Using a Genetic Algorithm to Control Randomized Unit Testing

James H. Andrews, *Member, IEEE,* Felix C. H. Li, *Member, IEEE,* and Tim Menzies, *Member, IEEE*

Abstract

Randomized testing has been shown to be an effective 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 find 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. We show that we can achieve the same coverage as previous studies did, and that we can achieve high coverage on real-world software units. Finally, we describe how we used data mining techniques to analyze which genes were the most useful, and used this information to optimize the system so that it ran more quickly with no loss of coverage.

Index Terms

Software testing, randomized testing, genetic algorithms, data mining.

I. 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

Manuscript received January 1, 2001; revised January 1, 2001.

J. Andrews and F. Li are with the Department of Computer Science, University of Western Ontario, London, Ont., Canada, N6A 2B7. E-mail: andrews@csd.uwo.ca.

T. Menzies is with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-610. E-mail: tim@menzies.us.

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 [1]–[4] have found that randomized testing of software units is effective 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 [2], [5], [6].

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 different methods are called, and the ranges from which numeric arguments are chosen. The manner in which previously-used arguments or previously-returned values are used in new method calls, which we refer to as the *value reuse policy*, is also a crucial factor. It is often difficult to work out the optimal values of the parameters and the optimal value reuse policy 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 specified as genes in a chromosome, including parameters that encode a value reuse policy. The upper level is a genetic algorithm (GA) which uses fitness evaluation, selection, mutation and recombination of chromosomes to find good values for the genes. Goodness is evaluated on the basis of test coverage and number of method calls performed.

Users can use Nighthawk to find 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.

A. 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 the receiver[1], and with each call possibly followed by code that stores and 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 [3], [7]–[9] 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 [2]. 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 Nighthawk system 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.

Designing a GA means making decisions about what features are worthy of modeling and mutating. For example, much of the effort on this project was a laborious trial-and-error process of trying different chromosomes. To simplify that process, we describe experiments here with automatic feature subset selection (FSS). First, we try a very large and elaborate set of chro-

[1]We use the word "receiver" to refer to the object that a method is called on. For instance, in the Java method call "`t.add(3)`", the receiver is `t`.

mosomes. Next, we filter that set using automatic feature subset selection. The filtered set ran over 40% more quickly, with no loss of coverage. We therefore propose that automatic feature subset selection should be a routine part of the design of any large GA system.

B. 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 show that Nighthawk can achieve high coverage automatically on these units.
4) We describe how we optimized Nighthawk by using FSS to systematically analyze which genes have the greatest effect on the fitness of a chromosome, and eliminating genes that we found to have little effect. We show that the optimized system achieved the same good results in significantly less time, demonstrating the utility of augmenting GAs with automatic feature subset selection.

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. The procedure and results of our optimization are in Section 8.

II. Related Work

A. Randomized Unit Testing

"Random" or "randomized" testing has a long history, being mentioned as far back as 1973 [10]; Hamlet [11] gives a good survey. The key benefit of randomized testing is the ability to generate many distinct test inputs in a short time, including test inputs that may not be selected by test engineers but which may nevertheless force failures. There are, however, two main problems with randomized testing: the oracle problem and and the question of thoroughness.

Since randomized testing depends on the generation of many inputs, it is infeasible to get a human to check all test outputs; an automated test oracle [12] is needed. There are two main approaches to the oracle problem. The first is to use general-purpose, "high-pass" oracles that pass many executions but check properties that should be true of most software. For instance, Miller et al. [1] judge a randomly-generated GUI test case as failing only if the software crashes or hangs; Csallner and Smaragdakis [13] judge a randomly-generated unit test case as failing if it throws an exception; and Pacheco et al. [3] check general-purpose contracts for units, such as one that states that a method should not throw a "null pointer" exception unless one of its arguments is null. Despite the use of high-pass oracles, all these authors found randomized testing to be effective in forcing failures.

The second approach to the oracle problem for randomized testing is to write oracles in order to check properties specific to the software [2], [14]. These oracles, like all formal unit specifications, are non-trivial to write; tools such as Daikon for automatically deriving likely invariants [15] could help here.

Since randomized unit testing does not use any intelligence to guide its search for test cases, there has always been justifiable concern about how thorough it can be, given various measures of thoroughness, such as coverage and fault-finding ability. Michael et al. [5] 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. [6] compared randomized unit 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 coverage measures, such as a measure derived from Ball's predicate coverage [16].

Other researchers, however, have found that the thoroughness of randomized unit testing depends on how exactly it is implemented. Doong and Frankl [7] 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. Antoy and Hamlet [8], who checked the Java `Vector` class against a formal specification using random input, similarly found that if they avoided calling some of the methods (essentially setting their relative frequencies to zero), they could cover more code in the class. Andrews and Zhang [17], performing randomized unit testing on C data structures, found that varying the ranges from which integer key and data parameters were chosen increased the fault-finding ability of the random testing.

Finally, the aforementioned research by Pacheco et al. [3] enhances the thoroughness of randomized testing by doing a partial randomized breadth-first search of the search space of possible test cases, pruning branches that lead to redundant or illegal values which would cause the system to waste time on unproductive test cases.

Of the cited approaches, the approach described in this paper is most similar to Pacheco et al.'s. The primary difference is that we achieve thoroughness by generating long sequences of method calls on different receivers, while they do so by deducing shorter sequences of method calls on a smaller set of receivers. The focus of our research is also different. Pacheco et al. focus on identifying contracts for units and finding test cases that violate them. In contrast, we focus on maximizing code coverage; coverage is an objective measure of thoroughness that applies regardless of whether failures have been found, for instance in situations in which most bugs have been eliminated from a unit.

B. Analysis-Based Test Data Generation Approaches

Approaches to test data generation via symbolic execution have existed as far back as 1976 [18], [19]. Such approaches typically attempt to generate a thorough set of test cases by deducing which combinations of inputs will cause the software to follow given paths. Korel's TESTGEN system [20], for example, transforms each condition in the program to one of the form $e < 0$ or $e \leq 0$, and then searches for values that minimize (resp. maximize) e, thus causing the condition to become true (resp. false).

Other source code analysis-based approaches have used such methods as iterative relaxation of a set of constraints on input data [21] and generation of call sequences using goal-directed reasoning [22]. Some recent approaches use model checkers such as Java Pathfinder [23]. These approaches are sometimes augmented with "lossy" randomized search for paths, as in the DART and CUTE systems [24], [25], the Lurch system [26], and the Java Pathfinder-based research of Visser et al. [6].

Some analysis-based approaches are limited in the range of different conditions they consider; for instance, TESTGEN's minimization strategy [20] cannot be applied usefully to conditions involving pointers. In addition, most analysis-based approaches incur heavy memory and processing time costs, and are as yet infeasible except for small software units. These limitations are the primary reason why researchers have explored the use of heuristic and metaheuristic approaches to test case generation.

C. Genetic Algorithms for Testing

Genetic algorithms (GAs) were first described by Holland [27]. GAs typically represent a solution to a problem as a "chromosome", with various aspects of the solution to the problem represented as "genes" in the chromosomes. The possible chromosomes form a search space and are associated with a fitness function, which typically represents how good a solution the chromosome encodes. Search proceeds by evaluating the fitness of each of a population of chromosomes, and then performing point mutations and recombination on the most successful chromosomes. GAs have been found to be superior to purely random search in finding solutions to many complex problems. Goldberg [28] argues that their power stems from being able to engage in "discovery and recombination of building blocks" for solutions in a solution space.

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 [29], 44% of the applications related to testing. Approaches to GA test suite generation can be black-box (requirements-based) or white-box (code-based); here we focus on four representative white-box approaches, since our approach is coverage-based and therefore also white-box.

Pargas et al. [30] represent a set of test data as a chromosome, in which each gene encodes one input value to the software. Michael et al. [5] represent test data similarly, and conduct experiments comparing various strategies for augmenting the GA search. Both of these approaches evaluate the fitness of a chromosome by measuring how close the input is to covering some desired statement or condition direction. Guo et al. [31] generate unique input-output (UIO) sequences for protocol testing using a genetic algorithm; the sequence of genes represents a sequence of inputs to a protocol agent, and the fitness function computes a measure related to the coverage of the possible states and transitions of the agent. Finally, Tonella's approach to class testing [32] represents the sequence of method calls in a unit test as a chromosome; the approach features mutation operators customized to the problem, such as one that inserts method invocations.

D. Analytic Comparison of Approaches

Once a large community starts comparatively evaluating some technique, then *evaluation methods* for different 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 field of data mining [33], [34]. Much of our future work should hence focus on a meta-level analysis of the advantages and disadvantages of different assessment criteria. Currently, there are no clear conventions on how this type of work should be assessed. However, we can attempt some analytic comparisons.

It is clear that there are situations in which a source code analysis-based approach such as symbolic evaluation or model checking will be superior to any randomized approach. For instance, for an `if` statement decision of the form `(x==742 && y==113)`, random search of the space of all possible `x,y` pairs is unlikely to produce a test case that executes the decision in the true direction, while a simple analysis of the source code will be successful. The question is how often these situations arise in real-world programs. The system Nighthawk of this paper cannot guess at constants like 742, but is still able to cover the true direction of decisions of the form `x==y` because the value reuse policies it discovers will often choose `x` and `y` from the same value pool.

It is therefore likely that randomized testing and analysis-based approaches have complementary strengths. Groce et al. [4] conclude that randomized testing is a good first step, before model checking, in achieving high quality software, especially where the existence of a reference implementation allows differential randomized testing [35].

Genetic algorithms do not perform well when the search space is mostly flat, with steep jumps in fitness 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

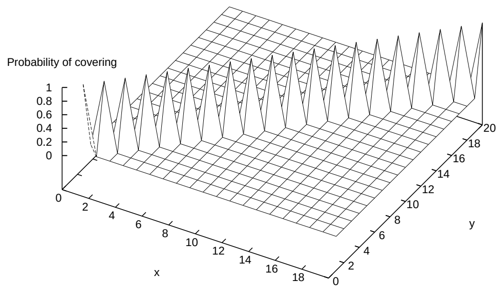

Fig. 1. Spiky search space resulting from poor fitness function.

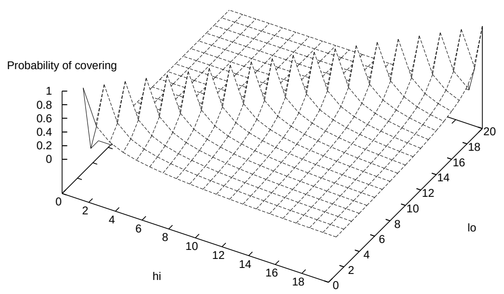

Fig. 2. Smooth search space resulting from recasting problem.

as in Figure 1: a flat plain of zero score with spikes along the diagonal. Most approaches to GA white-box test data generation attempt to address this problem by proposing fitness functions that detect "how close" the target decision is to being true, often using analysis-based techniques. For instance, Michael et al. [5] use fitness functions that specifically take account of such conditions by measuring how close `x` and `y` are. Watkins and Hufnagel [36] enumerate and compare fitness functions proposed for GA-based test case generation.

In contrast, in our research, we essentially 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 spiky "cliff", as seen in Figure 2, but the cliff is approached on one side by a gentle slope. We further consider not only numeric data, but data of any type.

In all the approaches to GA-based test input generation that we are aware of, each run of the GA results in a single test case, which is meant to reach a particular target. A test suite is built up by aiming the GA at different targets, resulting in a minimal test suite that achieves coverage of all targets. Herein lies a potential disadvantage of such approaches. Rothermel et al. [37] have shown that reducing a test suite to a minimal subset that achieves the same coverage can significantly reduce its fault detection capability. The GA can be re-run to generate more test cases, but there is a potential performance penalty since each run of the GA generates only one new test case. In contrast, in our approach, each run of the GA results in a setting for parameters for randomized testing, which can be applied cheaply many times to generate many distinct high-coverage test cases.

All analysis-based approaches share the disadvantage of requiring a robust parser and source code analyzer that can be updated to reflect changes in the source language. These complex tools are not often provided by language providers. Our approach does not require source code or bytecode analysis, instead depending only on the robust Java reflection mechanism and commonly-available coverage tools. 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 modified to introduce templates. However, model-checking approaches have other strengths, such as the ability to analyze multithreaded code [38], further supporting the conclusion that the two approaches are complementary.

III. Exploratory Study

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

A. Software Developed

Using code from RUTE-J (see above) and the open-source genetic algorithm package JDEAL [39], 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 $2 + n$ integer genes; these genes represented 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 fitness of each chromosome c proceeded as follows. We got the random testing level to generate the number of test cases of the length specified in c, using the method weights specified in c. We then measured the number of coverage points covered using Cobertura [40], which measures line coverage. If we had based the fitness function *only* on coverage, however, then any chromosome would have benefitted 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 fitness function as:

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

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.

B. 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 by other researchers [23]. 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 final chromosome. To test whether the weights in the chromosomes were useful given the length and number of method calls, for each final 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 [41].

C. 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 significantly 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 through 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 later return an incorrect "length" for the `BitSet`. 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.

IV. 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.

A. *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[2]:

- 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

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

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.

Fig. 3. Algorithm constructRunTestCase.

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 in 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 3. It 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

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 $t \in I_M$ 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 $t \in I_M$ 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 $t \in I_M$ 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.

Fig. 4. Algorithm tryRunMethod.

uses an auxiliary method called tryRunMethod, described in Figure 4, which takes a method as input, calls the method and returns a call description. In the algorithm descriptions, the word "choose" is always used to mean specifically 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 IV-E.

For conciseness, the algorithm descriptions omit some details which we now fill 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 find 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 "seed strings". 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 sufficient to use a small number of specific 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 IV-E).

B. 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 finite number of *genes*. Each gene is a pair consisting of a name and an integer, floating-point, or `BitSet` value. Figure 5 summarizes the different types of genes that can occur in a chromosome. We refer to the receiver (if any) and the return value (if non-`void`) of a method call as *quasi-parameters* of the method call. Parameters and quasi-parameters have *candidate types*:

- A type is a candidate type for a receiver if it is a subtype of the type of the receiver. These are the types from whose value pools the receiver can be drawn.
- A type is a candidate type for a parameter if it is a subtype of the type of the parameter. These are the types from whose value pools the parameter can be drawn.
- A type is a candidate type for a return value if it is a supertype of the type of the return value. These are the types into whose value pools the return value can be placed.

Gene type	Occurrence	Type	Description
`numberOfCalls`	One for whole chromosome	int	the number n of method calls to be made
`methodWeight`	One for each method $m \in C_M$	int	The relative weight of the method, i.e. the likelihood that it will be chosen
`numberOf-ValuePools`	One for each type $t \in I_M$	int	The number of value pools for that type
`numberOfValues`	One for each value pool of each type $t \in I_M$ except for `boolean`	int	The number of values in the pool
`chanceOfTrue`	One for each value pool of type `boolean`	int	The percentage chance that the value *true* will be chosen from the value pool
`lowerBound`, `upperBound`	One for each value pool of each range primitive type $t \in I_M$	int or float	The lower and upper bounds on values in the pool; initial values are drawn uniformly from this range
`chanceOfNull`	One for each argument position of non-primitive type of each method $m \in C_M$	int	The percentage chance that `null` will be chosen as the argument
`candidate-BitSet`	One for each parameter and quasi-parameter of each method $m \in C_M$	BitSet	Each bit represents one candidate type, and signifies whether the argument will be of that type
`valuePool-ActivityBitSet`	One for each candidate type of each parameter and quasi-parameter of each method $m \in C_M$	BitSet	Each bit represents one value pool, and signifies whether the argument will be drawn from that value pool

Fig. 5. Nighthawk gene types.

Note that the gene types `candidateBitSet` and `valuePoolActivityBitSet` essentially encode a value reuse policy by determining the pattern in which receivers, arguments and return values are drawn from and placed into value pools.

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

Consider the “triangle” unit from [5]. 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 keys will be 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.

C. 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. We chose GAs over other metaheuristic approaches such as simulated annealing because of our belief that recombining parts of successful chromosomes would result in chromosomes that are better than their parents. However, other metaheuristic approaches may have other advantages and should be explored in future work.

The parameter to Nighthawk’s GA is the set M of target methods. The GA performs the usual steps of chromosome evaluation, fitness selection, mutation and recombination. 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 that of the exploratory study (Section III).

Nighthawk uses default settings of $p = 20, g = 50, m = 20$. These settings are different from those taken as standard in GA literature [42], 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 [40] 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) [3]. 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.

D. 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

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

Nighthawk finds a parameter setting that achieves high coverage, a test engineer can automatically generate a large number of distinct, high-coverage test cases with RunChromosome.

E. 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 [43] 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. [3].

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 IV-A). 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.

V. 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.

A. *Pure GA Approach*

To compare the results of our genetic-random approach with those of the purely genetic approach of Michael et al. [5], 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 [4]. 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 find 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.

B. *Model-Checking and Feedback-Directed Randomization*

To compare our results with those of the model-checking approach of Visser et al. [6] and the feedback-directed random testing of Pacheco et al. [3], 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 first 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.

[4]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

Fig. 6. Comparison of results on the JPF subject units.

Figure 6 shows the results of the comparison. We show the block coverage ratio achieved by the best Java-Pathfinder-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 more quickly than JPF, but for the other two units it runs more slowly than both JPF and Randoop. This difference in run times may be in part because Nighthawk relies on general-purpose Cobertura instrumentation, which slows down programs, rather than the efficient, but application-specific, hand instrumentation that the other methods used. It may also be in part due to the fact that our experiments were run on a different machine architecture than that of Pacheco et al.

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 "Line Cov" 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 significantly 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 [16] whose implementation is linked to the underlying Java Pathfinder code, and is difficult for Nighthawk to access. While this predicate coverage is an interesting assessment criterion, there is no consensus in the literature on the connection of this criterion to other measures. 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

	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)

Fig. 7. Multiple condition coverage of the subject units.

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 effectively causing it to optimize MCC rather than just line coverage.

The results are in Figure 7. 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 "code coverage of 70-80% is a reasonable goal for system test of most projects with most coverage metrics" [44].

In summary, the comparison suggests that Nighthawk was achieving good coverage with respect to the results achieved by previous researchers, even when strong coverage measures such as decision/condition and MCC were taken into consideration.

VI. Case Study

In order to study the effects of different 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 files 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.

Source file	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

Fig. 8. Coverage achieved by configurations of Nighthawk on the `java.util` Collection and Map 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 IV-E). We studied two different option sets for Nighthawk: with no command-line switches (N), and with the `--deep` switch (see Section IV-D) turned on (D). For each $\langle$UUT, test wrapper, option set$\rangle$ triple, we ran Nighthawk for 50 generations and saved the best chromosome it found. For each triple, we then executed RunChromosome (see Section IV-D) specifying that it generate 10 test cases with the given chromosome, and we measured the coverage achieved.

Source file	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

Fig. 9. Time in seconds taken by configurations of Nighthawk to achieve highest coverage on the `java.util` Collection and Map classes.

Figure 8 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 file 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 significant differences between every pair except the (EN, PD) pair.

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 fixed enumerated type, Nighthawk with the ED configuration covered 204 lines of code (coverage ratio .85), raising the total coverage ratio for all classes to .88.

Table 9 shows the amount of time taken by Nighthawk on the various configurations. In columns PN-ED, we report the number of seconds of clock time taken for Nighthawk to first achieve its best coverage. t tests showed that the only pairs of columns that were different to a statistically significant level were (PN, ED) and (EN, ED). This suggests that generating the enriched wrappers allowed Nighthawk to cover significantly more code without running significantly longer; the deep target class analysis also caused Nighthawk to cover significantly more code, but took significantly longer (though still less than 100 seconds per unit on average).

In column RC of Table 9, 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 configuration. 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.

VII. 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 significant 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, making this a threat to construct validity.

Time measurement is also 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 reflect the actual cost to the user of the testing.

VIII. Data Mining-Based Optimization

It is recognized that the design of genetic algorithms is a "black art" [45], and that very little is known about why GAs work when they do work and why they do not work when they do not. In the initial design of Nighthawk, we got the GA to control many aspects of the randomized testing algorithm, as reflected by the ten gene types listed in Figure 5. This was the result of our expectation that any of them might turn out to be crucial to the success of the system, and our inability to predict which.

However, for a genetic algorithm, every gene that is not useful results in time wasted: time is spent mutating genes that do not lead to better solutions, and time is also spent extracting values from genes instead of simply using constant values. This effect is especially pronounced for Nighthawk, since some genes control code inside the innermost loops of the randomized testing algorithm. We were therefore motivated to find whether we could speed up Nighthawk by eliminating gene types that did not lead to better performance.

In this section, we describe how we used feature subset selection (FSS) to analyze data from Nighthawk runs, in order to systematically examine the usefulness of each gene type. We also describe how we optimized the system by eliminating the least useful genes, resulting in a system that achieved the same quality of results in less time.

A. Feature Subset Selection

A repeated result in the data mining community is that simpler models with equivalent or higher performance can be built via *feature subset selection*, algorithms that intelligently prune useless features [46]. Features may be pruned for several reasons:

- They may be noisy, i.e. contain spurious signals unrelated to the target class;
- They may be uninformative, e.g. contain mostly one value, or no repeating values;
- They may be correlated to other variables, in which case they can be pruned since their signal is also present in other variables.

The reduced feature set has many advantages:

- Miller has shown that models generally containing fewer variables have less variance in their outputs [47].
- The smaller the model, the fewer are the demands on interfaces (sensors and actuators) to the external environment. Hence, systems designed around small models are easier to use (less to do) and cheaper to build.
- In terms of this article, the most important aspect of learning from a reduced features set is that it produces smaller models. Such smaller models are easier to explain (or audit).

The literature lists many feature subset selectors. In the Wrapper method, a *target learner* is augmented with a pre-processor that used a heuristic search to grow subsets of the available features. At each step in the growth, the target learner is called to find the accuracy of the model learned from the current subset. Subset growth is stopped when the addition of new features did not improve the accuracy. Kohavi and John [48] report experiments with Wrapper where 83% (on average) of the measures in a domain could be ignored with only a minimal loss of accuracy.

The advantage of the Wrapper approach is that, if some target learner is already implemented, then the Wrapper is simple to implement. The disadvantage of the wrapper method is that each step in the heuristic search requires another call to the target learner. Since there are many steps in such a search (N features have 2^N subsets), Wrappers may be too slow.

Another feature subset selector is Relief. This is an instance based learning scheme [49], [50] that works by randomly sampling one instance within the data. It then locates the nearest neighbors for that instance from not only the same class but the opposite class as well. The values of the nearest neighbor features are then compared to that of the sampled instance and the feature scores are maintained and updated based on this. This process is specified for some user-specified M number of instances. Relief can handle noisy data and other data anomalies by averaging the values for K nearest neighbors of the same and opposite class for each instance

[50]. For data sets with multiple classes, the nearest neighbors for each class that is different from that of the current sampled instance are selected and the contributions are determined by using the class probabilities of the class in the dataset.

The experiments of Hall and Holmes [46] reject numerous feature subset selection methods. Wrapper is their preferred option, but only for small data sets. For larger data sets, the stochastic nature of Relief makes it a natural choice.

Another reason to prefer Relief is that it can take advantage of Nighthawk's stochastic search. Currently, Relief is a batch process that is executed after data generation is completed. However, it may not be necessary to wait till the end of data generation to gain insights into which features are most relevant. Given the stochastic nature of the algorithm, we can see feature work where Relief and a genetic algorithm work in tandem. Consider the random selection process at the core of Relief: a genetic algorithm exploring mutations of the current set of parents is an excellent stochastic source of data. In the future, we plan to apply Relief incrementally and in parallel to our GAs.

B. Data Collection and Analysis

We instrumented Nighthawk so that every time a chromosome was evaluated, it printed the current value of every gene and the final fitness function score. (For the two BitSet gene types, we printed only the cardinality of the set.) For each of the 16 Collection and Map classes from `java.util`, we ran Nighthawk for 40 generations. Each class therefore yielded 800 observations, each consisting of a gene value vector and the chromosome score.

Relief assumes discrete data and Nighthawk's performance scores are continuous. To enable feature subset selection, we first discretized Nighthawk's output. A repeated pattern across all our experiments is that the Nighthawk scores fall into three regions:

- The 65% majority of the scores are within 30% of the top score for any experiment. We call this the *high plateau*.
- A 10% minority of scores are less than 10% of the maximum score. We call this region *the hole*.
- After the high plateau there is a *slope* of increasing gradient that falls into *the hole*. This *slope* accounds for 25% of the results.

α	$\|Selected\|$
0.9	39
0.8	62
0.7	112
0.6	217
0.5	439

Fig. 10. Numbers of selected features for values of α.

Accordingly, to select our features, we sorted the results and divided them into three classes: bottom 10%, next 25%, remaining 65%. Relief then found the subset of features that distinguished these three regions.

Using the above discretization policy, we ran Relief 10 times in a 10-way cross-validation study. The data set was divided into 10 buckets. Each bucket was (temporarily) removed and Relief was run on the remaining data. This produced a list of "merit" figures for each feature (this "merit" value is an internal heuristic measure generated from Relief, and reflects the difference ratio of neighboring instances that have different regions). We took the maximum merit (Max) and generated a *Selected* list. A feature was *Selected* if its merit m was $m > \alpha Max$ (e.g. at $\alpha = 0.5$ then we selected all features that score at least half the maximum merit).

Figure 10 shows the number of features that were *Selected* in our 19 examples, using different values for α. Note that as α increases, we selected fewer and fewer features. That is, this method allowed us to discover the genes that were most important in selecting for the *high plateau* and not the *slope* or *hole*.

Since our goal was to identify gene types that were not useful and that could therefore possibly be eliminated from chromosomes, we tabulated the properties of each of the different types of genes. Figure 11 shows the result. The three most useful types of genes for the software under test were clearly `candidateBitSet`, `valuePoolActivityBitSet`, and `upperBound`, since genes of these types sometimes had merit 90% or more of the maximum merit, often were ranked first or second in merit, and were the only gene types such that some genes of that type had merit greater than 0.2 on some runs. This result confirms our intuition that changing the value reuse

Gene type	$\alpha = 0.9$	$\alpha = 0.5$	Best avg ranks	Best merit
`candidateBitSet`	18	106	1, 1, 1	0.373
`chanceOfNull`	0	3	24.1, 24.3, 25.4	0.166
`chanceOfTrue`	2	7	1, 3.9, 5.7	0.150
`lowerBound`	1	9	4.1, 5.6, 7.2	0.129
`methodWeight`	0	11	7.5, 7.8, 10.6	0.144
`numberOfCalls`	0	1	7.2, 8.4, 16.7	0.195
`numberOfValuePools`	0	6	14.1, 17.9, 20	0.186
`numberOfValues`	0	28	5.2, 5.2, 7.4	0.160
`upperBound`	8	16	1, 1, 1	0.267
`valuePoolActivityBitSet`	10	252	1, 1.4, 2	0.267

Fig. 11. Merit analysis of Nighthawk gene types.

policy encoded in the genes of type `candidateBitSet` and `valuePoolActivityBitSet` is an important way of improving the performance of a Nighthawk chromosome. The good performance of `upperBound` is attributable to the fact that for all the hash table-related units, adjusting this parameter caused a "load factor" parameter in some of the constructors to be able to take on values that led to the table being reallocated frequently.

Just as clearly, the two least useful gene types for the software under test were `chanceOfNull` and `numberOfValuePools`, since neither were ranked better than than 14th in merit for any run. This does not indicate that null values and multiple value pools for each type were not important, only that changing the default values of these genes (3% chance of choosing null, and two value pools) did not usually result in improved performance.

C. Optimization and Performance Comparison

In order to see whether our analysis was useful, we optimized the Nighthawk code. We decided to retain the four gene types with the highest maximum merit (`candidateBitSet`, `valuePoolActivityBitSet`, `upperBound`, and `numberOfCalls`), and also two other gene types: `lowerBound`, because it was paired with `upperBound`; and `numberOfValues`, because it had the highest maximum merit of the remaining well-ranked gene types.

Accordingly, we adjusted Nighthawk's code to delete all references to the other four gene types (`chanceOfNull`, `numberOfValuePools`, `chanceOfTrue`, and `methodWeight`). We changed the code that depended on the values stored for such genes in the current chromosome, so that it instead used the default initial values for those gene types.

On each of the 16 Collection and Map classes, we performed one run of the original Nighthawk, and one run of the new, optimized version. For each version of Nighthawk, we then performed the same analysis that is reflected in Figures 8 and 9; that is, we asked RunChromosome to run 10 test cases with the winning chromosome and measured the coverage, and we calculated the clock time taken by Nighthawk to achieve the highest coverage it achieved. We then compared the original and optimized Nighthawk.

The chromosomes resulting from the optimized Nighthawk achieved slightly higher coverage than those from the original, though a t test concluded that the difference was not statistically significant ($p = 0.058$). However, the optimized Nighthawk was faster to a statistically significant degree ($p = 0.010$). The time ratio between the original and optimized systems was 1.46, i.e. the optimized system was 46% faster. We can therefore say that the process of analyzing the usefulness of the genes resulted in a system that ran substantially more quickly but achieved the same high coverage of the SUT.

D. Discussion

The feature subset selection and optimization procedure worked well for us when we collected data from a set of classes and then optimized Nighthawk for those classes. This does not mean that the optimized Nighthawk would necessarily work well for other classes. For instance, for some classes there might be a great deal of code that is accessible only by giving null pointers as parameters; we could expect our newly optimized version of Nighthawk to perform poorly on such classes, since there would be a fixed 3% chance of choosing null.

However, the results of the optimization exercise indicate that FSS-based analysis is a valuable adjunct to the GA in this application. It is for this reason that we envision in the future integrating FSS into a genetic-random testing algorithm, so that the algorithm can improve its performance dynamically, while GA mutation and recombination is going on.

IX. Conclusions and Future Work

Randomized unit testing is a promising technology that has been shown to be effective, 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 the same coverage as earlier studies, and high coverage of complex, real-world Java units, while retaining the most desirable feature of randomized testing: the ability to generate many new high-coverage test cases quickly. We have also shown that we were able to simplify the design of the GA system and improve its runtime using automatic feature subset selection.

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 effect of program options on coverage and efficiency. We also wish to integrate a feature subset selection learner into the GA level of the Nighthawk algorithm for dynamic optimization of the GA.

Acknowledgment

Thanks to Willem Visser for making the source code of the Java Pathfinder 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 first author's grant from the Natural Sciences and Research Council of Canada (NSERC).

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, December 1990.

[2] 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)*, Portland, Maine, July 2006, pp. 36–45.

[3] 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)*, Minneapolis, MN, May 2007, pp. 75–84.

[4] A. Groce, G. J. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, May 2007, pp. 621–631.

[5] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, December 2001.

[6] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for Java containers using state matching," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, Portland, Maine, July 2006, pp. 37–48.

[7] R.-K. Doong and P. G. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 2, pp. 101–130, April 1994.

[8] S. Antoy and R. G. Hamlet, "Automatically checking an implementation against its formal specification," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 55–69, January 2000.

[9] 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)*, Montreal, Canada, September 2000, pp. 268–279.

[10] W. C. Hetzel, Ed., *Program Test Methods*, ser. Automatic Computation. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

[11] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[12] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, November 1982.

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

[14] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: Adaptive random testing for object-oriented software," in *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, May 2008, pp. 71–80.

[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*, vol. 27, no. 2, pp. 99–123, February 2001.

[16] T. Ball, "A theory of predicate-complete test coverage and generation," in *Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Leiden, The Netherlands, November 2004, pp. 1–22.

[17] J. H. Andrews and Y. Zhang, "General test result checking with log file analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 634–648, July 2003.

[18] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, September 1976.

[19] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[20] B. Korel, "Automated software test generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, August 1990.

[21] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," in *Sixth International Symposium on the Foundations of Software Engineering (FSE 98)*, November 1998, pp. 224–232.

[22] W. K. Leow, S. C. Khoo, and Y. Sun, "Automated generation of test programs from closed specifications of classes and test cases," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, Edinburgh, UK, May 2004, pp. 96–105.

[23] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, MA, July 2004, pp. 97–107.

[24] 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)*, Chicago, June 2005, pp. 213–223.

[25] 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)*, Lisbon, September 2005, pp. 263–272.

[26] 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)*, San Francisco, July 2003, pp. 158–165.

[27] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

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

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

[30] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, pp. 263–282, December 1999.

[31] 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)*, ser. LNCS, vol. 2931. Springer, 2004, pp. 164–177.

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

[33] R. R. Bouckaert, "Choosing between two learning algorithms based on calibrated tests," in *Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003)*, Washington, DC, USA, August 2003, pp. 51–58.

[34] J. Demsar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[35] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[36] A. Watkins and E. M. Hufnagel, "Evolutionary test data generation: A comparison of fitness functions," *Software Practice and Experience*, vol. 36, pp. 95–116, January 2006.

[37] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, Washington, DC, USA, November 1998, pp. 34–43.

[38] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[39] J. Costa, P. Silva, and N. Lopes, "JDEAL Java Distributed Evolutionary Algorithms Library version 1.0: Getting started," LaSEEB Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal, Tech. Rep., 2005.

[40] Cobertura Development Team, "Cobertura web site," accessed February 2007, `cobertura.sourceforge.net`.

[41] 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.

[42] 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*. Springer, 1990, pp. 38–47.

[43] 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*, vol. 7, no. 3, pp. 212–232, June 2005.

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

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

[46] M. Hall and G. Holmes, “Benchmarking attribute selection techniques for discrete class data mining,” *IEEE Transactions On Knowledge And Data Engineering*, vol. 15, no. 6, pp. 1437– 1447, 2003.

[47] A. Miller, *Subset Selection in Regression (second edition)*. Chapman & Hall, 2002.

[48] R. Kohavi and G. H. John, “Wrappers for feature subset selection,” *Artificial Intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997. [Online]. Available: citeseer.nj.nec.com/kohavi96wrappers.html

[49] K. Kira and L. Rendell, “A practical approach to feature selection,” in *The Ninth International Conference on Machine Learning*. Morgan Kaufmann, 1992, pp. pp. 249–256.

[50] I. Kononenko, “Estimating attributes: Analysis and extensions of relief,” in *The Seventh European Conference on Machine Learning*. Springer-Verlag, 1994, pp. pp. 171–182.

PLACE
PHOTO
HERE

James H. Andrews received the BSc and MSc degrees in computer science from the University of British Columbia, and the PhD degree in computer science from the University of Edinburgh. He worked from 1982 to 1984 at Bell-Northern Research, from 1991 to 1995 at Simon Fraser University, and from 1996 to 1997 on the FormalWare project at the University of British Columbia, Hughes Aircraft Canada and MacDonald Dettwiler. He is currently an Associate Professor in the Department of Computer Science at the University of Western Ontario, in London, Canada, where he has been since 1997. His research interests include software testing, semantics of programming languages, and formal specification. He is a member of the IEEE.

PLACE
PHOTO
HERE

Felix C. H. Li received the BSc and MSc degrees in Computer Science from the University of Western Ontario, in 2005 and 2007 respectively. He is currently a software developer based in Toronto.

PLACE
PHOTO
HERE

Tim Menzies received the CS and PhD degrees from the University of New South Wales and is the author of more than 160 publications. He is an associate professor at the Lane Department of Computer Science at West Virginia University (USA), and has been working with NASA on software quality issues since 1998. His recent research concerns modeling and learning with a particular focus on lightweight modeling methods. His doctoral research explored the validation of possibly inconsistent knowledge-based systems in the QMOD specification language. He is a member of the IEEE.