# Controlling Randomized Unit Testing With Genetic Algorithms

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

## Abstract

Randomized testing is an effective method for testing software units. 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 Nighthawk, a system which uses a genetic algorithm (GA) to find parameters for randomized unit testing that optimize test coverage.

Designing GAs is somewhat of a black art. We therefore use a feature subset selection (FSS) tool to assess the size and content of the representations within the GA. Using that tool, we can prune back 90% of our GA's mutators while still achieving most of the coverage found using all the mutators. Our pruned GA achieves almost the same results as the full system, but in only 10% of the time. These results suggest that FSS for mutator pruning could significantly optimize meta-heuristic search-based software engineering tools.

## Index Terms

Software testing, randomized testing, genetic algorithms, feature subset selection, search-based optimization

Manuscript received January 1, 2009; revised September 4, 2009.

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.

## 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 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 uses randomization for some aspects of test input data selection. Several 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 the thoroughness of randomized testing. 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 dependent on when and how randomization is applied; e.g. 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 call 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.

This paper describes the Nighthawk unit test data generator. Nighthawk has 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.

This paper also discusses optimization techniques for GA tools like Nighthawk. Using feature subset selection techniques, we show that we can prune many of Nighthawk's mutators (gene types) without compromising coverage. The pruned Nighthawk tool achieves nearly the same coverage as full Nighthawk (90%) and does so ten times faster. Therefore, we recommend that meta-heuristic search-based SE tools should also routinely perform subset selection.

## 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 [2], [3], [6]–[10] 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 here 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. Chromosomes specifies relative frequencies of methods, method parameter ranges, and other testing parameters. The upper, genetic-algorithm, level searches for the parameter settings that increases the coverage seen in the lower level. The information that Nighthawk uses about the SUT is only information about type names, method names, parameter types, method return types, and which classes are subclasses of others; this makes its general approach robust and adaptable to other languages.

Designing a GA means making decisions about what features are worthy of modeling and

[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`.

mutating. For example, much of the effort on this project was a laborious trial-and-error process of trying different types of genes within a chromosome. To simplify that process, we describe experiments here with automatic feature subset selection (FSS), which lead us to 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 Nighthawk, a novel two-level genetic-random testing system that encodes a value reuse policy in a manner amenable to meta-heuristic search.
2) We demonstrate the value of feature subset selection (FSS) for optimizing genetic algorithms. Using FSS, we can prune Nighthawk's gene types while achieving nearly the same coverage. Compared to full Nighthawk, this coverage is achieved ten times faster.
3) We offer evidence for the external validity of our FSS-based optimization: the optimization learned from one set of classes (Java utils) also works when applied to another set of classes (classes from the Apache system).

We discuss related work in Section II. Section III. describes our system and gene type pruning using feature subset selection is described in Section IV. Section VI describes threats to validity, and Section VII concludes.

This paper differs from prior publications as follows:

- The original Nighthawk publication [11] studied its effectiveness on the JAVA util classes.
- A subsequent paper [12] offered an FSS-based optimization. That paper found ways to prune 60% of the gene types when trying to cover the Java util classes.
- This paper shows that that FSS-based optimizer was sub-optimal. We present here a better optimization method that prunes 90% of the gene types (see §IV-D2). An analysis of that optimized system in §IV-D3 revealed further improvements. We have successfully tested the optimized and improved version of Nighthawk on Apache system classes (see §IV-D4).

## II. Related Work

### A. *Randomized Unit Testing*

"Random" or "randomized" testing has a long history, being mentioned as far back as 1973 [13]; Hamlet [14] 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.

Randomized testing depends on the generation of so many inputs that it is infeasible to get a human to check all test outputs. An automated test oracle [15] 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 [16] 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], [17]. These oracles, like all formal unit specifications, are non-trivial to write; tools such as Daikon for automatically deriving likely invariants [18] 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 a measure derived from Ball’s predicate coverage [19].

Other researchers, however, have found that the thoroughness of randomized unit testing depends on the implementation. Doong and Frankl [7] tested several units using randomized sequences of method calls. By varying some parameters of the randomized testing, they could greatly increase/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 the relative frequencies of those methods to zero), they could cover more code in the class. Andrews and Zhang [20], 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.

Pacheco et al. [3] show that randomized testing can be enhanced via 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 date back to 1976 [21], [22]; typically these approaches generate a thorough set of test cases by deducing which combinations of inputs will cause the software to follow given paths. TESTGEN [23], 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 tools have applied iterative relaxation of a set of constraints on input data [24] and generation of call sequences using goal-directed reasoning [25]. Some recent approaches use model checkers such as Java Pathfinder [26]. These approaches are sometimes augmented with "lossy" randomized search for paths, as in the DART and CUTE systems [10], [27], the Lurch system [28], and the Java Pathfinder-based research of Visser et al. [6].

Some analysis-based approaches limit the range of different conditions they consider; for instance, TESTGEN's minimization strategy [23] cannot be applied to conditions involving pointers. In addition, most analysis-based approaches incur heavy memory and processing time costs. 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 [29]. Candidate solutions are represented as "chromosomes", with solution represented as "genes" in the chromosomes. The possible chromosomes form a search space and are associated with a fitness function representing the value of solutions encoded in the chromosome. Search proceeds by evaluating the fitness of each of a population of chromosomes, and then performing point mutations and recombination on the successful chromosomes. GAs can defeat purely random search in finding solutions to complex problems. Goldberg [30] 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 SE [31], 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 focuses on increasing coverage, and is therefore also white-box.

Pargas et al. [32] represent a set of test data as a chromosome, in which each gene encodes one input value. 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. [33] 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 [34] represents the sequence of method calls in a unit test as a chromosome; the approach features customized mutation operators, such as one that inserts method invocations.

## D. Nighthawk

In work reported in [11], we developed Nighthawk, the two-level genetic-random test data generation system explored further in this paper, and carried out experiments aimed at comparing it with previous research and finding the optimal setting of program switches.

Unlike the methods discussed above, Nighthawk's genetic algorithm does not result in a single test input. Instead, it finds settings to parameters which control aspects of randomized testing. We designed Nighthawk by identifying aspects of the basic randomized testing algorithm that would benefit from being controlled by parameters, and then encoding each parameter as a gene in a chromosome. Details of the design of Nighthawk are presented in Section III.

Our initial empirical studies showed that, when run on the subject software used by Michael et al. [5], Nighthawk reached 100% of feasible condition/decision coverage on average after 8.5 generations. They also showed that, when run on the subject software used by Visser et al. [6] and Pacheco et al. [3], Nighthawk achieved the same coverage in a comparable amount of time. Finally, our studies showed that Nighthawk could achieve high coverage (82%) automatically, when using the best setting of system parameters, when run on the 16 Collection and Map classes from the `java.util` package in Java 1.5.0. These results encouraged us to explore Nighthawk further. The full empirical results are described in [11].

## E. 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*. 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 Nighthawk system 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.

Genetic algorithms do not perform well when the search space is mostly flat, with steep jumps in fitness score. Consider the problem of finding integer inputs $x$ and $y$ that cover the true direction of the decision "$x$==$y$". If we cast the problem as a search for the two values, and the score as whether we have found two equal values, the search space is shaped as in Figure 1: a flat plain of zero score with spikes along the diagonal. Most approaches to GA white-box test data generation use 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 [35] enumerate and compare fitness functions proposed for GA-based test case generation.

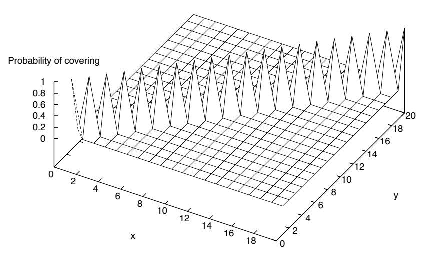

Fig. 1. Spiky search space resulting from poor fitness function.

In contrast, we recast the problem as a search for the best values of $lo$ and $hi$ that will be used as the lower and upper bound for random generation of $x$ and $y$, and the score as whether we have generated two equal values of $x$ and $y$. 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.

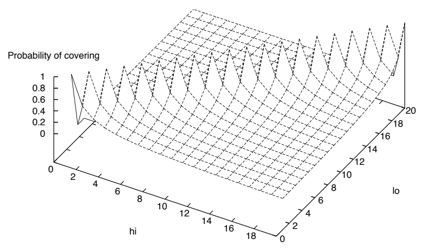

Fig. 2. Smooth search space resulting from recasting problem.

If the inputs in Figure 1 were floating-point numbers (not integers), the search space would consist of a flat plain of zero score with a thin, sharp ridge along the diagonal. In this case the solution depicted in Figure 2 would yield only a small improvement. This is where value pools come in. We draw each parameter value from a value pool of finite size; each numeric value pool has size $s$ and bounds $lo$ and $hi$, and is initially seeded with values drawn randomly within $lo$ to $hi$. For the example problem, we will be more likely to choose equal $x$ and $y$ as $s$ becomes smaller, regardless of the value of $lo$ and $hi$ and regardless of whether the values are integers or floating-point numbers, because the smaller the value pool, the more likely we are to pick the same value for $x$ and $y$.

This approach generalizes to non-numeric data. Each type of interest is associated with one or more value pools; the number and size of which are controlled by genes. At any time, a value in a value pool may be chosen as the receiver or parameter of a method call, which may in turn change the value in the pool. Also, at any time a value in a value pool may be replaced by a return value from a method call. Which value pools are drawn on, and which value pools receive the return values of which methods, are also controlled by genes. A test case may consist of hundreds of randomized method calls, culminating in the creation of values in value pools which, when used as parameters to a method call, cause that method to execute code not executed before. Changing gene values therefore makes this more/less likely to happen.

To the best of our knowledge, each run of previous GA-based tools has resulted 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 fixed-size test suite that achieves coverage of all targets. However, Frankl and Weiss [36] and Andrews and Siami Namin [37] have shown that both size and coverage exert an influence over test suite effectiveness, and Rothermel et al. [38] have shown that reducing test suite size while preserving coverage can significantly reduce its fault detection capability. Therefore, given a choice between three systems achieving the same coverage, (a) which generates one fixed set of test cases, (b) which generates many different test cases slowly, and (c) which generates many different test cases quickly, (c) is the optimal choice. A GA which generates one test case per run is in class (a) or (b) (it may generate different test cases on each run as a result of random mutation). In contrast, Nighthawk is in class (c) because each run of the GA results only in a setting of randomized testing parameters that achieves high coverage; new high-coverage test suites can then be generated quickly at low cost.

All analysis-based approaches share the disadvantage of requiring a robust parser and/or an analyzer for source code, byte code or machine code that can be updated to reflect changes in the source language. As an example from the domain of formal specification, Java 1.5 was released in 2004, but as of this writing, the widely-used specification language JML does not fully support Java 1.5 features [39]. Our approach does not require source code or bytecode analysis, instead depending only on class and method parameter information (such as that supplied by 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 multi-threaded code [40], further supporting the conclusion that the two approaches are complementary.

## III. Nighthawk: System Description

Our exploratory studies [11] suggested that GAs generating unit tests should search method parameter ranges, value reuse policy and other randomized testing parameters. This section describes Nighthawk's implementation of that search. We 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 automatically-generated test wrappers for precondition checking, result evaluation and coverage enhancement.

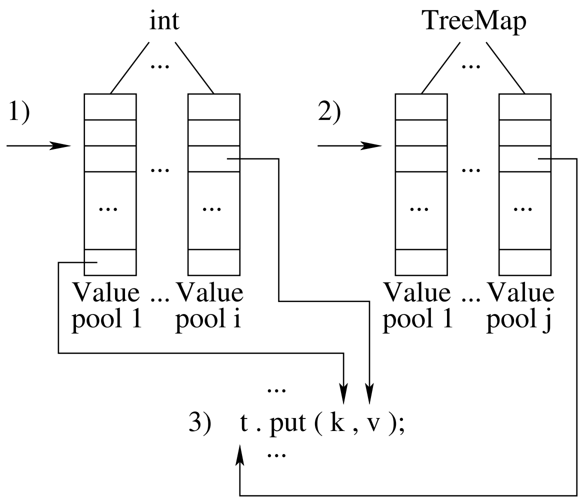

Fig. 3. Value pool initialization and use. Stage 1: random values are seeded into the value pools for primitive types such as `int`, according to bounds in the pools. Stage 2: values are seeded into non-primitive type classes that have initializer constructors, by calling those constructors. Stage 3: the rest of the test case is constructed and run, by repeatedly randomly choosing a method and receiver and parameter values. Each method call may result in a return value, which is placed back into a value pool (not shown).

### A. *Randomized Testing Level*

Nighthawk's lower level constructs and runs one 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 behavior, such as the number of method calls to be made, and will be described in more detail in the next subsection. We say that $M$ is the set of "target methods". $I_M$, the *types of interest* corresponding to $M$, is the union of the following sets of types[2]:

[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 $t$, and call tryRunMethod($i, c$). If the call returns a non-null value, 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 tryRunMethod($m, c$). Add the returned call description 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. 4. Algorithm constructRunTestCase.

- 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 $t \in I_M$ has an array of *value pools*. 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 chromosome $c$.

See Figure 3 for a high-level view of how the value pools are initialized and used in the test case generation process. The algorithm chooses initial values for primitive type pools, before considering non-primitive type pools. A constructor method is an *initializer* if it has no parameters, or if all its parameters are of primitive types. A constructor is a *reinitializer* if it has no parameters, or if all its parameters are of types in $I_M$. (All initializers are also reinitializers.) We define the set $C_M$ of *callable methods* to be the methods in $M$ plus the reinitializers of the types in $I_M$ (Nighthawk calls these *callables* 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 4. 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 uses an auxiliary method called tryRunMethod (Figure 5), which takes a method as

```
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 call throws AssertionError, return a failure indication call description.
  5) Otherwise, if the call threw another exception, return a call description with an exception indication.
  6) Otherwise, if the method return is not void, & the return value $ret$ is non-null:
       a) Choose type $t \in I_M$ that is a supertype of the 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 $p$ bounds, then replace an element of $p$ with $ret$.
       d) Return a call description with a success indication.
```

Fig. 5. Algorithm tryRunMethod.

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 $c$.

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. We facilitate checking correctness of return values and exceptions by providing a generator for "test wrapper" classes. The generated test wrapper classes can be instrumented with assertions; see Section III-E for more details.

Return values may represent new object instances never yet created during the running of the test case. If these new instances are given as arguments to method calls, they may cause the method to execute statements never yet executed. Thus, the return values are valuable and are returned to the value pools when they are created.

Although we have targeted Nighthawk specifically at Java, note that its general principles apply to any object-oriented or procedural language. For instance, for C, we would need only information about the types of parameters and return values of functions, and the types of fields in `structs`. `struct` types and pointer types could be treated as classes with special constructors, getters and setters; functions could be treated as static methods of a single target class.

## *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 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 | Lower and upper bounds on pool values; 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 |
| `candidateBitSet` | One for each parameter and quasi-parameter of each method $m \in C_M$ | BitSet | Each bit represents 1 candidate type, signifying if 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. 6. Nighthawk gene types.

gene is a pair consisting of a name and an integer, floating-point, or `BitSet` value. Figure 6 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 *receiver candidate type* 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 *parameter candidate type* 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 *return value candidate type* 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.

Note that the gene types `candidateBitSet` and `valuePoolActivityBitSet` encode value reuse policies 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 behavior of the algorithm on the methods. We illustrate this point with two concrete examples.

Consider the "triangle" unit from [5]. If the value pool for all three parameter values contains 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 0 to 10, then the chance rises dramatically due to reuse of previously-used values (the additional coverage this would give would depend on the SUT, but is probably $> 0$).

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 chromosome evaluation steps (fitness selection, mutation & recombination). The GA 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 loops 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 evaluation of the fitness of each chromosome $c$ proceeds as follows. The random testing level of Nighthawk generates and runs a test case, using the parameters encoded in $c$. It then collects the number of lines covered by the test case. If we based the fitness function *only* on coverage, then any chromosome would benefit from having a larger number of method calls and test cases, since every new method call has the potential of covering more code. Nighthawk therefore calculates the fitness of the chromosome as:

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

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

For the three variables mentioned above, Nighthawk uses default settings of $p = 20, g = 50, m = 20$. These settings are different from those taken as standard in GA literature [41], and are motivated by a need to do as few chromosome evaluations as possible (the primary cost driver of the system). The population size $p$ and the number of generations $g$ are smaller than standard, resulting in fewer chromosome evaluations; to compensate for the lack of diversity in the population that would otherwise result, the mutation rate $m$ is larger. 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 [42] 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). However, Nighthawk's algorithm is not specific to this measure.

## D. *Top-Level Application*

The Nighthawk application takes several switches and a set of class names as command-line parameters. Our empirical studies [11] showed that it is best to consider the set of "target classes" as the command-line classes together with all non-primitive types of parameters and return values of the public declared methods of the command-line classes. The set $M$ of target *methods* is computed as all public declared methods of the target *classes*.

Nighthawk runs the GA, monitoring the chromosomes and retaining the first chromosome that has the highest fitness ever encountered. This most fit chromosome is the final output of the program. After finding the most fit chromosome, test engineers can apply the specified randomized test. To do this, they run a separate program, RunChromosome, which takes the chromosome description as input and runs test cases 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, 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. 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 with 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 passes calls to the wrapped object.

To improve test wrapper precondition checking, users can add checks in a wrapper method before the target method call. When preconditions are violated, the wrapper method just returns. To customize a 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. Switches to the test wrapper generation program can make the wrapper check commonly-desired properties; e.g. a method throws no `NullPointerException` unless one of its arguments is null. The switch `--pleb` generates a wrapper that checks the Java Exception and Object contracts from Pacheco et al. [3]. Test wrapper generation is discussed further in [11].

## IV. Analysis and Optimization of the GA

Our earlier work convinced us that Nighthawk was capable of achieving high coverage. However, a pressing question remains: is Nighthawk spending time usefully in its quest for good chromosomes? In particular, are all the aspects of the randomized testing level that are controlled by genes the best aspects to be so controlled? If the answer to this question is "no", then the GA level of Nighthawk could be wasting time mutating genes that have little effect on cost-effective code coverage; furthermore, the randomized testing level could be wasting time retrieving the values of genes and changing its behavior based on them, when it could be using hard-coded values and behavior. If this is the case, then further research work, on areas like comparing GA search to other techniques such as simulated annealing, could be a waste of time because the GA under study is using useless genes.

In general, it could be the case that some types of genes used in a GA are more useful and some are less useful. If this is the case, then we need to identify the least useful genes, determine if there is any benefit to eliminating them, and if so, do so.

In order to check the utility of Nighthawk genes, we turn to feature subset selection (FSS). As shown below, using FSS, we were able to identify and eliminate useless genes, find a better initial value for one major gene type, and come to a better understanding of the tradeoffs between coverage and performance of Nighthawk. In particular, we found that Nighthawk can run an order of magnitude faster while maintaining nearly the same level of coverage.

In this section, we first discuss the motivation of this work in more detail, and then describe the FSS method we selected. We describe the four major types of analysis activities we undertook, and then describe how we iteratively applied them. We end the section with conclusions about Nighthawk and about FSS-based analysis of GAs in general.

### A. Motivation

The search space of a GA is the product of the sets of possible values for all genes in a chromosome. In the simplest case, where all genes have $R$ possible values and there are $L$ genes, the size of this search space is $R^L$. The run time cost to find the best possible chromosome is therefore proportional to this size times the evaluation cost of each chromosome:

$$cost = R^L * eval \tag{1}$$

Nighthawk's chromosomes for the `java.util` classes range in size from 128 genes to 1273 genes (recall that the number of genes is dependent on such things as the number of target methods and the numbers of parameters of those methods), and each gene can have a large number of values. Nighthawk's chromosomes store information related to the gene types of Figure 6. For example, for the public methods of `java.util.Vector`, Nighthawk uses 933 genes, 392 of which are `valuePoolActivityBitSet` genes, and 254 of which are `candidateBitSet` genes. If we could discard some of those gene types, then Equation 1 suggests that this would lead to a large improvement in Nighthawk's runtimes.

We can get information about which genes are valuable by recording, for each chromosome, the values of the genes and the resulting fitness score of the chromosome. This leads to a large volume of data, however: since the population is of size 20, there are 20 vectors of data for each generation for each unit being tested. We can interpret our problem as a data mining problem. Essentially, what we need to know from this data is what information within it is not needed to accurately predict the fitness score of a chromosome.

Feature subset selection (FSS) is a data mining technique that removes needless information. A repeated result is that simpler models with equivalent or higher performance can be built via FSS [44]. Features may be pruned for several reasons:

- *Noisy:* spurious signals unrelated to the target;
- *Uninformative:* contain mostly one value, or no repeating values;
- *Correlated to other variables:* so, if pruned, their signal will remain in other variables.

Apart from reduced runtimes, using fewer features has other advantages. Miller has shown that models generally containing fewer variables have less variance in their outputs [45]. Also, the smaller the model, the fewer are the demands on interfaces to the external environment. Hence, systems designed around small models are easier to use (less to do) and cheaper to build.

## B. Selecting an FSS Method

The RELIEF feature subset selector [46], [47] assumes that the data is divided into $groups$[3] and tries to find the features that serve to distinguish instances in one group from instances in other groups.

RELIEF is a stochastic instance-based scheme that works by randomly selecting $N$ reference instances $R_1..R_N$; by default, $N = 250$. For data sets with two groups, RELIEF can be implemented using the simple algorithm of Figure 7. For each instance, the algorithm finds two other instances: the "hit" is the nearest instance to $R$ in the same group while the "miss" is the nearest instance to $R$ in another group. RELIEF's core intuition is that features that change value between groups are more meritorious than features that change value within the same group. Accordingly, the merit of a feature (denoted $M_f$) is *increased* for all features with a different value in the "miss" and *decreased* for all features with different values

```
for $f \leftarrow 1$ to $|features|$ do
    $M_f = 0$                          // set all merits to 0
done
for i $\leftarrow 1$ to $N$ do
    randomly select instance $R$ from group $G$
    find nearest hit $H$     // closest thing in the same group
    find nearest miss $M$    // closest thing in a different group
    for $f \leftarrow 1$ to $|features|$ do
        $M_f \leftarrow M_f - \frac{\Delta(f,R,H)}{N} + \frac{\Delta(f,R,M)}{N}$
    done
done
```

Fig. 7. Binary RELIEF (two group system) for $N$ instances for merit of different features.

[3]Technically, RELIEF assumes that instances have been classified using some "class" attribute. However, to avoid confusion with the concept of "class" discussed above, we will describe RELIEF in terms of "groups".

in the "hit". The $\Delta$ function of figure Figure 7 detects differences between feature values. If a feature is discrete then the distance is one (if the symbols are different) or zero (if the symbols are the same). If a feature is numeric, then the distance is the difference in value (normalized to 0...1). If a feature has a missing value, then a Bayesian statistic is used to generate an estimate for the expected value (see [46] for details). For a complex data set with $k > 2$ groups, RELIEF samples the $k$ nearest misses and hits from the same or different groups.

Hall and Holmes [44] review and reject numerous FSS methods. Their favorite method (called WRAPPER) is suitable only for small data sets. For larger data sets, they recommend RELIEF.

## *C. Analysis Activities*

In our FSS analysis of Nighthawk, we iteratively applied three distinct analysis activities, which we refer to as *merit analysis*, *gene type ranking*, and *progressive gene type knockout*.

*1) Merit Analysis:* Using data from a run of Nighthawk on a set of subject units, merit analysis finds a "merit" score between 0.0 and 1.0 for each of the genes corresponding to the subject units (higher merits indicates that that gene was more useful in producing a fit chromosome).

To prepare for merit analysis, we modified Nighthawk so that each chromosome evaluation 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.) The input to the merit analysis, for a set of subject classes, is the output of one run of the modified Nighthawk for 40 generations on each of the subject classes; by 40 generations, the fitness score had usually stabilized, and we did not want to bias our dataset by including many instances with high score. Each subject class therefore yielded 800 instances, each consisting of the gene values and the chromosome score.

RELIEF assumes discrete data, but Nighthawk's fitness scores are continuous. We therefore discretized Nighthawk's output:

- 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 (called *the hole*).
- The remaining data *slopes* from the plateau into the hole.

We therefore assigned each instance to one of three groups (plateau, slope and hole), and gave the data to RELIEF to seek features (in this context, genes) that distinguished between the groups. This produced one merit figure for each feature (gene).

| Rank | Gene type $t$ | $avgMerit$ |
|---|---|---|
| 1 | `numberOfCalls` | 85 |
| 2 | `valuePoolActivityBitSet` | 83 |
| 3 | `upperBound` | 64 |
| 4 | `chanceOfTrue` | 50 |
| 5 | `methodWeight` | 50 |
| 6 | `numberOfValuePools` | 49 |
| 7 | `lowerBound` | 44 |
| 8 | `chanceOfNull` | 40 |
| 9 | `numberOfValues` | 40 |
| 10 | `candidateBitSet` | 34 |

Fig. 8. Nighthawk gene types sorted by $avgMerit$, the average RELIEF merit over all genes of that type and all subject units.

Each run therefore also yielded a ranked list $R$ of all genes, where gene 1 had the highest merit for this run, gene 2 had the second highest merit, and so on. We define:

- $merit(g, u)$ is the RELIEF merit score of gene $g$ derived from unit $u$.
- $rank(g, u)$ is the rank in $R$ of gene $g$ derived from unit $u$.

Note that higher/lower $merits/rank$ indicate a more important gene (respectively).

*2) Gene Type Ranking:* Nighthawk uses the ten gene types listed in Figure 6. However, recall that each gene type $t$ may correspond to zero or more genes, depending on the unit under test. In order to eliminate gene *types*, we need to rank them based on the merit scores for the *individual* genes. We refer to this activity as *gene type ranking*.

We used four gene type rankings in our analysis. Each was based on assigning a numerical score to the gene type derived from the merit scores of the genes of that type.

- $bestMerit(t)$ is the maximum, over all genes $g$ of type $t$ and all subject units $u$, of $merit(g, u)$. Ranking by $bestMerit$ favors genes that showed high merit for some unit.
- $bestRank(t)$ is the minimum, over all genes $g$ of type $t$ and all subject units $u$, of $rank(g, u)$. Ranking by $bestRank$ favors genes that were important in Nighthawk's handling of some unit, regardless of their absolute merit score.
- $avgMerit(t)$ is the average, over all genes $g$ of type $t$ and all subject units $u$, of $merit(g, u)$. Ranking by $avgMerit$ favors genes that consistently showed high merit across all units.
- $avgRank(t)$ is the average, over all genes $g$ of type $t$ and all subject units $u$, of $rank(g, u)$. Ranking by $avgRank$ favors genes that were consistently important across all units.

For example, Figure 8 shows the ten gene types from Figure 6, ranked in terms of their $avgMerit$ as defined above, resulting from running version 1.0 of Nighthawk on the first set of subject units. This ranking places `numberOfCalls` at the top, meaning that it considers

| | Rank of gene type when ranked by measure | | | |
|---|---|---|---|---|
| Gene type | $bestMerit$ | $bestRank$ | $avgMerit$ | $avgRank$ |
| `numberOfCalls` | 4 | 7 | 1 | 1 |
| `valuePoolActivityBitSet` | 3 | 3 | 2 | 2 |
| `upperBound` | 2 | 2 | 3 | 3 |
| `chanceOfTrue` | 8 | 4 | 4 | 4 |
| `methodWeight` | 9 | 8 | 5 | 8 |
| `numberOfValuePools` | 5 | 9 | 6 | 6 |
| `lowerBound` | 10 | 5 | 7 | 5 |
| `chanceOfNull` | 6 | 10 | 8 | 9 |
| `numberOfValues` | 7 | 6 | 9 | 7 |
| `candidateBitSet` | 1 | 1 | 10 | 10 |

Fig. 9. Ranks of all gene types, when ranked by four measures computed from data on the first set of subject units.

genes of that type to be the most valuable; it also places `candidateBitSet` at the bottom, meaning that it considers genes of that type to be the most expendable.

However, note also Figure 9, which compares the ranks of the gene types from the first set of subject units. Some gene types, such as `valuePoolActivityBitSet`, have fairly consistent ranks, whereas others, such as `candidateBitSet` and `numberOfCalls`, have quite different ranks when different ranking measures are used.

*3) Progressive Gene Type Knockout:* To validate the rankings found by gene type rankings, we conducted a *progressive gene type knockout* experiment. We instrumented the Nighthawk source code so that we could easily "knock out" all genes of a given type, by replacing the code controlled by that gene type by code that assumed a constant value for each gene of that type. For example, if we chose to knock out the `numberOfCalls` gene type, then no information about the number of method calls to be made was contained in the chromosome, and the randomized testing level made the same number of calls in each case.

We then ran Nighthawk on the subject units again, first with all ten gene types, then with the lowest (least useful) gene type in the gene type ranking knocked out, then with the lowest two gene types knocked out, and so on until we were left with one gene type not knocked out. We collected two result variables from each run on each subject unit: the amount of time it took, and the coverage achieved by the winning chromosome. We then inspected the results using data visualization methods in order to evaluate the tradeoffs in cost (time) and benefits (coverage).

## D. Analysis Procedure

*1) Stage 1: Initial Analysis:* In the first stage of our analysis, we ran Nighthawk on 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, which we used for our earlier experiments [11]. The source files total 12137 LOC, and Cobertura reports that 3512 of those LOC contain executable code.

We performed a merit analysis and gene type ranking based on $bestMerit$ and $bestRank$. We then proceeded with progressive gene type knockout based on the bestMerit and bestRank rankings of gene types. The results of this initial progressive gene type knockout experiment were reported in [12]. We showed that with only the best four gene types according to the $bestMerit$ ranking, or the best seven gene types according to the $bestRank$ ranking, we were able to achieve 90% of the coverage achieved by all ten gene types, in about 10% of the time.

*2) Stage 2: Re-Ranking:* However, we also noticed that a very large (around 10%) drop in coverage occurred in each case at the point at which we knocked out the gene corresponding to `numberOfCalls`. This finding cast doubt on the validity of the $bestMerit$ and $bestRank$ rankings. We therefore ranked the gene types according to $avgMerit$, and realized that `numberOfCalls` was the highest ranked, and that the $avgRank$ ranking largely agreed with this assessment. We therefore performed a progressive gene type knockout according to the avgMerit ranking instead, and tabulated the results.

In the following, each run was compared to the runtime and coverage seen using all ten gene types and running for $g = 50$ generations. Figure 10 shows how the coverage changed for one of the `java.util` classes; the results for this subject unit are typical. The axes of that figure is defined such that the point (50,1) represents the coverage of all 10 gene types after 50 generations. The thick black curve on those figures shows the performance of Nighthawk using all ten gene types. Other curves show results from using $1 \leq i \leq 9$ gene types.

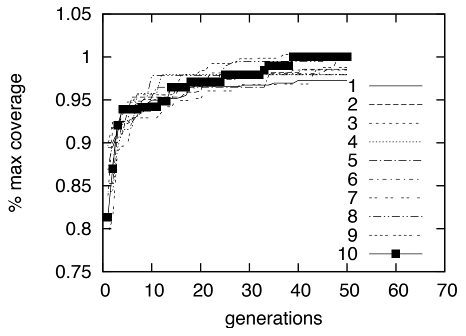

Fig. 10. Nighthawk on Hashtable unit, eliminating gene types according to $avgMerit$ ranking.

A measure of interest in Figure 10 is the area under the curve, which is maximal when Nighthawk converges to maximum coverage in a few generations. Due to the random nature of the GA and the randomized test data generation, some of the curves are sometimes higher than the $i = 10$ line.

If we calculate the ratio of the area under a curve (AUC) with the area under the thick black

| Number of used gene types | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | class being tested |
| 1.00 | 1.00 | 1.01 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | ArrayList |
| 1.13 | 1.00 | 1.03 | 0.96 | 0.65 | 0.68 | 0.67 | 0.88 | 0.71 | 1.00 | EnumMap |
| 0.98 | 0.99 | 0.98 | 0.96 | 1.01 | 1.00 | 1.00 | 1.00 | 0.98 | 1.00 | HashMap |
| 0.99 | 1.00 | 0.99 | 1.00 | 1.00 | 1.00 | 0.99 | 1.00 | 1.00 | 1.00 | HashSet |
| 0.99 | 1.00 | 1.00 | 1.00 | 0.99 | 1.00 | 0.98 | 1.01 | 1.01 | 1.00 | Hashtable |
| 0.97 | 0.97 | 0.98 | 0.97 | 0.98 | 0.97 | 1.00 | 0.98 | 0.98 | 1.00 | IdentityHashMap |
| 0.98 | 1.00 | 1.00 | 1.00 | 0.99 | 1.00 | 1.00 | 0.99 | 0.99 | 1.00 | LinkedHashMap |
| 0.99 | 1.01 | 1.01 | 1.00 | 1.00 | 1.00 | 1.00 | 1.01 | 1.01 | 1.00 | LinkedHashSet |
| 1.01 | 1.01 | 1.01 | 1.02 | 1.00 | 1.01 | 1.01 | 1.02 | 1.00 | 1.00 | LinkedList |
| 1.01 | 0.99 | 0.97 | 1.01 | 1.03 | 0.99 | 0.95 | 0.98 | 1.00 | 1.00 | PriorityQueue |
| 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.99 | 1.00 | Properties |
| 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | Stack |
| 0.90 | 0.97 | 0.95 | 0.93 | 0.99 | 0.97 | 0.97 | 1.02 | 1.01 | 1.00 | TreeMap |
| 0.90 | 0.95 | 0.98 | 0.93 | 0.97 | 0.98 | 0.98 | 1.00 | 0.98 | 1.00 | TreeSet |
| 0.95 | 0.97 | 0.99 | 0.96 | 0.99 | 0.92 | 0.99 | 0.97 | 1.01 | 1.00 | Vector |
| 0.96 | 0.97 | 0.97 | 0.98 | 0.98 | 0.97 | 0.98 | 0.99 | 0.97 | 1.00 | WeakHashMap |
| 0.98 | 0.99 | 0.99 | 0.98 | 0.97 | 0.97 | 0.97 | 0.99 | 0.98 | 1.00 | mean |

Fig. 11. Coverage found using the top $i$ ranked gene types for $1 \leq i \leq 10$, using the avgMerit ranking. Coverages are expressed as a ratio of the coverages found using all gene types.

curve, then we can summarize all the curves of Figure 10 as the Hashtable row of Figure 11. In that figure, each column shows how many gene types were used in a particular run (and when $used = 10$, we are ignoring the FSS results and using all gene types). The number 1.00 informs us that we achieved 100% of the coverage reached by using all ten gene types.

Figure 11 summarizes Figure 10 as well as results from all the other `java.util` classes. The last row of Figure 11 shows that the mean AUC is very similar using the top-ranked $i$ gene types. From this observation, we concluded that Nighthawk's GA gained most of its efficacy just for mutations of the top-ranked gene type `numberOfCalls`. However, a statistical comparison of the coverage measures with only the top gene type and those with all ten gene types does show a statistically significant difference (a

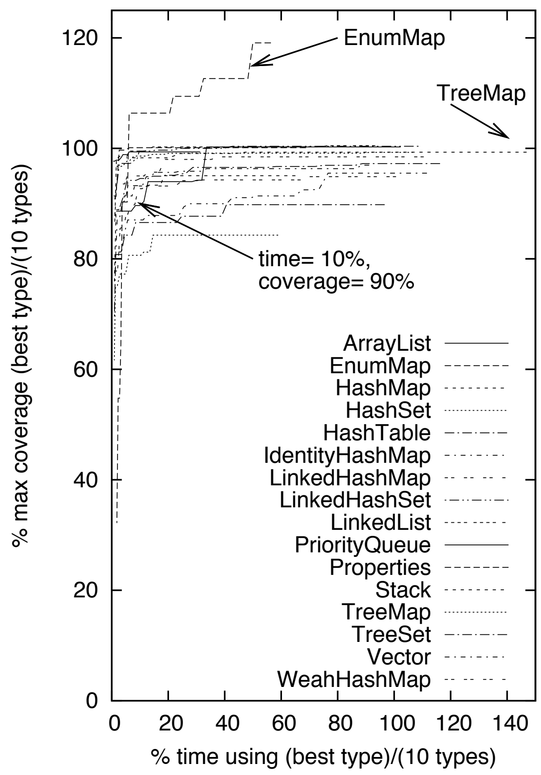

Fig. 12. Time vs coverage result, compared between one and ten gene types, for the Java util classes.

paired Wilcoxon test with $\alpha = 0.05$).

To check if eliminating gene types from Nighthawk's GA is cost-effective, we must consider both the coverage achievable and the time taken to achieve that coverage. We therefore made two runs of Nighthawk using (a) all the gene types and using (b) just the top gene type ranked by $avgMerit$ (i.e. `numberOfCalls`). We then divided the runtime and coverage results from (b) by the (a) values seen after 50 generations, and plotted the results.

Figure 12 shows the results, with time percentage on the X axis and coverage percentage on the Y axis. Note the point indicated by the arrow in Figure 12. This point shows that it is usually (in $\frac{13}{16}$ cases) possible to achieve 90% of the coverage in under 10% of the time required to run all gene types for 50 generations. The data for the two outlier subject units in Figure 12, `EnumMap` and `TreeMap`, can be attributed to the low coverage achieved by the original Nighthawk on `EnumMap` and the stochastic nature of both the GA and random testing level.

The results of this progressive gene type knockout procedure corroborated the rankings that we had derived using the $avgMerit$ and $avgRank$ measures, lending support to the validity of these measures as opposed to the $bestMerit$ and $bestRank$ measures. We therefore continued to use only $avgMerit$ and $avgRank$ in our subsequent research.

*3) Stage 3: Optimizing* `numberOfCalls`*:* The implication of the result regarding the `numberOfCalls` gene type was that changing the number of method calls made in the test case was most important to reach higher coverage. What was the cause of this importance? If it was due simply to the fact that we had chosen a sub-optimal initial value for the number of calls, then changing the initial value could result in a quicker convergence to an optimal value. Moreover, a sub-optimal initial value for `numberOfCalls` could have the effect of skewing our FSS analysis, since `numberOfCalls` could assume a skewed importance due to the need to change it.

Nighthawk assigns an initial value of $k \cdot |C_M|$ to `numberOfCalls`, where $k$ is a constant and $C_M$ is the set of callable methods. The motivation for this initial value is that if there are more callable methods associated with the unit under test, then we expect to have to make more method calls in order to test them. In version 1.0 of Nighthawk, we set $k$ to 5, based on our earlier experience with randomized testing.

To investigate whether this initial value of `numberOfCalls` was optimal, we generated a scatter plot (Figure 13) plotting the number of callable methods against the final value of `numberOfCalls` for the winning chromosome found by Nighthawk for each of our subject units. From this scatter plot, it became obvious that the initial value was indeed sub-optimal: a value of $5$ for $k$ results in the lower line in the figure, which is below the final value for all but one of the subject units. A value of $50$ for $k$ results in the upper line in the figure, which is much closer to the final value for all units.

Based on this analysis, we changed the value of $k$ to 50; that is, we set the initial value of `numberOfCalls` to 50 times the number of callable methods. We refer to Nighthawk with this change as Nighthawk version 1.1, since it exhibits quite different behavior from Nighthawk 1.0.

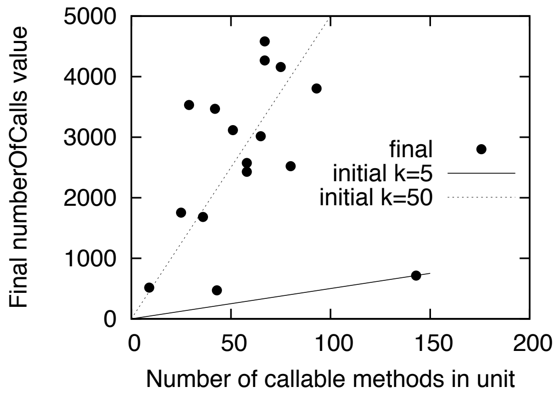

Fig. 13. Finding the optimal initial number of calls.

*4) Stage 4: Analyzing the Optimized Version:* Because of our concern that the sub-optimal initial `numberOfCalls` value had skewed our previous analysis, we re-ran the merit analysis using Nighthawk 1.1 and the `java.util` classes.

To corroborate our conclusions, we ran a merit analysis using Nighthawk 1.1 and a new set of classes. The Apache Commons Collection classes is a set of classes implementing efficient data structures, developed by the open-source Apache development community. We focused on the 36 concrete top-level classes in this effort. Two of these classes (`ExtendedProperties` and `MapUtils`) tended to generate large numbers of files when Nighthawk was run on them; we could have dealt with this by modifying the appropriate test wrappers, but for greater reproducibility we decided to omit them. We therefore ran a merit analysis using Nighthawk 1.1 and the other 34 concrete top-level classes.

We then performed a gene type ranking using only the $avgMerit$ and $avgRank$ measures. The results of this ranking are in Figure 14. Note that while `numberOfCalls` is still ranked highly, now the gene type `chanceOfTrue` emerges as the most influential gene type, with `upperBound` and `valuePoolActivityBitSet` also being influential for some classes. However, as noted even for Nighthawk 1.0 using the $avgMerit$ and $avgRank$ rankings, the gene types `candidateBitSet` and `chanceOfNull` were consistently not useful.

A progressive gene type knockout based on the $avgMerit$ gene type ranking supported the validity of that ranking: knocking out the lowest-ranked gene types had almost no effect on

| | Rank of gene type when ranked by measure | | | |
|---|---|---|---|---|
| | `java.util` | | Apache | |
| Gene type | *avgMerit* | *avgRank* | *avgMerit* | *avgRank* |
| `numberOfCalls` | 3 | 3 | 1 | 2 |
| `valuePoolActivityBitSet` | 4 | 5 | 2 | 3 |
| `chanceOfTrue` | 1 | 1 | 3 | 1 |
| `numberOfValuePools` | 7 | 7 | 4 | 4 |
| `methodWeight` | 9 | 9 | 5 | 8 |
| `numberOfValues` | 6 | 6 | 6 | 7 |
| `upperBound` | 2 | 2 | 7 | 5 |
| `lowerBound` | 5 | 4 | 8 | 6 |
| `chanceOfNull` | 8 | 8 | 9 | 9 |
| `candidateBitSet` | 10 | 10 | 10 | 10 |

Fig. 14. Ranks of all gene types according to average merit and average gene rank, after running Nighthawk 1.1 on the `java.util` classes and the Apache Commons classes.

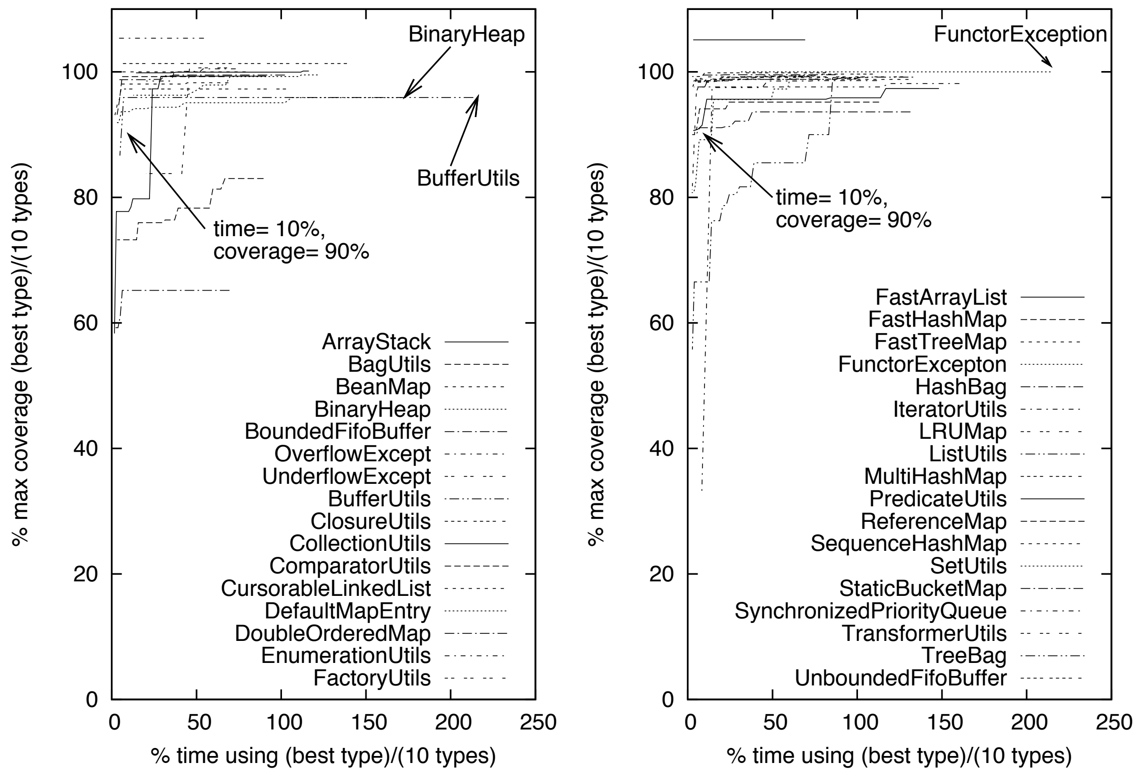

Fig. 15. Time vs coverage result, compared between one and ten gene types, for the Apache classes.

eventual coverage, while knocking out the higher-ranked gene types had a small but more noticeable effect.

Furthermore, as shown in Figure 15, it is still the case that we can achieve 90% of the coverage achieved by all 10 gene types in 10% of the time if we use only the top-ranked gene type and stop the genetic algorithm early. The Figure 15 results were generated in the same manner as above (in Figure 12):

- We made two runs using (a) all the gene types and using (b) just the top gene type ranked by $avgMerit$ (i.e. `numberOfCalls`).
- We then divided the runtime and coverage results from (b) by the (a) values seen after 50 generations, and plotted the results in Figure 15
- The points indicated by the two arrow in Figure 15 shows that it is usually (in $\frac{28}{34}$ cases) possible to achieve 90% of the coverage in under 10% of the time required to run all gene types for 50 generations.
- There exist some outlier results (*BinaryHeap, BufferUtils, FunctorException*) where the run-times were much longer for the Nighthawk using only one operator than for full Nighthawk. Note that, in all those exception cases, the Nighthawk optimized by FSS still achieved 90% of the coverage in 10% of the time required of full Nighthawk.

## E. Discussion

There are two main areas of implications for this work: implications for Nighthawk itself, and implications for other systems.

*1) Implications for Nighthawk:* On the surface, the results reported above suggest that most gene types can be eliminated. However, there are at least two mitigating factors. First, the additional coverage may be of code that is difficult to cover, and thus this additional coverage might be more valuable to the user than the raw coverage numbers suggest. Second, the observations about gene types might not carry over to other, different subject units.

Nevertheless, the results show that it is very likely that Nighthawk can be modified to give users a better range of cost-benefit tradeoffs, for instance by eliminating gene types or using early stopping criteria that take advantage of the early plateaus in coverage seen in Figure 12. In particular, the results suggest that the gene types `candidateBitSet` and `chanceOfNull` were not useful. Translating this back into the terms of Nighthawk's randomized test input generation algorithm, this means that when it is looking for a value to pass to a parameter of class C, it should always draw parameter values from value pools in all the subclasses of C (the default value of `candidateBitSet`); furthermore, it is sufficient for it to choose `null` as a parameter 3% of the time (the default value of `chanceOfNull`).

*2) Implications for Other Systems:* At the *meta-heuristic level*, this work suggests that it may be useful to integrate FSS directly into meta-heuristic algorithms. Such an integration would enable the automatic reporting of the merits of individual features, and the automatic or semi-automatic selection of features. If the results of this paper extend to other domains, this would lead to meta-heuristic algorithms that improve themselves automatically each time they are run.

Also, on the *theory-formation level*, this work opens up the possibility of rapid turnover of the theoretical foundations underlying present tools, as aspects of heuristic and meta-heuristic approaches are shown to be consistently valuable or expendable. The expendability of Nighthawk's `candidateBitSet` and `chanceOfNull` gene types is an example of this latter phenomenon.

## V. Threats to Validity

The representativeness of the units under test is the major threat to external validity. We studied Java collection classes and the Apache 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.

Also, time measurement is a construct validity threat. We measured time using Java's `systemTimeInMillis`, which reports total wall clock time, not CPU time. This may show run times that do not reflect the testing cost to a real user.

## VI. 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 optimize and simplify meta-heuristic search tools. Metaheuristic tools (such as genetic algorithms and simulated annealers) typically mutate some aspect of a candidate solution and evaluate the results. If the effect of mutating each aspect is recorded, then each aspect can be considered a feature and is amenable to the FSS processing described here. In this way, FSS can be used to automatically find and remove superfluous parts of the search control.

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. Further, we can see a promising line of research where the cost/benefits of a particular meta-heuristic are tuned to the particulars of a specific problem. Here, we have shown that if we surrender $\frac{1}{10}$-th of the coverage, we can run Nighthawk ten times faster. While this is an acceptable trade-off in many domains, it may unsuitable for safety critical applications. More work is required to understand how to best match meta-heuristics (with or without FSS) to particular problem domains.

## 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] 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.

[11] J. Andrews, F. Li, and T. Menzies, "Nighthawk: A two-level genetic-random unit test data generator," in *IEEE ASE'07*, 2007, available from http://menzies.us/pdf/07ase-nighthawk.pdf.

[12] J. Andrews and T. Menzies, "On the value of combining feature subset selection with genetic algorithms: Faster learning of coverage models," in *PROMISE 2009*, 2009, available from http://menzies.us/pdf/09fssga.pdf.

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, August 1993.

[37] A. S. Namin and J. Andrews, "The influence of size and coverage on test suite effectiveness," in *International Symposium on Software Testing and Analysis (ISSTA'09)*, Chicago, IL, USA, 2009, pp. 57–68.

[38] 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.

[39] D. R. Cok, "Adapting jml to generic types and java 1.6," in *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, November 2008, pp. 27–34.

[40] 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.

[41] 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.

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

[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] 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.

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

[46] 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.

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

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

[49] S. Berner, R. Weber, and R. K. Keller, "Enhancing software testing by judicious use of code coverage information," in *29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA, May 2007, pp. 612–620.