting assessment criteria, there is no consensus in the literature on the connection of this criteria to other measures.

- Static code analysis can direct the generation of the test cases. Our method, on the other hand, generates test cases at random, so it is possible that we cover some code more than is necessary, and that parts of our value pools are useless. Our view is that this is not a major issue since our runtimes, on real-world systems, are quite impressive. Nevertheless, there needs to be more discussion on how to assess test suite generation; i.e. runtimes versus superÆuous tests versus any other criteria.

9. ACKNOWLEDGMENTS

Thanks to Willem Visser for making the source code of the Java Pathønder subject units available, and to the JDEAL development team for their tool package. Thanks also for interesting comments and discussions to Rob Hierons, Charles Ling, Bob Mercer and Andy Podgurski. This research is supported by the ørst author's grant from the Natural Sciences and Research Council of Canada (NSERC).

10. REFERENCES

[1] J. H. Andrews, S. Haldar, Y. Lei, and C. H. F. Li. Tool support for randomized unit testing. In Proceedings of the First International Workshop on Randomized Testing (RT'06), pages 36˘45, Portland, Maine, July 2006.

[2] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal speciøcation. IEEE Transactions on Software Engineering, 26(1):55˘69, January 2000.

[3] T. Ball. A theory of predicate-complete test coverage and generation. In Third International Symposium on Formal Methods for Components and Objects (FMCO 2004), pages 1˘22, Leiden, The Netherlands, November 2004.

[4] R. R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003), pages 51˘58, Washington, DC, USA, August 2003.

[5] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, and G. T. Leavens. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, 7(3):212˘232, June 2005.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), pages 268˘279, Montreal, Canada, September 2000.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, SE-2(3):215˘222, September 1976.

[8] Cobertura Development Team. Cobertura web site. cobertura.sourceforge.net, accessed February 2007.

[9] S. Cornett. Minimum acceptable code coverage. http://www.bullseye.com/minimum.html, 2006.

[10] J. Costa, P. Silva, and N. Lopes. JDEAL Java Distributed Evolutionary Algorithms Library version 1.0: Getting started. Technical report, LaSEEB Instituto Superior Tcnico, Universidade Tcnica de Lisboa, Portugal, 2005.

[11] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. Software Practice and Experience, 34(11):1025˘1050, 2004.

[12] K. A. DeJong and W. M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In First Workshop on Parallel

Problem Solving from Nature, pages 38˘47. Springer, 1990.

[13] J. Demsar. Statistical comparisons of classiøers over multiple data sets. Journal of Machine Learning Research, 7:1˘30, 2006.

[14] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology, 3(2):101˘130, April 1994.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 27(2):99˘123, February 2001.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), pages 213˘223, Chicago, June 2005.

[17] D. E. Goldberg. Genetic Algorithm in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[18] A. Groce, G. J. Holzmann, and R. Joshi. Randomized diœerential testing as a prelude to formal veriøcation. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pages 621˘631, Minneapolis, MN, May 2007.

[19] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Computing unique input/output sequences using genetic algorithms. In 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003), volume 2931 of LNCS, pages 164˘177. Springer, 2004.

[20] N. Gupta, A. P. Mathur, and M. L. Socea. Automated test data generation using an iterative relaxation method. In Sixth International Symposium on the Foundations of Software Engineering (FSE 98), pages 224˘232, November 1998.

[21] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer, 2(4):366˘381, 2000.

[22] W. C. Hetzel, editor. Program Test Methods. Automatic Computation. Prentice-Hall, Englewood Clices, N.J., 1973.

[23] M. Kelly. Beyond the black art. EvoWeb News and Features, July 2001. evonet.lri.fr/evoweb/.

[24] W. K. Leow, S. C. Khoo, and Y. Sun. Automated generation of test programs from closed speciøcations of classes and test cases. In Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), pages 96˘105, Edinburgh, UK, May 2004.

[25] F. C. H. Li. Applications of genetic algorithms to randomized unit testing. Master's thesis, Department of Computer Science, University of Western Ontario, December 2006.

[26] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. IEEE Transactions on Software Engineering, 27(12), December 2001.

[27] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. Commun. ACM, 33(12):32˘44, December 1990.

[28] D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2003), pages 158˘165, San Francisco, July 2003.

[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pages 75˘84, Minneapolis, MN, May 2007.

[30] L. Rela. Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, 2004.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), pages 263˘272, Lisbon, September 2005.

[32] P. Tonella. Evolutionary testing of classes. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), pages 119˘128, Boston, Massachusetts, USA, July 2004.

[33] W. Visser, C. S. Psreanu, and S. Khurshid. Test input generation with Java PathFinder. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), pages 97˘107, Boston, MA, July 2004.

[34] W. Visser, C. S. Psreanu, and R. Pelnek. Test input generation for Java containers using state matching. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006), pages 37˘48, Portland, Maine, July 2006.