2008

Automatically Generating Random Test Data for Relevant and Implicitly Defined Subdomains

John Alexander Murphy
College of William & Mary - Arts & Sciences

AUTOMATICALLY GENERATING RANDOM TEST DATA FOR RELEVANT AND IMPLICITLY DEFINED SUBDOMAINS

John Alexander Murphy

Oakton, VA

Bachelor of Science, College of William and Mary 2005

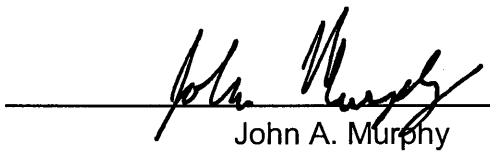
A Thesis presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Master of Science

Department of Computer Science

The College of William and Mary
January 2008

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

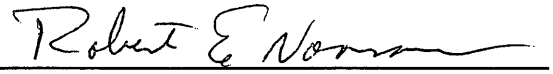


John A. Murphy

Approved by the Committee, January 2008

Committee Chair
Assistant Prof. David Coppit, Computer Science
College of William and Mary

Associate Prof. Peter Kemper,Computer Science
College of William and Mary



Prof. Robert Noonan, Computer Science
College of William and Mary

ABSTRACT PAGE

Many methods have been proposed to evaluate the correctness of software. One such strategy is random testing, in which inputs are randomly selected or generated from the entire input space of a method. In many cases, random testing is desirable because it is a highly automatable process, relieving the tester of the tedious task of generating test cases by hand. However, in the case where the input space is sufficiently complex or left undefined prior to testing, three difficulties arise: One: It may become prohibitively difficult to ensure that all inputs generated are in fact valid inputs to the software under test. Two: Within the valid input space of the software under test, not all tests are potentially error-revealing. Restricting the search for errors to the relevant subdomain of valid and potentially error-revealing inputs further complicates the test generation strategy. And three: It may become prohibitively difficult to ensure that the test case generation is truly random; that is, a uniformly distributed selection among all inputs in the relevant subdomain with no bias towards specific regions of the input space.

To that end, this work presents a method which expands the scope of scenarios in which random testing is feasible. First, a novel algorithm is presented for the random selection of relevant test cases in n-dimensional real space in which the relevant subdomain is orders of magnitude smaller than the valid input space and potentially involves arbitrarily complex interactions between constraints involving multiple variables. A testing framework integrating this algorithm with more traditional automated test generation strategies is also presented and is used in a case study to evaluate the correctness of an implementation of the KB3D aircraft collision avoidance algorithm. The objectives of this strategy include ensuring an approximately uniform distribution of test cases across the valid and relevant subdomain of the input space as well as minimizing the amount of time spent examining irrelevant test cases. The case study will be evaluated with these criteria along with the ability to discover faults in the software itself.

To my family.

Table of Contents

ACKNOWLEDGMENTS

This work would not be what it is today without the help and guidance of many people. First I would like to express my appreciation for my advisor, Dr. David Coppit, for both his high-level guidance in finding a problem needing solving, helping with algorithm design, and participating in the gruntwork of implementation and debugging. I would like to thank César Muñoz for serving as my domain expert in the KB3D case study and explaining its secrets. I would also like to thank my committee including Dr. Robert Noonan and Dr. Peter Kemper, whose comments and criticisms made this work better and more complete. My fellow research group consisting of Meghan Revelle and Brian Meckstroth were integral in listening to ideas of varying quality and providing moral support. I would also like to thank all who attended my practice talks for being a great audience and confidence booster. Finally, I would like to thank my father for his valuable comments on my thesis and for explaining the Metropolis-Hastings algorithm over the phone the night before my thesis defense.

List of Tables

List of Figures

AUTOMATICALLY GENERATING RANDOM TEST DATA FOR RELEVANT AND IMPLICITLY DEFINED SUBDOMAINS

Chapter 1

Introduction

1.1 Software Verification

One of the hurdles to producing quality software is establishing confidence in the correctness of the implementation. There are several methods of establishing such confidence. For critical systems in which lives or other valuables are at stake a mathematical proof of correctness may be required to be sufficiently confident in the system. In practice, very few developers actually prove their software to be correct and instead choose to follow the less rigorous route of testing software on a representative set of inputs. Proponents of software proofs such as Edsger Dijkstra warn that "Program testing can be used to find the presence of bugs, but never to show their absence!" [12]. Indeed, if one is to treat the software under test as a black box, short of exhaustively executing all possible inputs (an infeasible task for all but the smallest of input spaces) it is impossible to be supremely confident in the correctness of the software through testing alone. Instead, a tester will make some generalizing assumptions that the tests selected are indeed representative of the entire input

space. The tester simplifies the problem of establishing confidence at the expense of rigor.

There are several angles of attack one may follow to establish confidence via software testing. One angle involves examining the structure of the software under test and fabricating inputs which thoroughly exercise the software. White box testing such as this may have multiple objectives in its test selection criteria. One objective may be to test as much of the software under test as possible by one of several metrics such as statement, condition, or path coverage. Another objective may be to determine "likely" errors and craft inputs to test for them. Boundary conditions, off by one errors, null pointers, etc. are all common programming errors which a white box testing strategy may identify and test.

Rather than using the structure of the software as a guide for testing, one may also use the usage patterns of users of the software to construct a test suite and establish confidence in the software. By examining the typical usage patterns of users of the software, one may determine the distribution of inputs presented to the software as modeled by a random variable. By generating inputs according to this random variable, one may establish a statistical confidence in the correctness of the software under typical use over a period of time. However, in practice it is unlikely to have an accurate usage profile of software—especially software currently under development—so a common assumption is made by assuming all inputs are equally likely to occur. This work builds upon this testing strategy of uniform selections from an input space and expands upon it by expanding the set of spaces from which one may sample from a uniform distribution.

1.2 Difficulties of Random Testing

As useful as random testing can be, there are some difficulties. While the generation of pure random "fuzz" data is trivial, generating structured random inputs can be much more difficult. There may exist dependencies between variables in an input that one must take into account. While there exists some work in generating tests with dependent variables, some (like Java Pathfinder [48]) depend on linear equation solvers to generate inputs and fail when the dependencies are nonlinear. Others (like Directed Automated Random Testing [15]) degrade more elegantly on nonlinear constraints by regressing from symbolic to concrete execution when such a constraint is encountered.

However, in many cases the structure of the input is ill-defined, or perhaps the tester wishes to test some subdomain of which he or she has only some vague concept. Perhaps the tester knows which type of behavior he or she wants to elicit, but cannot determine a strategy to generate such inputs (or maybe the tester simply wishes to visualize the space of all inputs eliciting such a behavior). In such a case, it is impossible to create a uniformly distributed test selection strategy since the tester does not know the distribution from which he or she is selecting. This work presents a method for the simultaneous modeling and uniformly distributed sampling of a space which is largely unknown prior to sampling, and a testing framework combining this novel algorithm with more traditional random testing strategies.

In testing, we make a distinction between the *domain* and the *relevant subdomain.* We consider the *domain* to be the set of all possible syntactically valid inputs (typically the type-complete cross product of all input parameters) since even nonsensical or poorly

structured inputs are still possible inputs and must be properly accounted for. We consider the *relevant subdomain* to describe a class of inputs eliciting similar behavior (typically passing some set of preconditions). A relevant subdomain is always a subset of the domain; however, one unit under test may have multiple relevant subdomains depending on what behavior the tester wishes to test. In addition, two relevant subdomains of a domain need not be disjoint sets.

This work also uses the term *implicitly defined subdomain* to describe any class of inputs which elicits a specific behavior when used as inputs to a program but are not formally defined in any documentation. The implicitly defined subdomain is (as its name would suggest) defined implicitly by the behavior of the software itself. If an input elicits the behavior, it is by definition within the implicitly defined subdomain. Executing an input to observe its behavior is the only way to determine if an input is within the subdomain. While the logic behind the implicitly defined subdomain is indeed circular, it is also useful in that it is often much simpler to verify a solution rather than to generate one. This work leverages this disparity to elevate the importance of recognizing valid inputs rather than solving for new inputs from scratch by permuting already known valid inputs to other probable valid inputs. The novel algorithm presented in this work is the Implicit Subdomain Exploration (ISE) algorithm, which is designed to sample from an approximately uniform distribution from an arbitrary space whose shape and dimensions may be unknown prior to testing. This algorithm is used in a new framework combining the ISE algorithm with other more traditional automated test generation strategies and is used to evaluate the correctness of real-world software.

The remainder of this work is organized as follows: Chapter 2 provides some examples

in which random testing may be improved. Chapter 3 discusses other attempts to improve the effectiveness and feasibility of random testing. Chapter 4 discusses the ISE algorithm in detail. Chapter 5 describes the architecture of the testing framework built around the ISE algorithm. Chapter 6 evaluates the ISE algorithm in several synthetic but representative examples as well as presents a case study where the ISE algorithm was used to test real-world software. Chapter 7 concludes.

Chapter 2

Motivation

While one of the biggest obstacles to random test generation is the creation of an oracle to verify the correctness of the software for an arbitrary input, an often ignored difficulty is in the generation of inputs, especially if one wishes for the inputs to have specific qualities. Pure fuzz testing has been shown to be surprisingly effective in discovering faults in software [37], however to be confident in the correctness of software one must also test on inputs likely to be presented in actual use of the software. This work has previously discussed the difficulties in generating inputs which lie within the 2-dimensional unit circle, an input space with a very basic structure. As complexity of the structure of the input space increases, so does the difficulty in crafting an unbiased test generation strategy. Often, the structure of the input is not explicitly defined, compounding the difficulty in creating a test strategy. Or perhaps the tester has identified a subdomain of the input space which requires more extensive testing. For example, one implementation of a square root method may read as follows:

```
double sqrt(double x)
{
  if(x < 0)
    return NaN;
  else
  {
    ...
    sqrtOfX = ...
    return sqrtOfX;
  }
}
```

The *domain* of this method is the set of all *double*s. However, the tester will probably identify all values of x less than zero as trivial, perhaps only requiring a single test case to ensure that the method correctly identifies negative inputs. The vast majority of the logic is only exercised for nonnegative inputs, therefore the tester will most likely focus the majority of his or her efforts on the *relevant subdomain* of nonnegative *double*s.

In the previous scenario, once the tester identified the relevant subdomain devising a generation strategy was a relatively simple matter. Generating a single value selected uniformly from a range is exactly what random number generators do best. Matters get more complicated when tests use multiple dependent input values. For example, consider a piece of software whose relevant subdomain is the set of all two-dimensional points lying within the unit circle centered on the origin. This is a relatively simple space to explore, but one must take into account that a value chosen for the x component affects the possible y component (and vice versa) which requires a more complicated generation strategy.

When testing this software, one has two goals to achieve: (1) only test inputs within the domain, as any other test would be a wasted effort, and (2) uniformly sample from the domain, as with no prior knowledge of the usage profile of the software, one must assume

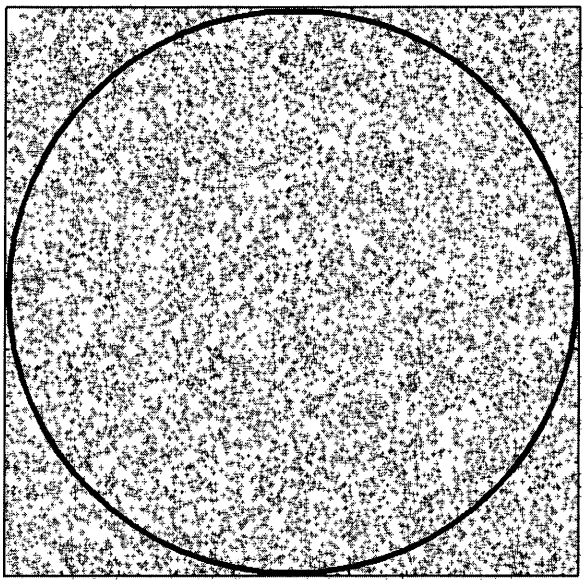


Figure 2.1: 10,000 tests generated using a Monte Carlo test selection strategy

all valid inputs are equally likely to occur in practice.

The first problem the tester may encounter is that of selecting a region too large, encompassing both relevant and irrelevant inputs. Testing an irrelevant input is wasted testing effort. For example, one's first attempt at sampling the unit circle space might be a simple Monte Carlo [35] rejection sampling method. All points within the unit circle will lie within the bounding box defined by the points $(-1, -1)$ and $(1, 1)$, so sampling will consist of two selections from a uniformly distributed random variable on the interval $[-1, 1]$ with one selection determining the x coordinate and the other determining the y coordinate. This will guarantee a uniform distribution across the input space. However, as can be seen in Figure 2.1, the bounding box has a larger area than the domain of the software under test. This means that some of the generated test cases will lie outside the domain and testing effort is wasted. In this case, the amount of wasted effort is $1 - \frac{\pi}{4}$, or approximately 21% of

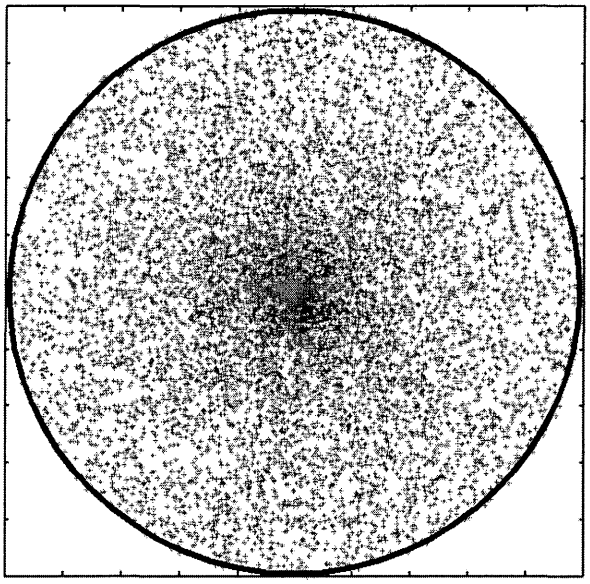


Figure 2.2: 10,000 tests generated using a biased test selection strategy

all tests are irrelevant. Depending on the shape of the input space, the portion of irrelevant inputs may become arbitrarily high and defining a bounding box and sampling from within that space becomes infeasible.

Further refinements may rectify this by using a more appropriate selection strategy. However, the tester faces a second problem: ensuring tests are sampled from a uniform distribution. This is important for two reasons. One, the tester may wish to make statistical arguments as to the reliability of the software, which requires an intelligently selected distribution. Two, an unintentionally biased sampling method may waste testing effort in one region by overtesting, to the detriment of other, relatively undertested regions.

In the case of the unit circle, one may instead choose an angle θ from the interval $[0, 2\pi)$ and a distance from the origin r from the interval $[0, 1]$. This will define a point (θ, r) in polar coordinates within the unit circle. In this strategy, one is guaranteed that

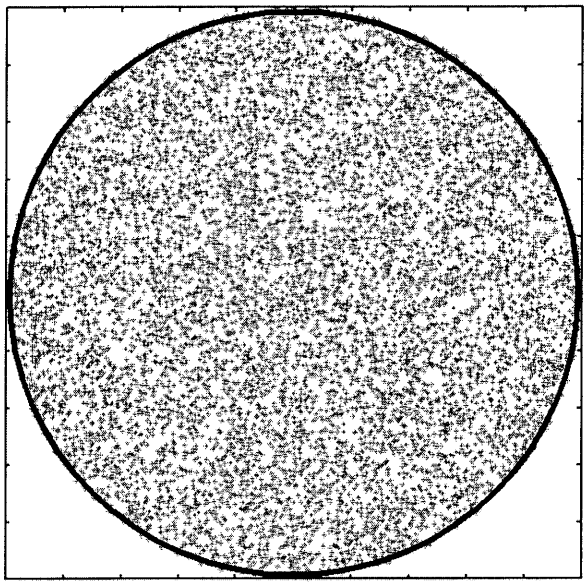


Figure 2.3: 10,000 tests generated using an unbiased test selection strategy

all generated tests will indeed lie within the domain of the space, and all valid tests are possible to generate. However, Figure 2.2 shows that this selection strategy is biased to generate inputs near the origin. This goes against one's original assumption that with no prior knowledge of the software under test, one assumes a uniform usage profile and that all inputs are equally likely to reveal a fault. By biasing the testing towards the origin at the expense of the boundaries of the space, any confidence metrics used with the assumption of a uniform usage profile must immediately be called into question.

One final refinement of the selection strategy allows one to have an unbiased selection without wasting any effort testing outside the domain of the software under test. As in the previous example, an angle θ is chosen from the interval $[0, 2\pi)$ and a distance from the origin r is selected from the interval $[0, 1]$. However, the polar coordinates defining the test are taken to be $(\theta, \sqrt{r})$ rather than (θ, r). This removes the bias introduced due to the fact

that the circumference of a circle grows at the square of the radius. The previous solution would over time place the same number of tests "near" the origin as "far" from the origin without accounting for the quadratic growth in area the further one travels from the origin, which skewed test selection. An example of tests generated with this proper strategy can be seen in Figure 2.3.

Even in the seemingly trivial input space of the unit circle, determining a test selection strategy which wastes little testing effort while still selecting tests from a uniform distribution is a nontrivial task. Indeed, in this case the tester has complete knowledge of the input space of the software under test. In general, one cannot assume that the tester has such knowledge. Whether due to poor or no specifications, the tester may find the input space is only *implicitly defined* by the behavior of the software itself. In cases such as this, it is exceedingly difficult to obtain confidence metrics for the software under test as the tester does not even know the set of all of the possible inputs.

In Chapter 6, this work will evaluate testing strategies on the highly complex input space of the KB3D [39] software. KB3D is an aircraft conflict detection and resolution algorithm developed at the National Institute of Aerospace. The algorithm takes a pair of aircraft as input and determines if the two are on a "near collision" course. If it determines the aircraft will indeed violate the minimum separation required by the aircraft, the algorithm will also provide a set of new headings for the aircraft to follow to resolve the conflict. In this instance, the domain of the software is the set of all pairs of aircraft (where each aircraft is defined by a three dimensional location along with a three dimensional velocity vector). However, to thoroughly exercise the portion of the algorithm which generates resolution headings, one must look at the relevant subdomain of all pairs of aircraft *on a*

near-collision course. This subdomain is orders of magnitude smaller than the true domain, however it is also much more difficult to devise a testing strategy which only generates pairs of aircraft meeting the required precondition. One could generate pairs of aircraft within a certain distance of each other, but this is not ideal as this distance would be an arbitrary decision. Depending on the velocity and heading of the aircraft, the radius might be too large and capture too many pairs of aircraft not on a collision course, or too small and not capture the entire subdomain. There is simply no obvious strategy for the generation of inputs in this subdomain. This relevant subdomain and others like it are the primary motivation behind the development of the Implicit Subdomain Exploration algorithm.

Chapter 3

Related Work

3.1 The Random Testing Controversy

Random testing has often been criticized as a poor testing tool in comparison to nearly all other testing strategies. Ince states that random testing "is the default case by which other methods should be judged" and that "random testing seems to be the worst possible way of testing software" [24].

The crux of the argument against random testing stems from the fact that without using any information about the software under test, the testing is by definition unguided towards the detection of faults. Another issue with automated random testing includes the requirement of an oracle to verify the results of an arbitrary test. The confidence gained through testing hinges entirely on one's confidence in the oracle. While these criticisms are entirely valid, random testing has several very desirable qualities as well. Richard Hamlet notes the two primary reasons random testing is still pursued: (1)"...there are efficient methods of selecting random points algorithmically...thus a vast number of tests

can be easily defined" and (2)"statistical independence among test points allows statistical prediction of significance in the observed results." [19]

Several studies have been performed analyzing the effectiveness of random testing to that of "partition testing," or the subdivision of the input space into subdomains defined "whose points are somehow 'the same,' "[17] where 'sameness' can be based on "requirements or specifications,...features of the code,...the process by which the software was developed, or on the suspicions and fears of the programmer." Since all points in a partition are considered equivalent, only one representative test must be executed from each partition. The quality of partition testing therefore depends entirely on the quality of the partitions defined by the tester. Weyuker and Jeng performed a theoretical analysis and came to the somewhat surprising conclusion that partition testing can be either more or less effective than random testing depending on the distribution of faults within a subdomain [50]. Ideally, a subdomain would be *homogeneous*, that is, if one test selected from that subdomain is fault-revealing, all tests from that subdomain are guaranteed to reveal the same fault. In practice, subdomains are not necessarily homogeneous nor are partitions disjoint, which affects the efficiency of partition testing. Theoretical studies such as those of Duran and Ntafos [14] and Hamlet and Taylor [17] came to similar conclusions.

In fact, an entire area of testing called "fuzz testing" has risen which takes unstructured random testing to its extreme by presenting completely unstructured random data to a program with the intent of discovering defects. The first study of Fuzz testing was performed by Miller et al. [37] and tested various UNIX utilities when presented with unstructured random inputs. The study came to a very optimistic conclusion as to the effectiveness of fuzz testing (or a very pessimistic conclusion as to the quality of software in general, depending

on one's outlook). In this study, roughly a quarter of tested UNIX utilities were forced into either a hang or a crash when presented with random inputs. Fuzz testing requires no structured input generation and even may not require an oracle (as in this study, in which only crashes and hangs were detected). While fuzz testing is far from a complete evaluation of the correctness of software, the ease of random testing combined with its ability to capture "low hanging fruit" which apparently persists even in mature software is encouraging for random testing's utility as a useful testing strategy.

Bird and Munoz performed another early study on the utility of random testing [3]. Where fuzz testing is the application of completely unstructured inputs to a piece of software, Bird and Munoz created custom generators to create inputs to software. Bird and Munoz discuss the use random testing to generate tests for a compiler (using a grammar syntax generator, a concept first used by K. V. Hanford to generate tests for the PL/I programming language [20]), a graphical display manager, and a sorting algorithm. While the authors found many benefits to automated random testing—namely, its time-saving aspects when compared to manual test creation and its effectiveness in practice—it still required the design and implementation of a new input generator for each application. A general input generator for use on arbitrary software would be a significant boon to the automated testing cause.

3.2 Attempts to Improve Automated Test Generation

While random testing alone has shown mixed success, random generation in concert with a directed search has been an area of much recent research. One such approach has been

named hybrid concolic testing [30] after its combination of random, concrete testing alongside a symbolic execution of the same code. This hybrid approach allows the deep coverage one can get with symbolic execution of software while still being able to fall back on a concrete execution when symbolic execution becomes impossible, whether due to path conditions too complex to be automatically solved, or if the source code to some library functions is unavailable making symbolic execution impossible. Hybrid concolic testing has been implemented in the form of Directed Automated Random Testing (DART) by Godefroid, Klarlund, and Sen [15]. In that work, DART was used to test an implementation of the Needham-Schroder public key authentication protocol as well as an implementation of the Session Initiation Protocol (SIP), with encouraging results.

Korat is a similar tool for the generation of objects of complex structure for testing purposes [36, 4]. Korat uses the Java Modeling Language (JML) [28] to specify the structure of an object and systematically iterate through valid objects. This systematic iteration is sometimes referred to as *Bounded Exhaustive Testing*, or BET [46], or the testing of all possible inputs up to some bound on size. Studies have shown that exhaustively testing all inputs up to a small bound on size can be an effective method of detecting faults in software [46, 32]. Korat utilizes some novel heuristics to prune the search space to ensure that only valid objects are created as well as to ensure that all created objects are non-isomorphic; that is, an unordered tree consisting of a root node, a left leaf, and a right leaf is isomorphic to a tree with the same root but the two leaves transposed, and therefore only one of the two would be used for testing purposes. TestEra is a similar tool [33], but uses Alloy [25] rather than JML as its modeling language. Abdurazik and Offutt propose a method to automatically generate tests based on Unified Modeling Language (UML)

[1]. Windbladh et al. automate test generation using GoalML [51]. While the variety of specification languages in use can be both a hindrance and a curse, automated specification-based testing (creating tests based solely on the specification of the software rather than testing for a specific implementation) has been an area of much research.

In recent years, the concept of Search Based Software Engineering (SBSE) has been presented by some as a useful paradigm for software engineering in general and software testing in particular [21, 22]. SBSE is based on the realization that many problems in the field of software engineering (from estimating costs and timelines [2] to minimizing coupling between modules in a system [31]) can be formed as search or optimization problems—a class of problems with a preexisting body of mature algorithms from the field of operations research.

Perhaps nowhere else in the field of software engineering is the analogy to search problems more clear than in software testing, which can be classified as the search for faults in software. Harman clarifies the reformulation from testing to searching as follows: "...[T]he set of all possible inputs to the program forms a search space and the test adequacy criterion is coded as a fitness function." [21] This is a versatile and useful definition of testing as no matter what test adequacy criterion is used (whether it be achieving high code coverage, closely matching a usage profile distribution, etc.), if it can be translated into a fitness function it might be beneficial to view the test generation as a search problem. Examples of search based software testing include the use of genetic algorithms to generate inputs for an autonomous automobile parking system [49]. Genetic and hill-climbing algorithms have been used to prioritize regression tests and minimize test suites [10] (useful for when automated test generation is likely to produce many isomorphically equivalent inputs).

3.3 Black-box Sampling Strategies

Where tools like DART, Korat, and TestEra require either wholly or in part the use of a constraint solver with access to the source code to define and explore a space, there also exist other, more lightweight approaches which—while not using as much knowledge about the software under test—can still show improvements when compared to random testing. Pacheco et al. [43] use a black box strategy in their work to create the tool *RANDOOP* (*RAND*om tester for *O*bject-*O*riented *P*rograms). RANDOOP allows for the random creation of objects of complex structure by using feedback from previously generated tests. Starting with the simplest object possible, RANDOOP randomly calls allowable methods on the object to "grow" it. After each successful method call, the post-call object is added to a pool of valid objects, all of which can be selected for further alteration. Unsuccessful method calls (for example, calling *pop()* on an empty stack) are discarded, helping to prune the search space. This can be considered a sort of local search in that atomic changes are applied to known objects to expand the set of known, valid objects. A similar approach is used in this work to search and discover a region of connected and valid inputs to software. While Pacheco's work focuses mainly on object-oriented testing, the work presented here is based on functional testing with inputs of simple structure but with complex semantics applied to them.

Another blackbox testing approach is called Adaptive Random Testing (ART, not to be confused with the previously mentioned DART) [8, 6]. ART is a refinement of random testing to achieve wide coverage of the input space faster than a pure random approach. Chan et al. discuss several distinct subtypes of ART, including Restricted Random Testing

in which a zone of restriction is placed around previously generated inputs in an attempt to spread out future test inputs and Mirrored Random Testing, in which the domain is broken into disjoint subdomains from which analogous tests in each subdomain are selected. While ART showed promise in simulation, as far as the author is aware these methods of ART have not been applied in any actual case studies. Instead, simulations were performed on 2-dimensional and 3-dimensional rectangular spaces, where detecting faults were simulated by determining relatively small circular (or spherical, depending on the dimensionality) regions of failure. Generating an input from this error region indicated a successful error detection. One might criticize these simulations as being non-representative of real-world software input domains and error distributions.

The work presented here and in a previous publication [40] is designed similarly to improve upon random testing by examining previously generated inputs to explore an entire space quickly and uniformly, but also will attempt to discover and model input spaces of unknown size and shape. By defining the "search" as a search for both faults in the software as well as a search for the input space itself, the methods presented in this work can be (and have been) applied to real-world software of nontrivial complexity impossible with previous blackbox testing strategies such as ART.

Chapter 4

The Implicit Subdomain Exploration Algorithm

4.1 Objectives

The overarching goal of this work is to present a method for the sampling from a uniform distribution of an unknown space. To do so, the Implicit Subdomain Exploration (ISE) algorithm must build an internal model of the space and refine the model over time as more details about the space are learned. One of the primary goals of the algorithm is to develop an accurate model of the space in as few experimental tests as possible, since the sample cannot be considered uniform if portions of the space are left undiscovered. Second, once an accurate model has been built, the algorithm should bias the sampling towards relatively unsampled regions until the global distribution is approximately uniform, at which point future samples should maintain the uniform distribution until the user-defined number of tests has been generated. Third, minimal testing effort must be wasted on exploratory

sampling outside of the space under test. Finally, running time must be "reasonable," to the point where the time required to generate a test is no longer than the time required to execute a test. This also means that the running time of the algorithm must be $O(n)$ with regard to the number of tests generated; that is, the marginal cost of generating one additional test can not increase as more and more tests are generated.

4.2 Assumptions

The ISE algorithm must make several assumptions as to the nature of the space under test. The first assumption is that the ISE algorithm requires the tester to provide some method of determining if an executed test case lies in the relevant subdomain. However, the ISE algorithm allows much leeway in how this determination is implemented. If the software is properly annotated in a modeling language such as JML, the tester can use the already defined preconditions to determine relevance. If the software is not modeled in this manner (or if the tester wishes to test some emergent behavior not defined in the specifications) the tester may implement any method of his or her choice to determine relevance taking an input vector as its parameter and returning a boolean value of *true* for relevant inputs and *false* for irrelevant inputs.

Second, the ISE algorithm requires the tester to provide one valid test input as a "seed" to start the search. Initial samples will be biased towards tests geometrically "close" to this input, however, as the model of the relevant subdomain is refined the ISE algorithm will compensate and over time the distribution will approximate a uniform sampling of the entire region.

Third, the algorithm requires the input space to consist of one connected region; that is, to get from one relevant input vector v to any other relevant input vector v' in the relevant subdomain D, there must exist some list of vectors V of arbitrarily small (but nonzero) magnitude for which the following holds true:

$$v + \sum_{i=1}^{|V|} V_{[i]} \quad = \quad v' \tag{4.1}$$

and

$$\forall c : 1 \leq c \leq |V| : v + \sum_{i=1}^{c} V_{[i]} \in D \tag{4.2}$$

That is, the list of vectors V must have the property that when elements of V are added in succession to the original vector v, each intermediate vector must also lie within the relevant subdomain. While it is possible for the algorithm to jump across narrow "walls" of irrelevancy to land in a disconnected region due to the discontinuous sampling of a continuous space, this is not a design feature of the algorithm and should never be relied upon in practice.

For example, the set of all points lying within the unit circle is a connected subdomain. The set of all prime numbers is not a connected subdomain since there is not a series of arbitrarily small additions one can make to get from one prime number to any other without visiting a nonprime integer and would therefore not be a candidate input space for the ISE algorithm. Note that allowances are made for the type of the variable under test. The

set of integers from zero to one hundred is connected, but is not connected in the floating point space. While the ISE algorithm operates only in the floating-point space, Chapter 5 discusses how the ISE algorithm can be generalized to other spaces. Also, if the tester can identify ahead of time that there exist multiple connected regions in the subdomain, the tester may run the ISE algorithm separately on each of the regions. The tester may not always be able to determine this, however, so it is important to be aware that the ISE algorithm is unlikely to discover unconnected regions.

Also note that this method relies heavily on the concept of geometrical "nearness" and therefore works best with numerical inputs. It is difficult to define a way to mutate—for example—a string to a "similar" input since the string type is heavily dependent on the semantics applied to the string. If the relevant subdomain is the set of all strings representing valid C programs, there is no readily apparent set of mutations which can be applied to traverse from any valid C program to any other via a series of transformations which themselves only yield valid C programs. If the goal, however, is to only generate strings up to a length of n characters with no additional semantics applied, it is more apparent how to define "similar" strings. For the purposes of this thesis, strings will largely be left as an area for future work to be performed.

4.3 Mechanics

In brief, the ISE algorithm resembles a random sampling along vectors weighted towards areas of relatively low sample density, and with the ability to restart the sampling from any previously visited point when a "wall" in the space is reached. The algorithm can be

broken into two main phases: (1) choosing a relatively low-density region of the space, and (2) sampling along a random vector originating in that area. Once the vector completes (discovers a boundary in the space), the algorithm repeats by choosing a new low-density region and continues until the requested number of tests have been generated. The two phases will now be described in more detail. Pseudocode of the main program loop and the bin merge methods are presented in Algorithms 1 and 2, respectively.

More generally, the ISE algorithm can be classified as a "rejection sampling algorithm." In a rejection sampling algorithm, rather than sampling from an unknown or complex distribution, one samples from a known, simple distribution and decides if a given selection also matches the target distribution. If not, the sample is *rejected* and a new sample is taken. Rejection sampling methods work best when the candidate and target distributions closely match each other, as fewer samples are rejected. The ISE algorithm in particular resemblance to the Metropolis-Hastings rejection sampling algorithm [23, 34] as the most recently sampled input strongly advises the next sample to be taken, however the ISE algorithm is not a pure Markov chain Monte Carlo simulation as all previously generated points also have a bearing on the next sample, while in Metropolis- Hastings the previously sampled value completely defines the state of the simulation.

4.3.1 Selecting Low-density Regions

A naïve implementation of the ISE algorithm would model the input space as the set of all previously generated and verified valid inputs, treating each valid input as a potential origin of a new exploratory search vector origin. Such a naïve implementation would examine the "local density" of test inputs within some radius of each candidate point. Unfortunately,

such a strategy requires an all-pairs comparison between all previously generated points to calculate each point's local density, meaning that the marginal cost of generating one additional test case is $O(n^2)$ with regard to the number of previously generated tests. If the marginal cost of generating one input is $O(n^2)$, then the total cost of generating all n inputs will be $O(n^3)$, resulting in a cubic slowdown of test generation. Early implementations of the ISE algorithm used just such a strategy and and were found to be insufficient to generate large sets of inputs (on the order of tens of thousands of inputs or higher) requiring the model to be abstracted to scale to increasing test suite sizes.

Previous work by Chan et al. [7] demonstrates that "forgetting" previously generated tests can reduce overhead for a generation algorithm while still maintaining an adequate model for future input generation. The work presents three types of "forgetting": *random forgetting*, *consecutive retention*, and *restarting*, in which test cases are forgotten randomly, in a first-in, first-out manner, or all at once, respectively. Using the generation algorithm presented in Chan's work, all three forgetting strategies showed similar results, and depending on the parameters chosen were comparable in quality to the generation strategy with perfect memory. However, the evaluation in the work was performed on a 2-dimensional rectangular region whose dimensions were known prior to testing, a space for which a true uniform distribution can be generated trivially. It is unclear how Chan's generation strategy (or its forgetting properties) would perform in more realistic or complex scenarios.

The ISE algorithm takes a different tack. Rather than using a temporal memory as used in the *consecutive retention* strategy, ISE uses a spatial memory, lumping geometrically nearby tests into an abstract grouping called a *bin*. A bin is an n-dimensional rectangular region modeling a portion of the input space. A bin maintains a single representative known

valid input within its boundaries along with its current density, defined as the number of tests generated within the bin divided by its volume (uniform density is assumed within a single bin). This greatly reduces the amount of work incurred when selecting a low density region, as rather than doing an all-pairs comparison between all previously generated inputs, the algorithm need only select the bin with the lowest test density.

However, the ISE algorithm must contend with input spaces of unknown size and shape. If a bin is defined as having constant dimensions, as the known input space grows so too does the number of bins. In this case, the bin concept has only delayed the inevitable: a nonlinear running time for the generation algorithm. To combat this, the ISE algorithm allows for flexible bin resizing and merging, allowing a constant number of bins to be maintained as information is learned about the input space. However, to maintain consistency in the bin model, there are some restrictions on how to modify a bin's dimensions. First, there is a user-defined atomic bin, for which all bins in the input space must be exactly equal to or have dimensions of a power of two of the atomic bin. Second, all bins must be placed on a power of two of the relative origin of the input space (defined as the point containing the lowest value reached for each of the dimensions under test, not necessarily the actual origin). This greatly simplifies bin merging as no bin can ever partially overlap any other bin.

In the ISE algorithm, when an input is generated which does not lie within a preexisting bin, a new bin is created to accommodate it. If there already exist the maximum allowable number of bins, the algorithm applies a heuristic to determine which two (or more) bins should be merged to accommodate without sacrificing accuracy of the model. Figure 4.1 demonstrates the binning behavior on a relatively complex input space when allowed only

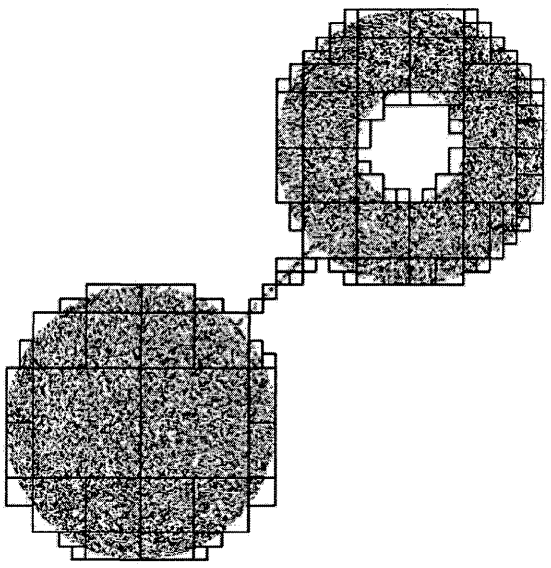


Figure 4.1: Binning behavior on complicated, 2-dimensional space

one hundred bins to model the space.

When a bin merge is necessary, several factors are considered. For one, the algorithm only tries to merge geometrically nearby bins, where nearby is defined as a user-set percentage of all pairwise comparisons of the midpoints of bins. If, for example, the user sets the nearby bin threshold to 10%, only the top ten percent of bin pairings will be considered as candidates when sorted by distance between the two bins. Second, the algorithm attempts to find the pair (or pairs) of candidates with minimal error when merged. Since bins are rectangular in nature and input spaces can be of arbitrary shape, along the boundaries of the space error can accumulate as portions of a bin may contain irrelevant inputs. The algorithm compensates for this by approximating the portion of each bin containing irrelevant inputs using an approximation strategy in the course of testing. When a "wall" is hit in the space, the ISE algorithm notes the remaining area to be covered within the

bin, and approximates the area of the bin which is invalid by computing the ratio of the line segment sampled containing valid inputs within the bin to the line segment had the sampling continued to the other side of the bin (if the samples stop halfway through a bin, it will be assumed that half of the bin is relevant). Over the course of many such samples, a better approximation of the relevant area is discovered. Finally, all other things being equal, the algorithm will choose to merge bin pairs which will encapsulate the most number of preexisting bins, reducing the number of bins in use the most and delaying the next bin merging operation. Bin merges are the most expensive operation in the ISE algorithm, and minimizing the number of bin merges required can expedite generation greatly. A bin merge requires $O(n^2)$ time to execute (required by an all-pairs comparison between midpoints of bins), where n is the number of bins in existence. However, in our experience intelligent parameter tweaking can give bin merging a very low constant multiplier with little impact on model accuracy. Further, once the input space has been discovered in its entirety bin merges are no longer required and test generation can continue at a more rapid rate.

The overall impact of the bin merge heuristic maintains high resolution near the boundaries of the space at the expense of interior regions. This is a healthy compromise as interior regions are regions that are already well understood by the ISE algorithm and are uninteresting in structure. Border regions can either indicate a true border of the input space or can signify a new frontier for which further testing should concentrate. Maintaining a high resolution near the boundaries allows for more accurate test density information to be gleaned in the areas where it is needed most. Note that in Figure 4.1 fewer bins are used to model the lower left circle. Most are used to model the more complex upper right region of the space. The algorithm automatically determined the lower left region to be simpler in

nature and require fewer bins to model with relatively high accuracy. Also note that fewer bins are used to model large interior regions, while border regions have smaller bins with a higher model resolution.

4.3.2 Test Sampling

Once a low-density bin has been selected, a random, n-dimensional unit vector is created originating from the representative input of that bin. New tests will be sampled along this vector, but first a scaling operation is applied if necessary to accommodate for the atomic bin size. The scaled vector is computed such that rather than projecting onto the (n-dimensional) unit circle, it projects onto the n-dimensional ellipsoid whose axes are defined by the atomic bin dimensions (i.e., the largest ellipsoid that can fit entirely within an atomic bin). By default, the atomic bin size is square, requiring no scaling operation. However, a vector biased to fit the atomic bin size will be more likely to generate inputs lying within the relevant subdomain (assuming the atomic bin size has been selected appropriately). The the next section will describe circumstances in which non-square atomic bins are desirable.

Next, samples are taken at exponentially increasing intervals along said vector until either a maximum number of points for the vector are created or an irrelevant input is discovered (by executing the method under test using the candidate input vector). There are two motivations for exponentially increasing sample intervals. First, less density information is known the further one travels from the representative input of the bin. The bin was chosen because it had a relatively low test density, so intuitively the majority of tests generated due to that bin's selection should lie in or near that bin. Second, the exponential speedup is used to quickly locate the boundaries of the space. Especially in input spaces which are

partially unbounded (in one or more directions), it is important to determine this quickly so that the entire space can be discovered as quickly as possible. The distribution generated by the ISE algorithm cannot be considered uniform until the entire space has been discovered. This means that up until some threshold of tests have been generated, the tests generated by the ISE algorithm are biased. Lowering this threshold as much as possible is one of the primary design goals of the ISE algorithm[1].

Algorithm 1 ISE Main Loop

```
generatedPoints ⇐ 0
put initialPoint in appropriate bin
while generatedPoints < requestedPoints do
  bin ⇐ getBinWithLowestAdjustedDensity()
  v ⇐ chooseRandomVector()
  currentTestInput ⇐ bin.startPoint + v
  while numberOfPointsOnVector < MaxPointsPerVector do
    if currentTestInput + v is relevant then
      put currentTestInput in appropriate bin
      while binCount > MaxAllowedBins do
        mergeBins()
      end while
      numberOfPointsOnVector + +
      currentTestInput ⇐ currentTestInput + (v * BoundaryScaleFactor)
    else
      if currentTestInput lies in a preexisting bin b then
        b.incrementMissCount() by the number of times tests would have been
        executed in that bin had sampling continued
      end if
    end if
  end while
end while
```

[1]Note that discovery of the entire region is a necessary but not a sufficient condition for a uniform distribution. Once an entire region is discovered, the ISE algorithm may have to "fill in the holes" left over by the algorithm, making the region discovery threshold less useful as a uniformity metric

Algorithm 2 Bin Merge Operation

for all Bins b1, b2 in InputSpace **do**
  $candidateBinPairs \Leftarrow candidateBinPairs \cup (b1, b2)$
end for
Sort $candidateBinPairs$ in increasing distance between midpoints of $b1$ and $b2$
$candidateBinPairs \Leftarrow candidateBinPairs.truncateAt(MergeCandidacyCutoff)$
$candidateBinPairs \Leftarrow candidateBinPairs$ with minimal estimated error
$candidateBinPairs \Leftarrow candidateBinPairs$ with maximal enclosing bin count
$mergedBin \Leftarrow$ random selection from remaining candidateBinPairs
for all Bins b in $InputSpace$ **do**
  if b overlaps $mergedBin$ **then**
    $mergedBin.hitCount+ = b.hitCount$
    $mergedBin.missCount+ = b.missCount$
    $mergedBin.startPoint = b.startPoint$
    $InputSpace \Leftarrow InputSpace/b$
  end if
end for
For any area in $mergedBin$ unaccounted for by preexisting bins,
assume area is irrelevant. To compensate, we adjust
$mergedBin.missCount \Leftarrow mergedBin.missCount * (adjustedVolume/TrueVolume)$
$InputSpace \Leftarrow InputSpace \cup mergedBin$

4.3.3 User-defined Parameters

There are several parameters to the ISE algorithm that can be manipulated by the tester. For most of these parameters, there exist defaults reasonable for most input spaces.

First is the *atomic bin size* parameter. This is an n-dimensional array describing the dimensions of the atomic bin size for the input space. By default, the atomic bin size has dimensions of 1 along each axis, and for most cases this is sufficient. However, if the tester is aware of the rough size of the input space, tweaking the atomic bin size can expedite bin merging and help achieve region discovery more quickly. For example, if the input is a two dimensional vector describing an automobile where the x axis is the weight of the car in kilograms and the y axis is the maximum velocity of the car in kilometers per hour, intuition states that the weight of the car can range in the thousands of kilograms, while

the maximum velocity of any car will most likely be in the hundreds of kilometers per hour. With this knowledge, the tester will know that the space will most likely appear "wider" than it does "tall." Choosing a similarly shaped atomic bin size will both simplify bin merges and will also expedite the input space discovery, as the search will be biased to take larger strides in the x direction than the y direction knowing that the space is wider than it is tall. This is the reasoning behind the scaling operation performed on the randomly selected uniform vector previously discussed in this chapter. As a rule of thumb, an optimal atomic bin size should be between one and two orders of magnitude smaller than the largest conceivable valid and relevant value in each dimension, but depending on the input space, the tester should use his or her discretion.

Second is the *maximum number of bins.* This parameter tells the algorithm how many bins are allowed to model the input space. More bins means a higher resolution model and a more accurate approximate uniform distribution of test cases, at the expense of running time. The cost of a bin merge operation increases at the square of this parameter. In the worst case, for an input space requiring a bin merge after every generated test, the running time to generate n tests would be $O(n^3)$. However, in all spaces tested in this work, bin merges happen much less frequently. Still, the ISE algorithm spends the majority of its time computing bin merges, so a change in the maximum number of bins will be evident in the running time of the algorithm. For the purposes of this work, the default maximum number of bins is one hundred. This is an arbitrary value, chosen because it resulted in adequate running times while still accurately modeling the evaluated input spaces.

Third is the *number of tests to be generated.* This value has no default as it is completely dependent on the space under test and the requirements of the tester. While the value can

be set arbitrarily small, the tester should be aware that the ISE algorithm only generates approximately uniform distributions *in the limit.* Setting the value too low will result in a biased distribution which may not be useful in evaluating the quality of the software.

Next is the *boundary search scale factor.* This parameter tells the ISE algorithm how quickly to increase the exponentially growing step size when sampling along a vector. The reasoning behind the exponentially growing step size was explained in the previous subsection. This parameter allows the user to modify the priorities of the algorithm; a larger scale factor will locate boundaries of the algorithm faster, but require more frequent low-density region selections and will potentially leave more "gaps" which must be filled in after all boundaries have been discovered. A lower value will keep the distribution more uniform *within the known region*, requiring less "filling in" after the region has been discovered. However, it will also take more iterations to discover the region. The default is value is 1.1, meaning that each step size is 10% larger than the previous step size. Again, this value is relatively arbitrary, but performed well in evaluations.

The *maximum points per vector* parameter places a cap on the number of points laid down by an exploratory vector, regardless of whether or not a boundary of the input space has been reached. This allows a similar tradeoff to the boundary search scale factor, prioritizing either finding boundaries or frequently placing new vectors, filling in previously discovered space.

The *bin merge candidacy cutoff* describes how many bin pairs to consider as candidates when a bin merge operation occurs. Next to the maximum number of bins, this parameter has the largest impact on running time of the ISE algorithm. The cutoff is expressed as a fraction of all pairs of bins when sorted in order of increasing distance, i.e., only consider the

n% closest bins as candidates. The default value for this parameter is 0.05 (5%). This value has been calculated to be sufficient for most two-dimensional spaces. As a rationale for this value, consider a two-dimensional space containing n bins. In two dimensions, each bin can have a maximum of four equidistant closest neighbors (one in each cardinal direction). To consider all neighboring pairs of bins as candidates, this requires at least $2n$ candidate pairs ($\frac{4n}{2}$ since neighboring is a commutative property; a neighbors b implies b neighbors a). Considering the set of all pairs of bins is $\frac{n(n-1)}{2}$, we use the equation

$$c\frac{n(n-1)}{2} \geq 2n \tag{4.3}$$

and find that a c value of 0.05 is sufficient to contain all neighboring bin pairs for all two dimensional input spaces containing more than 80 bins (100 being the default number used by the ISE algorithm). Higher dimensionality of the space may require one to increase the value of c as the number of neighbors increases with the square of the dimensionality. However, one should note that the c value chosen is very conservative and does not take into account bins with fewer than four neighbors, nor does it consider the fact that many bin merges are equally "good," which does not require the set of all bin neighbors to be considered. Further, it assumes that all bins are equally sized. Since bin distances are measured from their midpoints, larger bins are going to be more distant from their neighbors and may not be considered prime candidates for merging. In practice, values as low as 0.01 have been used with no impact on the quality of the distribution on a bounded two dimensional space.

Finally, the tester must provide a single relevant input as a seed for starting the input

space search. This is the only *required* parameter to be set by the tester and does not have a default value, as it is completely dependent on the input space under test. This will be the origin of the first search vector, and will be contained in the first bin created by the search. After the search has started, the seed is treated the same as any other previously generated input. However, since it is the starting point of the search, until the entire region is discovered there will be an inherent bias to select inputs near the starting point. Ideally, the starting point will be centrally located within the space, but this is not necessary to achieve total coverage.

Chapter 5

The Automated Testing Framework

The Automated Testing Framework (ATF) is written as a proof-of-concept regarding the feasibility and effectiveness of the ISE algorithm and allows for the hybrid integration of several test generation strategies. The ATF contains a Java implementation of the ISE algorithm (along with two other automated input generation strategies) accompanied with tools to simplify test definition and execution. The ATF requires a JRE version 1.5 or higher, the ant Java build tool, and optionally an installation of the Java Modeling Language (JML) to utilize the JML functionality.

One of the primary design goals of the ISE algorithm is to allow several test input generation strategies to act in concert in the creation of a single input. By delegating portions of input vector generation to different generators, the tester gains parameter-level control over how input generation criteria is defined.

5.1 Automated Testing Strategies

The ATF is designed to accommodate several typical automated testing strategies to be used independently or in tandem. First is the ISE generation strategy presented in this work. The ISE strategy is designed to explore input spaces to methods whose parameters are interdependent or if any dependencies between parameters are unclear with limited or no documentation. Second is pure random testing, for which a true distribution for a parameter to a method can be easily solved. Random testing is fast and simple when there exist no dependencies between parameters and the parameter list can be easily decomposed into selections from several independent distributions. A tester which delegates independent parameters to pure random testing can accelerate ISE generation by reducing the dimensionality of the ISE problem. This also has the benefit of allowing the ISE algorithm to maintain a higher model resolution by simplifying the space for the bin model. Finally, the ATF supports Bounded Exhaustive Testing (BET), a strategy first proposed by Marinov et al. [33] and coined as a term in Sullivan et al.'s work [46]. Bounded exhaustive testing is the testing of all possible inputs up to a specified size (or within a specified range).

The ATF allows one to decompose the input space of a method such that different parameters may be generated using different strategies and then reconstituted as a test input for the method under test. For example, two dependent parameters may be generated using the ISE algorithm to leverage its ability to explore dependencies between parameters while a third, independent parameter may be tested with a random generation strategy. As the ATF tests at the method level where each test consists of an input/output tuple associated with the method, each execution of the method is treated as an independent

test with no interplay between previously executed tests on that method (or tests on other methods in the same test suite).

It is important for the tester to understand the interplay between the testing strategies to get the most out of a testing run. The input for a single execution is generated in two passes. The first pass "fills in" all parameters marked to be tested with the ISE algorithm and the pure random method. The second pass fills in the remaining parameters marked to be tested exhaustively. After the test is executed, the ISE and random parameters are held while all combinations of all exhaustive parameters are generated and executed in turn. Only after all exhaustively generated parameters are tested does the next set of ISE and random parameters get generated, and are then tested again with the same set of defined exhaustively generated parameters. Table 5.1 shows a portion of the tests executed in a mixed-strategy test suite.

Random	ISE	Exhaustive (Boolean)	Exhaustive (Boolean)
a_1	b_1	True	True
a_1	b_1	True	False
a_1	b_1	False	True
a_1	b_1	False	False
a_2	b_2	True	True
...	...	...	...

Table 5.1: Sample execution of mixed-strategy testing with ATF

Note that with this ordering, it is not advisable to request both a large number ISE/random tests and exhaustively generated tests, as the total number of executed tests is the *multiple* of the two numbers.

One other interplay between the generation strategies that a tester must be aware of is how the concept of "relevance" is handled in the ISE algorithm. By itself, the ISE algorithm

will generate an input, execute the test, and then determine the relevance of the input based on the behavior of the test execution. However, in a mixed-strategy environment it becomes less clear where to lay the blame for relevance. It is possible that the same parameters generated by the ISE algorithm would have resulted in a relevant whole input had the pure random strategy chosen different values for its parameters. However, the way the ATF has been defined and constructed allows the ISE algorithm to neatly sidestep this issue. All random and exhaustively tested parameters are assumed to be independent. All dependent parameters should be placed under the purview of the ISE algorithm to manage these dependencies. Therefore, any generated input that fails the relevancy check must involve ISE-managed variables. All other parameters are assumed to be independent of any relevancy check. Therefore, it is wise for a tester to include any parameter whose dependency is in question to err on the side of caution and include it within the ISE generation strategy.

However, this definition might not always be convenient, so the ATF relaxes this assumption somewhat. In the case where the ISE algorithm is used in tandem with exhaustive testing such that one set of ISE-controlled parameters is tested against multiple exhaustively generated parameters, if *any one* of the inputs is found to be relevant, the ISE algorithm will consider the parameters under its control to be relevant as there exists at least one known relevant input containing those parameters. This allows the tester to place "mostly" independent parameters (those which might possibly contribute to relevancy, but when iterated over several values are extremely unlikely to only result in irrelevant inputs) under the control of the pure random generation strategy. This is to encourage the tester to place as many parameters as possible outside of the ISE algorithm, as each additional parameter controlled by ISE increases the dimensionality of the explored space and as a result increases

the time complexity of test generation.

5.2 Supported Types

In an unmodified state, the ATF has full support for all Java primitives. This means that a method containing only Java primitives can be tested using any three of the generation strategies with no additional effort. It also provides an ordering of primitives such that ranges within a type may also be defined so exhaustive testing on a subset of a type or random testing within a range are also possible.

The ATF also provides an interface to add new classes to its library of known types. All that is required of the tester is to write a wrapper class conforming to the *InstantiatedParameter* interface (described in detail in Appendix A). In brief, this interface requires the tester to define three things regarding the class. First, the tester must define a strict ordering of all possible instantiated objects of the class, including an absolute minimum, an absolute maximum, and a method to increment from one object to another such that if one starts with the absolute minimum, repeated increments will pass through all possible objects of the class terminating with the object defined as the absolute maximum. This ordering is used to define the bounded exhaustive testing strategy.

Next, the tester needs to provide a method to generate a random object within the class. The randomly generated object must lie within the range set by the maximum and minimum configuration (either the absolute maximum and minimum or the user-defined maximum and minimum if configured, which may be some connected subset of the true range). This is used for the pure random generation strategy.

Finally, the tester must provide a method to serialize and unserialize an object to an array of floating point numbers. This is required for the ISE algorithm to work since it uses an underlying floating point model regardless of the types under test. The serialization may be defined any way the tester wishes, but best results will occur when "similar" serialized floating point vectors will translate to "similar" objects within the class under test so that relevant inputs are likely to lie in connected regions. Also, one additional caveat that must be placed on the tester is that the serialized array must be of a known, fixed length. This means that variable-sized classes (such as linked lists, queues, etc.) will not translate well to use in the ISE algorithm without a fair amount of abstraction.

The ATF also provides rudimentary support for arbitrary objects which have not been registered. Without any way to order elements in an arbitrary class, the ATF cannot define ranges for random testing nor can it know how to permute one object into a "similar" object of the same type. However, when an unknown object is used as a parameter to a method, the ATF will gracefully degrade and rely on the object's default constructor to create a new object of that type for testing purposes. Any attempts to generate a "random" object or to iterate to the next object in the exhaustive testing strategy will simply return the object created by the default constructor. While this means that this parameter effectively goes untested, it is still possible to run tests on methods containing non-primitive types as parameters. If the parameter list is a mix of primitive and non-primitive types, it may still be possible to glean useful testing information by only thoroughly testing the primitive input types.

5.3 Test Configuration and Execution

Once the tester is ready to begin testing, the tester must create an XML definition of the testing plan. The format of the configuration file is explained in detail in Appendix B, but the syntax is quite simple. A sample definition for the test suite generated in Table 5.1 would look like this:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE Configuration SYSTEM "TestGenConfig.dtd">
<Configuration>
        <ClassUnderTest>TestedClass</ClassUnderTest>
        <MethodSignatureSuite>
             <MethodSignature>
                  <Name>testedMethod</Name>
                  <Parameter type="int" strategy="random" count="5"/>
                  <Parameter type="int" strategy="ise" count="5"/>
                  <Parameter type="boolean" strategy="exhaustive"/>
                  <Parameter type="boolean" strategy="exhaustive"/>
             </MethodSignature>
        </MethodSignatureSuite>
</Configuration>
```

The above configuration file would execute *TestedClass.testedMethod(int, int, boolean, boolean)* twenty times with five different pairs of randomly/ISE generated integer inputs (each tested with all four combinations of possible boolean value pairs). In general, a configuration is defined as a set of methods within a class which the tester wishes to test. In this case, only one method was configured for testing. A fully configured method in turn contains a parameter list of all expected parameters of the method combined with information regarding the generation strategies to use.

When the test suite has been configured, execution begins. There are two pieces of information that must be gleaned from each executed test: (1) Was the test input relevant?

and (2) Was the test output correct? In its default configuration, the ATF treats any generated input as relevant and treats any unhandled exception as a detected failure of the system. However, if the software under test has been modeled using JML [28], the ATF gains all modeled design by contract knowledge and can more accurately determine deviations from specification. Specifically, JML allows preconditions and postconditions to be described in a formal, machine-readable language resembling Java syntax. The ISE algorithm will also use any defined preconditions of methods to automatically create an oracle to determine the relevancy of a given input; a violation of any precondition to a method will automatically flag the input as irrelevant. A violation of any postcondition, in turn, will be flagged as deviation from the specification and will be marked as an error. In this way, the tester can more accurately model the space under test and detect less obvious errors in the software. As an example, consider the following simple implementation and modeling of a setter method which sets the age of a Person object:

```
//@ requires newAge >= 0;
//@ ensures this.age == newAge;
public void setAge(int newAge)
{
    if(newAge == 42)
      // deviant behavior
      this.age = -newAge;
    else
      this.age = newAge;
}
```

In this example, the precondition of the method requires only positive integers be provided as inputs. Therefore, any negative integer will be disregarded as input and considered irrelevant by the ISE algorithm. The JML annotations also state a postcondition of the

method, requiring the age member of the Person object to be equal to the provided parameter by the end of execution of the method. In this implementation, this postcondition is violated when the number 42 is provided as input to the method. In this case, the ATF (running in the JML runtime environment) will catch this deviation from the specification and notify the tester.

Chapter 6

Evaluation

6.1 Performance on Synthetic Input Spaces

To evaluate the performance of the ISE algorithm, several synthetic input spaces of varying complexity were created. The first (and simplest) space is a *circle* with radius five. An example of this space can be seen in Figure 6.1. The second space is referred to as *figure-eight* and consists of two circles of radius five, one centered at the origin and one centered at $(0, 9)$. There also exists a void circular region of radius 2 also centered at $(0, 9)$. This region demonstrates the performance of the ISE algorithm in concave regions and regions containing "obstacles" which must be navigated to discover the entire region. An example of this space can be seen in Figure 6.2. Finally, the space named *barbell* consists of two circles of radius 5, one centered at the origin, and the other at $(9, 9)$. The circle at $(9, 9)$ again has a void circular region of radius 2 centered within it. The two major circular regions of the space are connected by a diagonal strip along the line $y = x$ of width 0.5. This region is crafted to demonstrate the utility of the ISE algorithm when large regions of

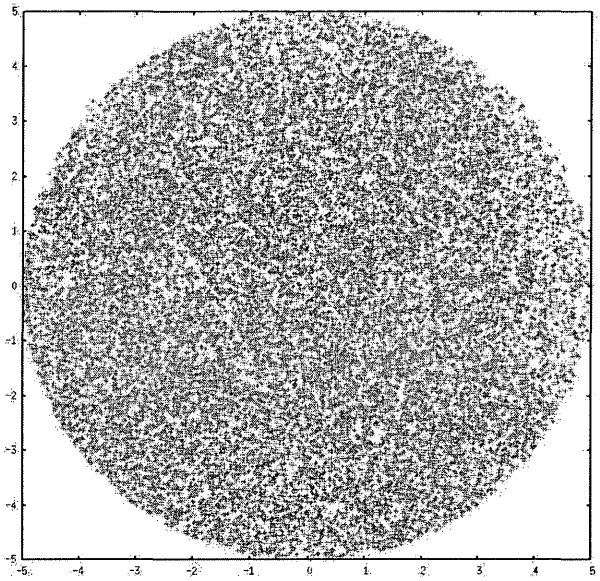


Figure 6.1: 20,000 tests generated in the *circle* input space

relevancy are loosely connected by narrow passages. An example exploration of this space can be seen in Figure 6.3.

There are three metrics by which the ISE algorithm is evaluated. First is accuracy. The ISE algorithm is designed as a replacement for random testing when selecting from a true uniform distribution is difficult. Therefore, it is important to evaluate how closely the ISE algorithm approximates a uniform distribution. Second is running time. One of the benefits of random testing is that it can generate many tests much more quickly than a human can. It is important that the ISE algorithm also generates tests in a timely fashion, and also generates tests at a constant rate (i.e., generation does not slow down over time). Finally, one of the benefits of the ISE algorithm over Monte Carlo random selection is that the ISE algorithm makes an effort to only generate tests within the defined relevant subdomain. While it must make exploratory steps outside the subdomain to refine the

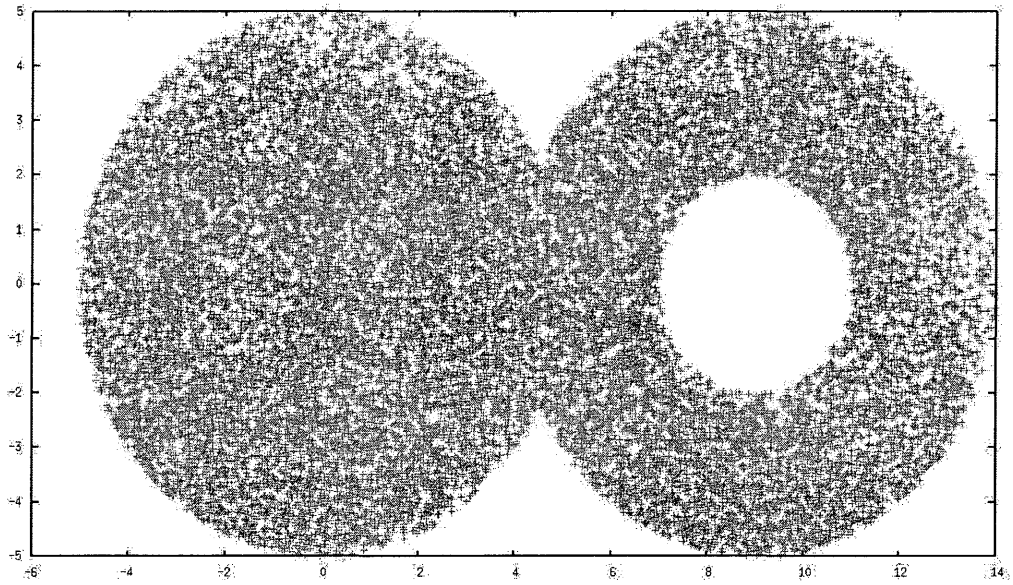


Figure 6.2: 20,000 tests generated in the *figure-eight* input space

model, minimizing the amount of irrelevant executed tests is an important feature of the algorithm. Therefore, the ability to "stay inside the lines" is also evaluated.

In this evaluation, the ISE algorithm generated from 20,000 to 100,000 tests at 20,000 test intervals, executing ten times at each interval to obtain an average. All data presented here is averaged over all executions. Unless otherwise noted, all tests were performed using the default ISE user-definable settings as described in Chapter 4.

6.1.1 Accuracy

To evaluate how closely the ISE algorithm approximates a uniform distribution, we define the *local density* and the *deviation metric*. The local density is defined as the number of generated tests within some distance (0.25 units in this evaluation) of a selected point. The deviation metric measures how closely two sets of samples resemble each other. One data

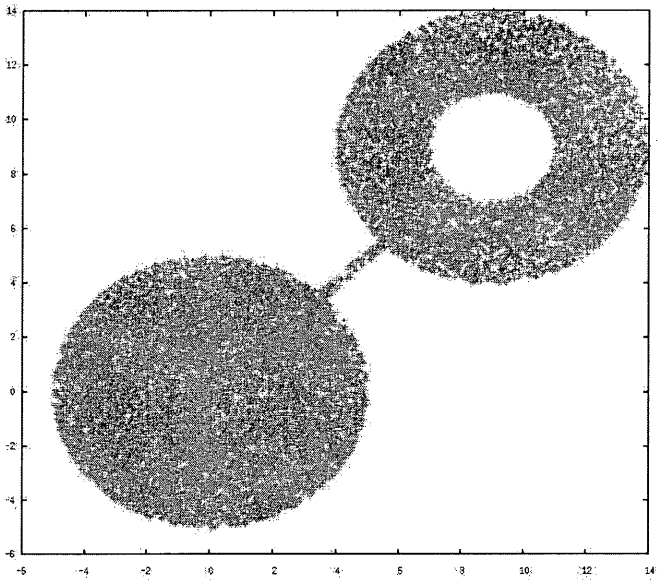


Figure 6.3: 20,000 tests generated in the *barbell* input space

set is treated as the canonical distribution (in this evaluation, the canonical distribution is always a sampling from a Monte Carlo random input generation strategy) and the other is the candidate distribution. Samples of the local density are taken at regular intervals from both regions (every 0.5 units in both the x and y dimensions in this evaluation) and the percent error between the canonical and the candidate distribution is computed. After all local density errors are computed, the deviation metric is defined as the average of all density errors across the entire input space. A deviation metric of zero indicates that the canonical and candidate data sets are identical, while, for example, a deviation metric of one indicates that the local density varies on average one hundred percent from the canonical distribution. The combination of local density and the deviation metric allows one to both discover local regions where testing was weak as well as have a single, global value to evaluate the overall quality of the ISE algorithm.

To perform this evaluation, tests generated by the ISE algorithm serve as the candidate data set. For each of the synthetic regions evaluated, selections from a true uniform distribution is possible by creating a bounding box around the defined region and sampling from within the bounding box, discarding any generated point within the bounding box but lying outside the defined region. The canonical data set is represented by a sampling of an equal number of tests from such a distribution.

Table 6.1 shows how accurate the ISE algorithm performs on the circle input space. For comparison's sake, the deviation metric comparing the biased distribution shown in Figure 2.2 with an accurate uniform distribution is also shown. Also, the deviation metric when comparing two unbiased random distributions is shown as a baseline. One should note that as the number of tests generated grows, the unbiased random and ISE algorithm's deviation metric do not converge. However, the ISE algorithm is far more accurate than the previously discussed biased distribution in Chapter 2. While the ISE algorithm does perform a best-effort attempt to approximate a uniform distribution, it is still an approximation and a tester should be aware that the distribution is not perfectly uniform. Most of this bias is due to a slight propensity of the algorithm to select points near the border of a space due to the fact that the space is modeled as a set of rectangular regions. Near the borders, the rectangular model breaks down. The algorithm compensates for this by both attempting to maintain higher resolution near the border of the algorithm as well as approximating the area of the bordering rectangle occupied by irrelevant space, but the model is not perfect. One possible way around this issue is to use the ISE algorithm to discover an unknown region and use the bin model generated by the ISE algorithm to perform true random selections using a Monte Carlo strategy from the resulting set of rectangular bins.

Strategy	Generated Points				
	20000	40000	60000	80000	100000
MC	0.194 (0.010)	0.142 (0.008)	0.116 (0.007)	0.107 (0.009)	0.092 (0.007)
ISE	0.215 (0.013)	0.182 (0.011)	0.179 (0.015)	0.161 (0.014)	0.154 (0.010)
Biased	0.528 (0.015)	0.515 (0.008)	0.509 (0.010)	0.508 (0.009)	0.509 (0.007)

Table 6.1: Deviation metric comparing unbiased Monte Carlo generation, ISE, and biased random generation strategies when generating in the *circle* input space (Standard deviation across 10 trials in parenthesis).

Tables 6.2 and 6.3 show similar results for the figure-eight and barbell regions, respectively. Again, while the two distributions start being nearly indistinguishable, as the number of tests grows, the disparity between the two distributions becomes more apparent. While the ISE algorithm does an excellent job of discovering regions of high complexity and covering reasonably well across the region, it is not a suitable replacement when a true uniform distribution is required to make statistical inferences on the quality of software. When a true uniform distribution is not available, the ISE algorithm can serve as a useful approximation as long as the tester is aware of its fallibility.

Strategy	Generated Points				
	20000	40000	60000	80000	100000
MC	0.258 (0.014)	0.188 (0.006)	0.169 (0.009)	0.145 (0.006)	0.132 (0.008)
ISE	0.308 (0.015)	0.257 (0.012)	0.254 (0.014)	0.242 (0.012)	0.241 (0.014)

Table 6.2: Deviation metric comparing unbiased Monte Carlo generation and ISE generation strategies when generating in the *figure-eight* input space (Standard deviation across 10 trials in parenthesis).

6.1.2 Running Time & Wasted Effort

Next, the running time of the algorithm is evaluated for generating inputs in all three input spaces, shown in Table 6.4. Even for the worst-case region, generating one hundred

Strategy	Generated Points				
	20000	40000	60000	80000	100000
MC	0.281 (0.015)	0.199 (0.009)	0.169 (0.009)	0.145 (0.007)	0.132 (0.008)
ISE	0.335 (0.018)	0.310 (0.015)	0.308 (0.017)	0.308 (0.024)	0.336 (0.033)

Table 6.3: Deviation metric comparing unbiased Monte Carlo generation and ISE generation strategies when generating in the *barbell* input space (Standard deviation across 10 trials in parenthesis).

thousand inputs takes less than a minute, well within the bounds of practicality. Also note that test generation also appears to perform in sub-linear time. This is most apparent in the barbell region, where generating the first twenty thousand inputs takes about forty seconds, while generating an additional eighty thousand inputs only takes another ten seconds. This is due to the fact that the majority of the effort in the ISE algorithm occurs when merging bins, which only occurs while the region is still being discovered. Once the region has been discovered in its entirety, no more bin merge operations are necessary and test generation speeds up dramatically. While all of these example scenarios are bounded and finite, an evaluation of the running time in an infinite space in which bin merges occur constantly is discussed in the case study later in this chapter.

Region	Generated Points				
	20000	40000	60000	80000	100000
Circle	16.379	19.800	23.305	25.886	28.182
Figure-eight	37.403	41.505	43.559	42.411	49.597
Barbell	39.362	41.874	45.235	48.108	50.482

Table 6.4: Time (seconds) to execute ISE algorithm on several regions for increasing amounts of generated points.

Finally, the wasted testing effort of the ISE algorithm is analyzed. In the course of testing from an unknown space, it is inevitable that some tests will lie outside the space.

One of the primary design goals of the ISE algorithm was to minimize the amount of wasted testing effort when sampling from a space. Table 6.5 demonstrates the efficiency of the ISE algorithm in this respect when compared to sampling from a Monte Carlo distribution defined by the smallest bounding box containing the entire input space.

Region	ISE Hitrate	Monte Carlo Hitrate
Circle	0.84310	0.78551
Figure-eight	0.83780	0.70867
Barbell	0.81664	0.37788

Table 6.5: Hit rates for ISE and Monte Carlo generation strategies on several input spaces.

Regardless of the space under test, approximately 80% of all generated tests lie within the relevant subdomain and will serve as useful tests. The efficiency of Monte Carlo generation on the other hand is very sensitive to the space under test. Monte Carlo generation simply becomes infeasible when the bounding box is very large but contains very few relevant tests to the point where it is unlikely to generate any valid inputs. In other words, the biggest gains in efficiency ISE algorithm can be seen in spaces where the relevant volume is dwarfed by than the type-complete volume of the space.

6.2 Case Study

The ISE algorithm was used to test the KB3D software written at the National Institute of Aerospace by Cesár Muñoz [39] in both C++ and Java. KB3D is an algorithm for aircraft collision detection and avoidance. The input to KB3D consists of two aircraft where an aircraft is defined by a three-dimensional position vector and a three-dimensional velocity vector. KB3D determines if the aircraft will violate each other's airspace within a set

amount of time following their current courses. If KB3D determines the aircraft will violate minimum separation, KB3D also provides a set of resolution vectors for the aircraft to follow to avoid the conflict while also minimizing deviation from their current courses. KB3D has been formally verified in the Prototype Verification System [42] and contains runtime sanity checking of its results. One of the ancillary goals of testing this software was to evaluate how much faith can be placed in formal verification over software testing.

While KB3D accepts any floating-point vectors as input, to properly test the course correction component of the software, one must generate pairs of aircraft on a near collision course, otherwise there is no correction for the aircraft to follow. This relevant subdomain is orders of magnitude smaller than the set of all pairs of aircraft in general, and creating a custom solver to generate only such pairs of aircraft is a difficult and time consuming task. Testing in this region is a prime candidate for the ISE algorithm. To test this region, the KB3D software was slightly modified to throw exceptions when runtime errors were detected rather than write to standard output. While this was not necessary (it is also possible to run a regular expression on the output to determine if an error was detected), it simplified matters greatly. No additional changes were required.

The evaluation consisted of ten trials generating and executing ten thousand inputs each followed by another ten trials generating and executing twenty thousand inputs each. All ISE configuration defaults were used except for the bin merge candidacy cutoff, which was set at 0.01 rather than 0.05 to accelerate test generation at the expense of model accuracy. This was deemed appropriate due to the infinite nature of the input space, making strict model accuracy impossible. The practical benefits of generating many useful tests outweighed the need for modeling uniformly across the input space. One sample pair

of aircraft from the relevant subdomain provided in the KB3D documentation was used as a seed for the ISE algorithm.

The effectiveness of the hybrid input generation functionality of the Automated Test Framework was evaluated by running an additional ten test executions generating ten thousand tests each on the KB3D software with the addition of one additional parameter—lookahead time. By default, KB3D looks five minutes into the future to predict a conflict. In the ATF evaluation, each aircraft pair generated by the ISE algorithm was exhaustively tested with a 5, 6, 7, 8, 9, and 10 minute lookahead. Additional logic was inserted to assert that if a conflict was detected with some lookahead time, all larger lookahead times with the same pair of aircraft must also detect the conflict.

6.2.1 Accuracy

In this case, there exists no canonical distribution for comparison. However, no such distribution is necessary; one can state unequivocally that the ISE algorithm does not generate anything resembling a uniform distribution. Since the input space is infinite (translate a pair of aircraft one hundred miles to the north and you get another pair within the relevant subdomain, for example), the ISE algorithm is constantly pushing the boundary of the known space. Because the search begins at the seed input provided at the start of testing, the known space expands around that point but never discovers the entire space.[1]

[1]Technically, the space is bounded trivially by the maximum and minimum floating point values allowed by the language, but in practice these bounds will not be reached.

6.2.2 Running Time & Wasted Effort

Over ten trials, the ISE algorithm was able to generate ten thousand tests in an average of 7.08 minutes. The algorithm was able to generate twenty thousand inputs in an average of 13.90 minutes. Here we see the linear growth which was not apparent in the synthetic input spaces. Where the synthetic input spaces were bounded and thus discovered in their entirety quickly, the ISE algorithm is constantly refining the input space model in the KB3D algorithm. This is the running time one would expect in a scenario with a complex and infinite input space.

The biggest benefit of the ISE algorithm is the efficiency of test generation. Over all executions, on average 89.1% of tests generated by the ISE algorithm were within the relevant subdomain and served as useful tests. In a case like this, the Monte Carlo generation of a dozen floating point numbers which happen to define two aircraft on a collision course is infinitesimal.

In the hybrid ISE/exhaustive test generation scenario, running time was largely unchanged, executing all fifty thousand tests in 7.37 minutes on average. Even though the number of tests executed is sextupled, the running time increases by less than five percent. This is due to the fact that both test execution and exhaustive generation are relatively trivial operations in this case, while the ISE algorithm is responsible for the majority of the running time. There is very little overhead when combining test generation strategies, and in fact testers are encouraged to offload "known quantities" from the ISE algorithm to more efficient generators to improve the performance of test generation in general.

6.2.3 Discovered Bugs

Over the course of testing, two errors were detected. First, there existed a memory leak in the C++ implementation of the KB3D algorithm which became apparent after repeated testing without restarting KB3D. This error could be a result of the simultaneous development of the Java version which need not worry about freeing unneeded data structures or could be a conscious decision based on the fact that KB3D is meant to be restarted for each pair of input vectors. Regardless, any stress testing whatsoever (including testing the same input many times) would have revealed this error, so attributing its discovery to the ISE algorithm itself is a bit presumptuous.

More interestingly, there exists a case where the runtime assertion checking will falsely flag an execution as failing. Specifically, in cases of steep ascent or descent, the runtime assertion checker will sometimes deem the resolution maneuvers as too "extreme," thinking there exists a solution which allows the aircraft to deviate less from their current course. This error occurs due to the use of an epsilon value which is not small enough to capture all information about the headings when aircraft are not in level flight. While it is important to note that the error is in the runtime checking and the solutions presented by KB3D are in fact correct, it is interesting to find any deviations from expected behavior in software that has been formally verified. In practice, this error occurred in approximately 1.8% of all generated tests.

The hybrid generation strategy discovered no further errors and had a similar error detection rate for the previously mentioned error of approximately 2.1% of all tests. While there is no compelling evidence in this evaluation that a hybrid ISE approach is more likely

to detect an error than an ISE-only approach, the minimal running time overhead accrued with a hybrid approach suggests that hybrid test generation is feasible and warrants further study.

Chapter 7

Conclusion

There are three primary contributions this work has made to the state of the art of random testing. First and foremost, it acknowledges that random testing from a space is impossible if the space is not explicitly defined prior to testing. In practice, the space may be an emergent property of the software only implicitly defined by the behavior of the software itself. In cases such as these, uniform random testing becomes impossible. Even the manual creation of test inputs may not instill confidence in the tester as without an explicit definition of the space, the tester may not capture all aspects of the software under test in his or her test suite. In software with such implicitly defined subdomains, both automated test generation as well as a visualization of the space under test to understand undefined emergent properties of the software would be beneficial to the software tester.

Second, this work presents a set of heuristics for the exploration of such an implicitly defined subdomain in the Implicit Subdomain Exploration algorithm. This algorithm allows for the approximate uniform sampling from an arbitrary connected space even if the boundaries of the space are not known before sampling begins. While the ISE algorithm

was designed primarily with software testing in mind, it may have applications in other areas such as simulations in which unbiased samplings from spaces of unknown structure may be necessary. The ISE's potential as an input space visualization tool has also been largely unexamined and may have some useful applications in that area. It is also important to note that the ISE algorithm is only a set of heuristics and there is nothing inherent in its design to be the best method of exploration of implicitly defined subdomains. It is entirely possible that other methods may be designed which may discover spaces faster or more accurately approximate uniform distribution sampling. Future work in both improving the ISE algorithm as well as the development of entirely different means of sampling from unknown spaces are promising areas of research.

Finally, this work presents a framework built around the ISE algorithm coupled with other automated testing strategies like bounded exhaustive testing and naïve random testing. This framework both allows for the practical use of the ISE algorithm in real-world software and also allows for the tester to factor test parameter generation, delegating different parameters to a generator to which the parameter is most suited. To the author's knowledge, this delegation of test input generation to multiple generators to be later reconstituted into a single test unit has also never been performed. The Automated Testing Framework with the use of the ISE algorithm presented here was used in the evaluation of real-world aircraft collision avoidance software to which naïve test input generation was infeasible. The testing required no custom generation software written for the input space under test and required only minor modifications to the software under test. Testing revealed two deviations from expected behavior, including one relatively rare error occurring in only a small fraction of generated tests. It is unlikely that such an error would have

been detected in a test suite designed by a human as it involved the confluence of several factors resulting in a floating-point error unlikely to be predicted beforehand. However, the case study served mainly as an evaluation of the ISE algorithm; the utility of delegating generation to several generation strategies remains largely untested and remains an open question of this work. Further evaluation of multiple-generator generation strategies in software more suited to such a method (i.e., software taking parameters of multiple types, some of which are independent and some of which are dependent parameters) is another avenue of future work.

In brief, this work identifies the problems surrounding implicitly defined subdomains, offers one possible method of addressing these problems in the ISE algorithm, and evaluates the effectiveness of the solution in using the ISE algorithm and the Automated Testing Framework to test aircraft collision detection and avoidance software. The evaluation presented here shows limited but promising success in the use of the ISE algorithm as a test input generation tool and opens the door for future refinements to the ISE algorithm and improvements to random test data generation in general.

Appendix A

Extending the Automated Testing Framework

By default, the Automated Testing Framework (ATF) only has support for testing methods whose parameters consist only of Java primitives. Non-primitives are treated as Java *Objects*, of which the ATF has no knowledge. Instead, the ATF falls back and will always instantiate the object using its default constructor, no matter which testing strategy is indicated for that parameter.

To extend the ATF to support other types, one must extend the *InstantiatedParameter* abstract class in the *edu.wm.test* package. An *InstantiatedParameter* consists of a payload indicating the parameter's current value and several methods of iterating through possible values for the parameter. Typically, derived classes of *InstantiatedParameters* follow the naming convention of *InstantiatedTypeParameter*, like *InstantiatedDoubleParameter* or *InstantiatedBooleanParameter*. While not implemented yet, future versions of the ATF might use reflection combined with this standard naming convention to allow a user of the ATF

to plug in new supported types without a recompile of the ATF package itself. Currently, adding a new supported type involves both creating the proper *InstantiatedParameter* derived class, registering it in the ATF source code, and recompiling the ATF libraries in full.

The *InstantiatedParameter* abstract class requires implementors to implement eight methods, described here:

public boolean atMaximum()

This method must determine if the payload is equal to the configured maximum allowable value for the *InstantiatedParameter.*

public Object getPayload()

This method must return the payload held in the *InstantiatedParameter.* The actual object returned must be of type *Type* as indicated by the derived class name *InstantiatedTypeParameter*, where *Type* is the type for which the implementor wishes to add support to the ATF.

public Object incrementPayload() throws IncrementPastMaximumException

This method must increment and return the newly incremented payload. The only requirement that the increment method must follow is that if the payload initially is the minimum configured allowable value for the type under test, successive calls to *IncrementPayload()* will return unique objects as defined by the *.equals()* method until the maximum allowable

value is reached, in which case an exception will be thrown. Some types are more conducive to ordering than others; as long as a unique ordering is defined, the testing will execute properly. Again, though *IncrementPayload* returns an *Object*, the object must be of the proper type for the parameter.

incrementPayload() is used exclusively by the bounded exhaustive testing strategy of the ATF.

public Object resetPayload()

The payload must be reset to the configured minimum allowable value. Along with *incrementPayload()*, this method is only used in the bounded exhaustive testing strategy of the ATF.

public Object randomizePayload()

This method must return a randomly selected object from the range defined by the configured minimum and maximum allowable value for the type. While the ATF assumes a uniform distribution of the random selection, it is not enforced by any means and a properly documented nonuniform distribution is acceptable. This method is used only in using the random test generation strategy of the ATF.

public void setPayload(Object obj) throws IncompatibleParameterException

This method is used to set the payload of the parameter to an arbitrary object. An exception must be thrown when the *obj* parameter is not of the appropriate type for the

*InstantiatedException.*This method is used in bounded exhaustive testing when the starting payload is configured to anything other than the minimum allowable value for the type.

protected double[] serializeToDouble(Object obj) throws IncompatibleParameterException

This method must convert *obj* to an array of *double*s such that *obj* may later be reconstituted unambiguously. This method is used in the translation of *Object*s understood by the software under test to the n-dimensional floating point model used by the ISE input generation strategy. Because of this, not only must the translation be unambiguous, but for best results it must also be created such that geometrically "near" points in the created floating point vector must translate to "near" objects by some definition of "near" understood by the tester of the software. The ISE algorithm also assumes that all objects of some type will translate to a fixed-length floating point array, so storing a linked list of *double*s as a variable-length array of *double*s will in fact cause the ISE algorithm to fail. In cases where the type under test is of variable size, the tester must abstract away portions of the data structure until it can be defined in a fixed-length array.

protected Object unserializeFromDouble(double[] serObj) throws IncompatibleParameterException

This method is the companion to *serializeToDouble()* used by the ISE generation strategy. It must reconstitute a serialized object back into its appropriate instantiated object as defined by the *.equals()* method. If the input floating point array is of an incorrect length for the type under test, the method must throw an IncompatibleParameterException.

Appendix B

Defining Test Suites in the Automated Testing Framework

A test suite in the Automated Testing Framework is defined using an XML configuration file. The DTD of a correctly formed configuration file is as follows:

```
<!--
DTD for TestGenerator Configuration file
-->

<!ELEMENT Configuration (ClassUnderTest, MethodSignatureSuite)>

<!ELEMENT ClassUnderTest (#PCDATA)>

<!ELEMENT MethodSignatureSuite (MethodSignature+)>

<!ELEMENT MethodSignature (Name, Parameter*)>

<!ELEMENT Name (#PCDATA)>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter type CDATA #REQUIRED>
<!ATTLIST Parameter strategy (ise | exhaustive | random) #REQUIRED>
<!ATTLIST Parameter initialvalue CDATA #IMPLIED>
```

```
<!ATTLIST Parameter min CDATA #IMPLIED>
<!ATTLIST Parameter max CDATA #IMPLIED>
<!ATTLIST Parameter count CDATA #IMPLIED>
```

The root element of any configuration file is a Configuration element which contains one element naming the class under test along with a MethodSignatureSuite containing one or more MethodSignatures. One method signature defines one method in ClassUnderTest to be tested. A method signature contains the name of the method to be tested along with a set zero or more Parameters which are the inputs to the method under test. The parameter list must be ordered the same way parameters are ordered in the method as defined by the source code of the method under test. A parameter contains several attributes including the type (*int*,*boolean*, etc.) and the generation strategy to be applied to that parameter. Optionally, the tester may define minimum and maximum values for the parameter, the starting value for the parameter, and the number of tests to generate for that parameter. This final attribute, however, has some unintuitive properties which an understanding of the way the ATF generates tests alleviates. The ATF generates inputs in two phases; in the first phase, all ISE and randomly generated parameters are generated. These values are then held constant while any exhaustive input generation is performed. If two exhaustive parameters exist, the number of tests generated will be the product of the two count attributes. If an ISE parameter is used in conjunction with an exhaustive parameter, the total number of tests will also be the product of the two count attributes. However, if an ISE parameter is used in conjunction with a random parameter, only the maximum of the two count attributes will be generated, not the product (a proper test suite would define identical counts for all ISE and randomly generated parameters).

For demonstration, the following configuration file will be used to explain this DTD:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE Configuration SYSTEM "TestGenConfig.dtd">
<Configuration>
    <ClassUnderTest>Test</ClassUnderTest>
    <MethodSignatureSuite>
        <MethodSignature><Name>testMethod</Name>
        <Parameter type="boolean" strategy="exhaustive"/>
        <Parameter type="byte" strategy="random" count="5" min="0"/>
        <Parameter type="char" strategy="random" count="5"/>
        <Parameter type="java.lang.String" strategy="random" count="1"/>
</MethodSignature>
        <MethodSignature><Name>otherMethod</Name>
        <Parameter type="boolean" strategy="exhaustive"/>
        <Parameter type="boolean" strategy="exhaustive" />
        </MethodSignature>
        <MethodSignature><Name>intISEtest</Name>
        <Parameter type="int" strategy="ise"
count="10000" initialvalue="0" />
        <Parameter type="int" strategy="ise"
count="10000" initialvalue="0" />
        </MethodSignature>
    </MethodSignatureSuite>
</Configuration>
```

In this test suite, three methods will be tested, *Test.testMethod(byte, char, String)*, *Test.otherMethod(boolean, boolean)*, and *Test.intISETest(int, int)*.

A total of ten tests will be generated for *TestMethod*, including five randomly selected *byte*s and *char*s, five empty *String*s (String is not a fully supported type of the ATF), all tested against both possible *boolean* values totaling ten tests.

Four tests will be generated for *otherMethod*, paring all combinations of *boolean*s.

Finally, a total of ten thousand tests ($max(10000, 10000)$)will be generated for *intISEtest* with a seeded relevant input of *intISEtest(0,0)*.

Bibliography

[1] Aynur Abdurazik and Je Outt. Generating test cases from UML specifications. Technical report, George Mason University, March 02 1999.

[2] Jesús S. Aguilar-Ruiz, Isabel Ramos, José Cristóbal Riquelme Santos, and Miguel Toro. An evolutionary approach to estimating software development projects. *Information & Software Technology*, 43(14):875–882, 2001.

[3] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 23(3):228–245, 1983.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 123–133, July 2002.

[5] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[6] Kwok Ping Chan, Tsong Yueh Chen, Fei-Ching Kuo, and Dave Towey. A revisit of adaptive random testing by restrictio. In *COMPSAC*, pages 78–85. IEEE Computer Society, 2004.

[7] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Forgetting test cases. In *COMPSAC*, pages 485–494. IEEE Computer Society, 2006.

[8] Tsong Yueh Chen and Dehao Huang. Adaptive random testing by localization. In *APSEC*, pages 292–298. IEEE Computer Society, 2004.

[9] Tsong Yueh Chen and Yuen-Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering*, 22(2):109–119, February 1996.

[10] Benjamin Cook. Search algorithms for regression test suite minimisation. Technical report, King's College London, September 2006.

[11] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for java. *Softw, Pract. Exper*, 34(11):1025–1050, 2004.

[12] Edsger W. Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, 1969.

[13] G. Dowek, C. Muñoz, and V. Carreño. Provably safe coordinated strategy for distributed conflict resolution. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047*, San Francisco, California, 2005.

[14] Joe W. Duran and Simeon C. Ntafos. A report on random testing. In *ICSE*, pages 179–183, 1981.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

[16] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, September/October 1999.

[17] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Softw. Eng.*, 16(12):1402, December 1990.

[18] Dick Hamlet. When only random testing will do. In *Random Testing*, Johannes Mayer and Robert G. Merkel, editors, pages 1–9. ACM, 2006.

[19] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, J.Marciniak, editor, pages 970–978. Wiley, 1994.

[20] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4), 1970.

[21] Mark Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[23] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[24] D.C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1):62–9, February 1987.

[25] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[26] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.

[27] A. Z. Javed, P. A. Strooper, and G. N. Watson. Automated generation of test cases using model-driven architecture. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[28] GARY T. LEAVENS AND YOONSIK CHEON. Design by contract with JML, 2005. Draft, available from jmlspecs.org.

[29] R. LINGER. Cleanroom process model. *IEEE Software*, 11(2):50–58, March 1994.

[30] RUPAK MAJUMDAR AND KOUSHIK SEN. Hybrid concolic testing. In *ICSE*, pages 416–426. IEEE Computer Society, 2007.

[31] SPIROS MANCORIDIS AND BRIAN S. MITCHELL. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.

[32] DARKO MARINOV, ALEXANDR ANDONI, DUMITRU DANILIUC, SARFRAZ KHURSHID, AND MARTIN RINARD. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[33] DARKO MARINOV AND SARFRAZ KHURSHID. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*, San Diego, CA, 26–29 November 2001. IEEE.

[34] N. METROPOLIS, A. ROSENBLUTH, M. ROSENBLUTH, A. TELLER, AND E. TELLER. Equations of state calculations by fast computing machines. *J. Chem. Physics*, pages 1087–1092, 1953.

[35] N. METROPOLIS AND S. ULAM. The Monte Carlo method. *J. Amer. Statist. Assoc.*, 44:335–341, 1949.

[36] ALEKSANDAR MILICEVIC, SASA MISAILOVIC, DARKO MARINOV, AND SARFRAZ KHURSHID. Korat: A tool for generating structurally complex test inputs. In *ICSE*, pages 771–774. IEEE Computer Society, 2007.

[37] B. P. MILLER, L. FREDRIKSON, AND B. SO. An empirical study of the reliability of unix utilities. *Comm. of the ACM*, 33(12):32, December 1990.

[38] BARTON MILLER, DAVID KOSKI, CJIN PHEOW LEE, VIVEKANANDA MAGANTY, RAVI MURTHY, AJITKUMAR NATARAJAN, AND JEFF STEIDL. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, Computer Science Department, University of Wisconsin, Madison, WI, 1995.

[39] C. MUÑOZ, R. SIMINICEANU, V. CARREÑO, AND G. DOWEK. KB3D reference manual - version 1.a. Technical Report NASA/TM-2005-213769, NASA Langley Research Center, NASA LaRC, Hampton VA 23681-2199, USA, June 2005.

[40] JOHN A. MURPHY AND DAVID COPPIT. Random generation of test inputs for implicitly defined subdomains. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 13, Washington, DC, USA, 2007. IEEE Computer Society.

[41] SIMEON C. NTAFOS. On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Software Eng*, 27(10):949–960, 2001.

[42] S. OWRE, J. M. RUSHBY, AND N. SHANKAR. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607:748–??, 1992.

[43] CARLOS PACHECO, SHUVENDU K. LAHIRI, MICHAEL D. ERNST, AND THOMAS BALL. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007.

[44] D. RICHARDSON, O. O'MALLEY, AND C. TITTLE. Approaches to specification-based testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 86–96, 1989.

[45] PETER SCHMITT, ISABEL TONIN, CLAUS WONNEMANN, ERIC JENN, STÉPHANE LERICHE, AND JAMES J. HUNT. A case study of specification and verification using jml in an avionics application. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 107–116, New York, NY, USA, 2006. ACM.

[46] KEVIN SULLIVAN, JINLIN YANG, DAVID COPPIT, SARFRAZ KHURSHID, AND DANIEL JACKSON. Software assurance by bounded exhaustive testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 133–142, 2004.

[47] MARKOS Z. TSOUKALAS, JOE W. DURAN, AND SIMEON C. NTAFOS. On some reliability estimation problems in random and partition testing. *IEEE Transactions on Software Engineering*, 19(7):687–697, July 1993.

[48] W. VISSER, C. S. PASAREANU, AND S. KHURSHID. Test input generation with Java PathFinder. *Software Engineering Notes*, 29(4):97–107, 2004.

[49] JOACHIM WEGENER AND OLIVER BÜHLER. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, volume 3103 of *Lecture Notes in Computer Science*, pages 1400–1412, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[50] ELAINE J. WEYUKER AND BINGCHIANG JENG. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[51] KRISTINA WINBLADH, THOMAS A. ALSPAUGH, HADAR ZIV, AND DEBRA J. RICHARDSON. An automated approach for goal-driven, specification-based testing. In *ASE*, pages 289–292. IEEE Computer Society, 2006.

VITA

John Alexander Murphy

John Alexander Murphy was born in Camp Springs, Maryland on April 10, 1984. He graduated as a valedictorian from James Madison High School in 2002 and then went to the College of William and Mary where he studied math and computer science. In 2005, he earned is B.S. in Computer Science from William and Mary, where he continued his studies in computer science towards his Master's degree. He now works at Zope Corporation as a software engineer.