

# Random Generation of Test Inputs for Implicitly Defined Subdomains

John A. Murphy, David Coppit  
Department of Computer Science  
The College of William and Mary  
Williamsburg, VA 23185  
jamur2@cs.wm.edu, david@coppit.org

## Abstract

*In traditional random testing, samples are taken from the set of all possible values for the input types. However, for many programs testing effectiveness can be improved by focusing on a relevant subdomain defined implicitly by the program behavior. This paper presents an algorithm for identifying and randomly selecting inputs from implicitly defined subdomains. The algorithm dynamically constructs and refines a model of the input domain and is biased toward sparsely covered regions in order to accelerate boundary identification and uniform coverage.*

*This method has several desirable qualities: (1) it requires no knowledge of the source code of the software being tested, (2) inputs are selected from an approximately uniform distribution across the subdomain, and (3) algorithmic running time overhead is negligible. We present the requirements for a solution and our algorithm. We also evaluate our solution for both an artificial model and a real-world aircraft collision-avoidance program.*

## 1. Introduction

Random testing can be a simple and effective method of detecting software faults. Traditionally, random testing has involved random selection of test inputs from some known input space. For most programs, random testing across the entire space of type-correct inputs is not effective. However, the programmer can often identify a smaller *relevant subdomain* of inputs that are more likely to reveal a fault.

The motivating example we use in this paper is KB3D, an aircraft collision avoidance program [2, 7]. The input space is a pair of multi-dimensional vectors, one for each plane, encoding values such as speed, direction, and position. The entire input space is extremely large, but the relevant subdomain of trajectories that will require a collision-avoidance maneuver is orders of magnitude smaller. Thus, the relevant subdomain is the set of near-miss trajectories that require a collision-avoidance maneuver.

Two key difficulties in performing random testing on such programs is that the relevant subdomain of inputs is both complex in structure and not explicitly defined *a priori*. The subdomain is, however, implicitly defined by the program itself. KB3D, for example, will determine whether two headings are on a collision course and output a message indicating whether an action is necessary.

We present the implicit subdomain exploration (ISE) algorithm, which heuristically generates random test inputs for an implicitly defined relevant subdomain. The algorithm dynamically discovers the boundaries of the subdomain and generates inputs in sparsely covered areas of the input space in order to avoid over-testing previously discovered regions. Performance of the algorithm is linear in the number of inputs generated, and the user can balance the accuracy of the underlying model of the input space with generation performance. Our algorithm generates numerical values—currently  $n$ -dimensional, floating-point inputs. A developer using this algorithm to test software need only provide a few search parameters, a starting input from the relevant subdomain, and an interface to the program under test which classifies inputs as relevant or not.

We evaluate the algorithm for an artificial model as well as the KB3D software. Our artificial model is designed to assess the ability of the algorithm to identify complex input spaces, and we use KB3D to assess performance. KB3D takes a 12-dimensional vector of floating-point numbers as input and has a very complicated input region defined by constraints on multiple values.

Section 2 describes related work. Section 3 presents the concept of relevant subdomains as it pertains to this work. Section 4 explains the inner workings of the ISE algorithm. Section 5 evaluates the performance and effectiveness of the ISE algorithm in artificial and real-world applications, and Section 6 concludes.

## 2. Related Work

Random testing [5] is a well-known black-box testing technique. Although it has deficiencies—namely the diffi-

culty of exercising segments of code guarded by equalities on large types—random testing has several attractive properties. First, test infrastructure is easy to implement. It also provides a large number of test inputs, which can be used to statistically estimate the quality of the software under test [6]. Comparisons of random and partition testing have shown random testing to be potentially more effective [4].

Traditionally, random testing has been performed upon the entire input space of the program. In this paper, we address the problem of random testing on an implicitly defined relevant subdomain of the program. As far as we are aware, we are the first to address this problem. The approach we take is dynamic in nature, exploring the relevant subdomain based on feedback gained from the program’s execution of generated inputs.

Perhaps most closely related to our work is the restricted random testing work of Chan, et. al [1]. The algorithm presented here is complementary to their work, providing an implementation of a generator for inputs within a restricted subdomain of the input space.

A widely used technique for solving problems for models with complex constraints on variables of multiple dimensions is Monte Carlo simulation [3]. For example, one can identify an implicitly defined relevant subdomain using Monte Carlo simulation by randomly choosing a value in the input space and then running the program under test to determine if the input is relevant. Although this technique can effectively identify the subdomain and sample it randomly, it can also result in wasted test effort. Because the technique does not model the underlying subdomain, subdomains that are small relative to the total sample space will result in the Monte Carlo method generating a large number of irrelevant inputs.

### 3. Random Exploration of Input Subdomains

In many cases, programs are written to accept a large number of inputs without failure but only perform significant computation for a small subset of those inputs. We can define the total input space of a program as the cross product of the values of its inputs. We can then define a *relevant subdomain* of the total input space as a subset of these inputs. Typically the relevant subdomain is chosen as a set of inputs that more thoroughly exercise the program, and are therefore more likely to reveal a fault.

#### 3.1. Difficulties in Random Testing

Random generation of input values can be difficult even for a known subdomain of the input space. Imagine, for instance, generating inputs within the unit circle. Naïve approaches are either biased in the generation of inputs, or waste testing time testing inputs outside the relevant sub-

domain. The correct unbiased generation method is to first select an angle from  $(0, 2\pi)$  and a radius of  $\sqrt{z}$ , where  $z$  is a uniformly random selection from the interval  $(0, 1)$ .

Although this example is trivial it demonstrates that unbiased generation of relevant inputs can be nontrivial. For subdomains involving more variables and more complex constraints, naïve approaches to input generation result in test time wasted due to either biased input generation or generation of irrelevant inputs.

The KB3D system demonstrates a second difficulty—the subdomain may be implicitly defined, perhaps only by the program itself. In KB3D we can define the relevant subdomain as “conflict” trajectories that require a collision avoidance maneuver. Conflicts occur when the *minimum separation* is breached, or when an “intruding” craft enters the cylindrical protected zone of another craft. An aircraft trajectory is defined as a six-dimensional vector of its  $x$ ,  $y$ , and  $z$  coordinates, its heading, and its velocity broken down into ground speed and vertical speed components.

Although this subdomain is not explicitly defined, the KB3D implementation implicitly defines it by identifying conflicting inputs. If it detects a conflict, KB3D provides several resolution maneuvers that one or both aircraft may perform to avoid the conflict. The only way to test this second phase of the algorithm is to generate inputs which are determined to be conflicts in the first phase.

Such implicitly defined subdomains create a number of challenges for random testing. First, it is impossible to guarantee a uniform distribution (lack of bias) at all times during the generation, since the universe of relevant values (the subdomain) is unknown. Second, unlike the circle example, there is no clear solution for unbiased generation for implicitly defined subdomains.

Given an implicitly defined subdomain, we must spend some test effort discovering the boundaries. We define *hit rate*, the percent of generated inputs that are relevant, as a metric for measuring this additional test effort.

#### 3.2. Requirements for a Solution

As a result of these difficulties, a heuristic approach must be used that automatically discovers the boundaries of the relevant subdomain while generating random samples from the subdomain. A good heuristic must exhibit several important qualities. First, input generation must have low latency, so that testing can begin as soon as possible, perhaps in parallel with generation. Second, generated inputs must approximate a uniform distribution. This implies that as new regions are revealed, they will tend to be more heavily sampled until their density equalizes with the rest of the known relevant subdomain.

Performance must be good, so that input generation does not dominate test execution time. Our goal is a constant

rate of input generation. The solution must be insensitive to the shape and scale of the input region. It should tolerate complex and nonlinear constraints on the input values. A full solution would discover multiple relevant subdomains, but in this paper we will focus on solving the simpler problem of discovering a single contiguous subdomain. The hit rate should also be as high as possible. A low hit rate indicates time spent validating inputs but not testing them.

Finally, the heuristic must provide a mechanism for the program under test to specify the relevant subdomain, for the case where the program’s implementation provides the only definition of the relevant subdomain. In this situation, the programmer may need to modify the program to provide an indication of whether the input is relevant or not.

## 4. The ISE Algorithm

We now describe the implementation and usage of the implicit subdomain exploration (ISE) algorithm. Space prevents us from describing the algorithm in detail. We focus here on the model of the subdomain, parameters and arguments to the algorithm, and key operations.

The algorithm is designed to explore an implicitly defined relevant subdomain characterized by an  $n$ -dimensional space of floating point values. The algorithm works by identifying the sparsest region and then generating points along a random vector originating from a relevant input in the sparse region. The algorithm then continues selecting sparse regions and generating inputs until the requested number of inputs have been generated.

### 4.1. A Model of Relevant Subdomains

A contiguous portion of the revealed subdomain is modeled by the `Bin` abstraction. Bins model the regions of the input space that they represent. For example, rather than storing all previously generated points in a bin, the bin stores the last generated point and a count of the total number of generated points. Bins also store a *miss count* that we use along with with number of points to estimate the extent to which the bin lies within the relevant subdomain.

These bins collectively represent an approximate decomposition of the relevant subdomain into  $n$ -dimensional rectangular subregions. The `InputSpace` abstraction contains a collection of bins that model the structure of the subdomain as it is discovered. As a result, bins typically are adjacent and have varied sizes, but do not overlap.

For performance, we constrain the total number of bins to a user-provided constant. Therefore, as inputs are generated the algorithm merges bins as necessary. The algorithm must detect which bins can safely be merged without a loss of accuracy while still maintaining a high resolution near subdomain boundaries.

## 4.2. Algorithm Parameters

There are four values that control the search. We already mentioned one, the limit on the number of bins. A higher limit provides a more uniform distribution of inputs at the cost of performance. A second related parameter controls the initial size of bins.

The two other parameters are the scale factor for stepping along the vector, and the maximum number of inputs generated along a vector. Once the algorithm chooses a random direction, it generates points at randomized intervals from the source of the vector. Scale factors higher than 1.0 will cause intervals to grow in size, possibly speeding the discovery of a subdomain boundary. We use the last parameter to prevent excessive generation of inputs along a vector for the case where the vector subdomain is much larger than the increment.

The user also provides a starting input for the search and a callback function that returns true if the argument input is within the relevant subdomain. Once a vector has been chosen, candidate inputs are generated along that vector at increasingly larger distances. As inputs are generated, they are checked for relevancy by calling the callback function.

## 4.3. Identifying Sparse Bins and Bin Merging

The two primary functions of the `InputSpace` type are `findASparseBin()` and `mergeBins()`. The function `findASparseBin()` computes the bin with the lowest density of generated inputs. The boundary of the subdomain may bisect a bin, resulting in an artificially low density. To address this problem, we use the bin counts to approximate the volume of the bin that is within the subdomain, then use that value to compute the density.

The `mergeBins()` algorithm performs bin merging. For each bin and each dimension, the algorithm computes the size of the bin that would be doubled in the direction of the dimension. It then computes the error between the volumes of the enclosed bins and the new larger bin. The bin with the lowest error is then chosen, replacing the enclosed bins.

## 5. Evaluation

### 5.1. Completeness

To evaluate completeness, we use an artificial input region. This region is shaped like a barbell with two circles of radius 5 centered at  $(0, 0)$  and  $(10, 10)$  loosely connected by a narrow rectangular band of width 0.1. Additionally, the circle at  $(10, 10)$  has a circular “hole” of radius 2. The starting point of the ISE algorithm is at the origin. This subdomain demonstrates two properties that any algorithm designed to select uniformly from an arbitrary and unknown

Table 1: Running time of ISE in seconds

Number of Inputs	Barbell	KB3D
10000	0.585	2.839
100000	1.674	15.772
1000000	12.072	1021.2

Table 2: Hit rate of the ISE algorithm

Example	ISE	Monte Carlo
Barbell	0.823	0.362
KB3D	0.898	—

space must exhibit: the ability to discover and traverse narrow regions and the ability avoid obstacles in order to reveal the region in its entirety.

To evaluate the distribution of inputs created by ISE, we compare it against a distribution created using a Monte Carlo simulation. In both cases we used 25,000 points. We divide the subdomain into squares of size 0.5, and compute the difference of the local density within each square between the two distributions.<sup>1</sup>

Over three trials, generating 25,000 points yields an average local density difference of approximately 3.5%. In all cases, 25,000 test cases was sufficient to traverse the narrow passage and discover the full barbell region.

## 5.2. Efficiency

We now examine the efficiency of the ISE algorithm using two metrics. Our analysis of the running time shows that executing the ISE algorithm will likely not be a limiting factor when testing software. Table 1 shows the time taken to generate various amounts of test cases from the barbell and KB3D models. All times are averages from executing three trials as run on a dual 2.5 GHz PowerPC processor with 2.5 GB RAM.

The algorithm runs in approximately linear time, with an increase in the order of magnitude of test cases generated eliciting a corresponding increase in running time. However, the shape of the region under test affects the generation rate. Input dimensions that are unbounded or nearly unbounded will require more bin merges, increasing the running time. The algorithm performs best under regions tightly constrained in all directions, but performs adequately for unconstrained regions. For KB3D, our algorithm explores the 12-dimensional space in a reasonable timeframe, generating a million test cases in approximately 17 minutes. This suggests that the ISE algorithm may be sensitive to the dimensionality of the space under test.

The next metric we study is the hit rate of the algorithm. The ISE algorithm discovers the subdomain experimentally. We found that our algorithm validates many fewer irrelevant inputs compared to a traditional Monte Carlo strat-

<sup>1</sup>We chose 25,000 points to allow the ISE algorithm to find the far region and then equalize the distribution. The size of 0.5 is arbitrary, chosen as simply ten times smaller than the subdomain size.

egy. In Table 2, we present a comparison between hit rates of the ISE algorithm and Monte Carlo simulation. Again, all ratios are averages from three trials.

The ISE algorithm wastes little effort testing points outside of the relevant subdomain. For pathological cases such as KB3D, in which the relevant subdomain is orders of magnitude smaller than the full domain, it is improbable that a Monte Carlo strategy will ever generate a valid test case. Note that for the barbell example, we needed the bounding box in order to do Monte Carlo generation, but we did not need this information for the ISE algorithm.

## 6. Conclusion

This paper has presents a new method for the random selection of points from an input space whose size and structure is unknown. This allows testers to focus random testing effort on subdomains defined implicitly by the program itself. Our algorithm only identifies contiguous subdomains, but can be run multiple times with different starting points. The ISE algorithm is also designed to test software that has numerical inputs, produces well-defined outputs, and is guaranteed to terminate. More generally, this paper has explored the concept of a *relevant subdomain* as it pertains to random testing. The ability to recognize and classify inputs as relevant is of paramount importance if one wishes to improve the cost-effectiveness of an automated random testing strategy.

## References

- [1] Kwok Ping Chan, Tsong Yueh Chen, Fei-Ching Kuo, and Dave Towey. A revisit of adaptive random testing by restriction. In *COMPSAC*, pages 78–85. IEEE, 2004.
- [2] 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.
- [3] George S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer-Verlag, 1996.
- [4] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402, December 1990.
- [5] Richard Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Soft. Eng.* Wiley, 2nd edition, 2005.
- [6] S. Mankefors, R. Torkar, and A. Boklund. New quality estimations in random testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 468–78, Denver, Colorado, 17–20 November 2003. IEEE.
- [7] 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.