A Learning-based Approach to Unit Testing of Numerical Software

Karl Meinke and Fei Niu

Royal Institute of Technology, Stockholm
karlm@csc.kth.se niu@csc.kth.se

Abstract. We present an application of learning-based testing to the problem of automated test case generation (ATCG) for numerical software. Our approach uses n-dimensional polynomial models as an algorithmically learned abstraction of the SUT which supports n-wise testing. Test cases are iteratively generated by applying a satisfiability algorithm to first-order program specifications over real closed fields and iteratively refined piecewise polynomial models.
We benchmark the performance of our iterative ATCG algorithm against iterative random testing, and empirically analyse its performance in finding injected errors in numerical codes. Our results show that for software with small errors, or long mean time to failure, learning-based testing is increasingly more efficient than iterative random testing.

1 Introduction

For black-box specification-based testing, (see e.g. [11]) an important scientific goal is *automated test case generation* (ATCG) from a formal requirements specification, by means of an efficient and practical algorithm. A general approach common to several tools is to apply a satisfiability algorithm to a formal specification and/or a code model in order to generate counterexamples (test cases) to correctness. For an SUT that involves floating point computations one important problem is therefore to find an an expressive formal requirements language suitable for modeling floating point requirements together with an efficient satisfiability algorithm for generating floating point counterexamples. One possibility is to consider the first-order language and theory of real closed fields for which satisfiability algorithms have been known since [14].

To achieve high coverage levels it is important to go beyond individual test case generation towards *iterative testing techniques* that can iteratively generate a large number of high quality test cases in a reasonable time. In [10] we identified one such iterative heuristic for ATCG that we term *learning-based testing* (LBT). Our earlier work concerned black-box unit testing of numerical programs based on:

(i) simple learning algorithms for 1-dimensional problems,

(ii) simple requirements specifications which are quantifier free first-order formulas, and

(iii) an elementary satisfiability algorithm based on algebraic root solving for cubic 1-dimensional polynomials.

In this paper we present a systematic and more powerful extension of LBT that is suitable for black-box testing of complex numerical software units, including high-dimensional problems by means of n-wise (e.g. pairwise) testing. We apply n-dimensional polynomial learned models that can support n-wise testing and thus tackle the dimensionality problem associated with software units. We generalise our previous learning algorithm to support n-dimensional piecewise polynomial models using non-gridded data. This step also tackles the dimensionality problem on the SUT level. Finally, we use the Hoon-Collins cylindric algebraic decomposition (CAD) algorithm for satisfiability testing (see e.g. [1]). This is a powerful satisfiability algorithm for first order formulas over the language of real closed fields. Thus we greatly expand the complexity of the requirements specifications that our tool can deal with.

It is natural to question the achieved quality of the test cases generated by any new TCG method. In the absence of a theoretical model of efficiency, we have benchmarked the quality of our learning-based ATCG empirically, by comparing its performance with that of an iterative random test case generator. This was the only other iterative ATCG method for floating point computations to which we had any access. Since iterative random testing (IRT) is closely related to measures of mean time to failure (MTF), our benchmarking results have a natural and intuitive interpretation.

To carry out this performance comparison systematically, it was necessary to avoid the experimental bias of focussing on just a small number of specific SUTs and specific requirements specifications. To generate the largest possible amount of comparative data we automated the synthesis of a large number of numerical SUTs, their black-box specifications and their mutations. In this way we compared the performance of learning-based and iterative random testing over tens of thousands of case studies. Our average case results over this data set demonstrate that when mutated errors have small effects (or equivalently when the MTF of the mutated SUT is long) then learning based testing is increasingly superior to random testing.

The structure of this paper is as follows. In Section 1.1 we review some related approaches to unit testing of numerical software and ATCG, found in the literature. In Section 2, we discuss the general principles of learning-based testing. In Section 3, we present our learning-based testing algorithm for numerical programs. In Section 4, we describe our evaluation technique, and present the results of evaluation. In Section 5 we draw some conclusions from our work.

1.1 Related Work

There are two fields of scientific research that are closely related to our own. On the one hand there is a small and older body of research on TCG for scientific computations. On the other hand, there is a more recent growing body of research on ATCG methods obtained by combining satisfiability algorithms

with computational learning. Finally, there is an extensive mathematical theory of polynomial approximation, see for example [12].

Unit Testing for Scientific Software It is widely recognised that the problem of testing scientific software has received relatively little attention in the literature on testing. The existing literature seems focussed either on manual techniques for TCG, or ATCG for individual test cases. Besides iterative random testing, we are not aware of any other approach to iterative ATCG for numerical programs.

The intrinsic unreliability of many existing libraries of scientific code for the earth sciences has been empirically studied at length in [6] and [7], which cite static error rates in commercial code of between 0.6% and 20% depending upon the type of program error.

The correctness of scientific library codes for metrology is considered in [4]. Here the authors consider black-box unit testing of well specified computations such as arithmetic mean and deviation, straight-line and polynomial regression. They apply 1-wise testing where all fixed parameter values and one variable parameter value are chosen from sets of reference data points that are manually constructed to span a given input profile. Reference values are then algorithmically computed using numerical codes which the authors posit as providing reliable benchmarks. The actual outputs produced by the SUT are compared with these reference values using a composite performance measure. This measure is more sophisticated than an individual error bound as it can account for the degree of difficulty of the chosen reference data set. The approach seems more oriented towards measuring overall code quality while our own approach focusses on finding individual coding errors quickly. Nevertheless, [4] successfully highlights problems of numerical stability occurring in a variety of well known scientific libraries and packages including NAG, IMSL, Microsoft Excel, LabVIEW and Matlab, thus confirming the real problem of correctness even among simple and widely used scientific codes.

The method of manufactured solutions (MMS) presented in [13] provides a theoretically well-founded and rigorous approach for generating a finite set of test cases with which to test a numerical solver for a PDE (usually non-linear). The test cases are constructed by analysis of the underlying mathematical model as a non-linear system operator $L[u(x,\ y,\ z,\ t)]$ to produce a finite set of analytical solutions for different input parameter values. Like our own approach (but unlike [4]) MMS does not make use of any benchmark or reference codes for producing test cases. This method is shown to be effective at discovering order-of-accuracy errors in fault injected codes in [8]. The main weakness of MMS is its restriction to a specific class of numerical programs (PDE solvers). By contrast, the generality of our requirements language (first-order logic) and our modeling technique (piecewise polynomials) means that by general results such as the Weierstrass Approximation Theorem, we can perform ATCG for a much larger class of programs and requirements.

ATCG using Computational Learning Within the field of verification and testing for reactive systems, the existence of efficient satisfiability algorithms for automata and temporal logic (known as model checkers) has made it feasible to apply computational learning to ATCG using similar principles to our own. This approach is known as *counterexample guided abstraction refinement* (CEGAR). Some important contributions include [3], [2] and [5]. The basic principles of CEGAR are similar to those we outline in Section 2, though our own research emphasizes *incremental learning algorithms* and related *convergence measures* on models both to guide model construction and to yield abstract black-box coverage measures. While existing research on CEGAR emphasizes reactive systems, temporal logic and discrete data types, our work presented here considers procedural systems, first order logic and continuous (floating point) data types. However, the success of CEGAR research strongly suggests to generalize this approach to other models of computation, and our work can be seen as such a generalization.

2 Learning-based Testing

The paradigm for ATCG that we term *learning-based testing* is based on three components:

(1) a (black-box) *system under test* (SUT) S,

(2) a *formal requirements specification Req* for S, and

(3) a *learned model* M of S.

Now (1) and (2) are common to all specification-based testing, and it is really (3) that is distinctive. Learning-based approaches are *heuristic iterative methods* to search for and automatically generate a sequence of test cases until either an SUT error is found or a decision is made to terminate testing. The heuristic approach is based on *learning a black-box system using tests as queries*.

A learning-based testing algorithm should iterate the following five steps:

(Step 1) Suppose that n test case inputs $i_1, \ldots, i_n$ have been executed on S yielding the system outputs $o_1, \ldots, o_n$. The n input/output pairs $(i_1, o_1), \ldots, (i_n, o_n)$ can be synthesized into a learned model M_n of S using an *incremental learning algorithm*. Importantly, this synthesis step involves a process of *generalization* from the data, which usually represents an incomplete description of S. Generalization gives the possibility to predict as yet unseen errors within S during Step 2.

(Step 2) The system requirements *Req* are satisfiability checked against the learned model M_n derived in Step 1. This process searches for counterexamples to the requirements.

(Step 3) If several counterexamples are found in Step 2 then the most suitable candidate is chosen as the next test case input i_{n+1}. One can for example attempt to identify the most credible candidate using theories of *model convergence* (see Section 3.3) or *approximate learning*.

(Step 4) The test case i_{n+1} is executed on S, and if S terminates then the output o_{n+1} is obtained. If S fails this test case (i.e. the pair $(i_{n+1},\ o_{n+1})$ does not satisfy *Req*) then i_{n+1} was a *true negative* and we proceed to Step 5. Otherwise S passes the test case i_{n+1} so the model M_n was inaccurate, and i_{n+1} was a *false negative*. In this latter case, the effort of executing S on i_{n+1} is not wasted. We return to Step 1 and apply the learning algorithm once again to $n+1$ pairs $(i_1,\ o_1),\ \ldots,\ (i_{n+1},\ o_{n+1})$ to synthesize a refined model M_{n+1} of S.

(Step 5) We terminate with a true negative test case $(i_{n+1},\ o_{n+1})$ for S.

Thus an LBT algorithm iterates Steps $1 \ldots\ 4$ until an SUT error is found (Step 5) or execution is terminated. Possible criteria for termination include a bound on the maximum testing time, or a bound on the maximum number of test cases to be executed. From the point of view of abstract black-box coverage, an interesting termination criterion is to choose a minimum value for the convergence degree d_n of the model M_n as measured by the difference $|M_n| - |M_{n-1}|$, according to some norm $|.|$ on models. We can then terminate when M_n achieves this convergence degree.

This iterative approach to TCG yields a sequence of increasingly accurate models M_0, M_1, M_2, ..., of S. (We can take M_0 to be a minimal or even empty model.) So, with increasing values of n, it becomes more and more likely that satisfiability checking in Step 2 will produce a true negative if one exists. Notice if Step 2 does not produce any counterexamples at all then to proceed with the iteration, we must construct the next test case i_{n+1} by some other method, e.g. randomly.

3 Algorithm Description

In this section we show how the general framework of learning-based testing, described in Section 2, can be instantiated by: (i) piecewise polynomial models M_i, (ii) incremental learning algorithms, and (iii) an implementation of the CAD algorithm in Mathematica$^{\text{TM}}$, used as a satisfiability checker. The resulting *learning-based ATCG* can be used to automatically unit test numerical programs against their requirements specifications expressed as first order formulas over the language of real closed fields.

3.1 Piecewise Polynomial Models

The learned models M_i that we use consist of a set of overlapping local models, where each local model is an n-dimensional and d-degree polynomial defined over an n-dimensional sphere of radius r over the input/output space. Since $(d+1)^n$ points suffice to uniquely determine an n-dimensional degree d polynomial, each local model has $(d+1)^n$ such points. One such point $c \in \mathbf{R}^n$ is distinguished as the *centre point*. The *radius* r of a local model is the maximum of the Euclidean distances from the centre point to each other point in that model. Figure 1 illustrates this for the simple case $n = 1$ and two overlapping 1-dimensional

cubic polynomial models of degree 3. With n-dimensional polynomials we can automate n-wise testing, e.g. for $n = 2$ we obtain pairwise testing. Concretely, a model M_i is represented as an array of *LocalModel* objects.

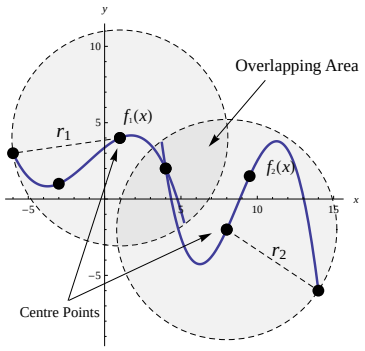


Fig. 1. Two cubic local models $f_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i,\ i = 1, 2$

We use a *non-gridded approach* to selecting the centre point for each local model. This means that the centre point c can take any value $x \in \mathbf{R}^n$ and is not constrained to lie on a vertex of an n-dimensional grid of any specific mesh size. Non-gridded modeling helps avoid an exponential blowup in the number of grid vertex points as the model dimension n increases. It also allows us to choose test case values with complete freedom. So test cases can cluster densely within areas of suspected errors, and sparsely within areas of likely correct behavior. This is one reason why LBT exceeds the performance of iterative random testing since the latter uniformly samples the search space of test cases.

3.2 A Learning-based Testing Algorithm

We now give a concrete instantiation of the abstract learning-based ATCG, outlined in Section 2, for numerical programs. This combines incremental learning methods for the piecewise polynomial models described in Section 3.1 together with an implementation of the CAD algorithm for satisfiability checking such polynomial models against requirements specifications. As is usual for procedural programs, a requirements specification for a numerical program S under test is expressed as a *Hoare triple* (see e.g. [9])

$$\{pre\}S\{post\},$$

where the *precondition pre* and *postcondition post* are first order formulas over the language of real closed fields. Thus *pre* and *post* describe constraints on the input and output floating point variables of S at the start and end of computation. A true negative (or failed) test case is any test case which satisfies the triple $\{pre\}S\{\neg post\}$ under the usual partial correctness interpretation of Hoare

triples. Our algorithm automatically searches for at least one true negative test case where S terminates. (We handle non-termination, should it arise, with a simple time-out.) This approach is easily extended to multiple negatives if these are required.

Before satisfiability checking can be applied, it is necessary to have at least one local polynomial model. Therefore the ATCG procedure given in Algorithm 1 below is divided into an *initialisation* phase and a *learning-based* phase. During the initialisation phase (lines 1-10), the minimum number $(d+1)^n$ of test cases necessary to build one local model, is randomly generated (line 4). Each such test case t is executed on the SUT S and the output is stored (line 6). During the iterative learning-based phase (lines 11-30), on each iteration we try to generate a new test case either through: (i) performing satisfiability checking on the learned model (line 16) or, (ii) random test case generation (line 22). Whenever a new test case is generated and executed, the result is added to the model (line 26). Then the local models nearest the test case are updated and refined using this test case (line 28).

Note that Algorithm 1 makes two API calls to the Mathematica kernel[1]. Line 16 of Algorithm 1 makes a call to the Mathematica kernel in order to satisfiability check the formula $pre \wedge \neg post$ against the local model m, for each of the C best converged local models. The kernel function *FindInstance*(...) is the Mathematica implementation of the Hoon-Collins CAD algorithm. If a satisfying variable assignment to $pre \wedge \neg post$ over m exists then this kernel call returns such a variable assignment. In the case that no counterexample is found among the C best converged local models, then line 22 of Algorithm 1 makes a call *FindInstance(M[random()], pre)* to the same kernel function to find a satisfying variable assignment over a randomly chosen local model for precondition pre.

As shown in Algorithm 1, an array M of *LocalModel* objects is maintained. We use $M[i]$ for $0 \leq i < length(M)$ to denote the i-th element of M and $M[i:j]$ for $0 \leq i \leq j < length(M)$ to refer to the subinterval of M between i and $j-1$, inclusive.

3.3 Learning Local Models

The two subprocedures *LearnModel* and *UpdateModels* called in Algorithm 1 are detailed in Algorithms 2 and 3. These implement a simple incremental learning method along the general lines described in Step 1 of Section 2. Algorithm *LearnModel* infers one new local model using the newly executed test case t on the SUT. We use a simple linear parameter estimation method in line 6. The use of more powerful non-linear estimation methods here is an important topic of future research. Algorithm *UpdateModels* updates all the other existing local models using t and sorts the C best converged models.

In Algorithm 2 (line 6), model learning is performed using a linear parameter estimation method. The Mathematica kernel function *LinearSolve*(*c, b*) is used to find a vector of $(d+1)^n$ coefficients satisfying the matrix equation $c \cdot x =$

[1] We use Mathematica™ version 7.0 running on a Linux platform.

Algorithm 1: Learning-BasedTesting

Input:
1. $S(t : TestCase)$ - the SUT expressed as a function
2. $pre, post$ - pre and postcondition for the SUT
3. $d : int$ - the maximum approximation degree of *poly*
4. $max : int$ - the maximum number of tests
5. $C : int$ - the maximum number of model checks for each test

Output: `ErrorFound` or `TimeOut` if no error was found

```
   // Initialisation phase
 1 T, M ← ∅ // Initialise set T of test cases and array M of models
 2 n ← input dimension of S
 3 support_size ← pow(d + 1, n)
 4 T ← set of support_size randomly generated test cases
 5 foreach t in T do
 6     t.output ← S(t) // run S on t
 7     if t violates post then
 8         return ErrorFound + t.toString()
 9 foreach TestCase t in T do
10     M.add(LearnModel(T, t, support_size))
   // Learning-based phase
11 count ← 0
12 while count < max do
13     t ← null
14     for i ← 0 to min(C, length(M)) do
15         m ← M[i]
16         t ← FindInstance (m, pre ∧ ¬post)
17         if m.converg < ϵ then
               // Delete converged local model
18             M.delete(m)
19         if t is NOT null then
20             break
21     if t is null then
22         t ← FindInstance (M[random()], pre)
23     t.output ← S(t) // run S on t
24     if t violates post then
25         return ErrorFound + t.toString()
26     M.add(LearnModel(T, t, support_size)) // cf. Algorithm 2
27     T.add(t)
28     M ← UpdateModels(M, t, support_size, C) // cf. Algorithm 3
29     count ← count + 1
30 return TimeOut
```

b, for the approximating d degree polynomial function in n variables. Note, if the call to *LinearSolve* fails, then temporarily we have no local model around $m.centrePoint$, but this can be corrected later by calls to Algorithm 3. Also note in Algorithm 2 that without any previous model history, we are not yet able to compute the convergence value of a newly constructed local model (as we do in Algorithm 3). Hence convergence is initialized to 0.0 (line 7). This forces the new local model to have the minimum possible convergence value, so it has the possibility to be satisfiability checked during the learning-based phase even though its convergence value is undefined.

In Algorithm 3, when updating a local model, a *Monte-Carlo method* (lines 6, 7) is used to efficiently approximate the convergence value obtained using the integral norm L_1 on bounded polynomials. Each time local models are updated, the first C best converged local models are randomly swapped to the first C positions of the array M, provided C is greater than or equal to the length of M (line 10, 11). This, together with line 14 in Algorithm 1, implements the prioritisation Step 3 of the abstract LBT algorithm of Section 2. In practice, the value of C is empirically determined. A higher value of C can probably take better advantage of model checking while it slows down the speed of learning-based testing if model checking is time consuming.

Algorithm 2: LearnModel

Input:
1. T - the set of all executed test cases
2. $t : TestCase$ - a newly executed test case
3. $support\_size : int$ - the number of test cases needed to build one local polynomial model

Output: a new local model with centre t

1 $m \leftarrow new\ LocalModel()$
2 $m.centrePoint \leftarrow t$
3 $m.localPoints \leftarrow$ from T pick $support\_size$ test cases nearest to t
4 $m.radius \leftarrow$ maximum Euclidean distance between t and each data point of $m.localPoints$
5 Form the matrix equation $c \cdot x = b$ from $m.localPoints$
6 $m.poly \leftarrow$ `LinearSolve`(c, b)
7 $m.converg \leftarrow 0.0$
8 **return** m

4 Experimental Evaluation

4.1 Construction of SUTs, Specifications and Mutations

We wished to benchmark the performance of LBT against another iterative ATCG method for floating point computations. The simplest and most obvious

Algorithm 3: UpdateModels

Input:
1. M - array of all local models obtained so far
2. $t : TestCase$ - a newly executed test case
3. $support\_size : int$ - cf. Algorithm 2
4. $C : int$ - cf. Algorithm 1

foreach *LocalModel* m *in* M **do**
  if t *inside* m **then**
    $T \leftarrow m.localPoints.add(t)$
    $\hat{t} \leftarrow m.centrePoint$
    $\hat{m} \leftarrow \texttt{LearnModel}(T, \hat{t}, support\_size)$
    randomly pick N test cases $t_1, t_2, \ldots t_N$
    $\hat{m}.converg \leftarrow \frac{\sum_{i=1,\ldots,N} |m.poly(t_i) - \hat{m}.poly(t_i)|}{N}$
    $m \leftarrow \hat{m}$
if $C < length(M)$ **then**
  Partially sort M to ensure $M[0:C]$ are linearly ordered by convergence
  Randomly permute $M[0:C]$
else
  Randomly permute M
return M

candidate for comparison was iterative random testing, which is fairly easy to implement when requirements specifications are simple. This comparison has the advantage that iterative random testing can be viewed as a Monte Carlo method to estimate the mean time to failure (MTF) of an SUT over an equiprobable distribution of input values. Our experiments confirmed that this estimated MTF value was inversely proportional to the size of injected mutation errors, as one would expect.

In order to obtain the largest possible data set of performance results, we used random generators to construct the numerical SUTs, their first-order logic specifications and their mutations. These also allowed us to perform experiments in a controlled way, in order to more accurately assess factors that influence the performance of our ATCG. The random numerical program generator (RPG) and specification generator (SG) were used to generate hundreds of SUTs with their specifications and introduce thousands of mutations. Iterative random testing was also repeated hundreds or thousands of times on each SUT, until the estimated MTF value appeared well converged.

Random Numerical Program Generation (RPG) To randomly generate a numerical program as an SUT, the RPG divides a global n-dimensional input space into subspaces at random. Within each of these randomly chosen subspaces, the RPG generates an n-dimensional polynomial surface of random shape (i.e. coefficients) and degree. We allowed for much higher degrees in such SUT models than in the learned models of Section 3.1 in order to ensure that the

learning problem would be non-trivial. When a random SUT was generated, we then mutated it randomly to inject faults by changing the shapes or positions of the polynomial surfaces in one or more subspaces. Figure 2 gives an example of a randomly generated SUT for one dimensional input and six randomly chosen subspaces. To mutate the SUT, we simply regenerate different curves over some of the same subspaces, and this is also shown in Figure 2. The ratio of the total size of the mutated subspaces to the total size of all subspaces represents the *percentage error size*. Controlling this percentage error size experimentally, was the key to understanding the relative performance of LBT and iterative random testing. For example in Figure 2, the percentage error size is 50%.

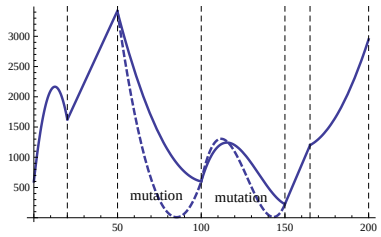


Fig. 2. A randomly generated SUT: 6 subspaces of which 2 are mutated

Random Specification Generation (SG) Since for evaluation purposes a large number of SUTs were automatically randomly generated, it was necessary to automatically generate their requirements specifications too. At the same time it was necessary to ensure that the requirement generated for each SUT was semantically correct in its unmutated state. It is well known (see [1]) that the logical complexity of requirements formulas has an effect on the efficiency of satisfiability checking, and hence on LBT as a whole. To explore the effects of this complexity, we studied the relative performance of LBT and IRT against two different logical types of requirements specifications.

Let S be a randomly generated SUT with k subspaces and 1-dimensional input $S = f_1(x),\ f_2(x),\ \ldots,\ f_k(x)$. Then for both types of requirements specifications, the same precondition was used. We call this an *interval bound precondition* on the input variable x of the form

$$pre(S) \equiv c_1 \leq x \leq c_{k+1}.$$

Here the input interval $[c_1,\ c_{k+1}]$ for an SUT is divided into k subintervals $[c_i,\ c_{i+1}]$ for $1 \leq i \leq k$ by the boundary values $c_1, c_2, \ldots, c_k$, and $f_i(x)$ describes the behaviour of S over the i-th subinterval $[c_i,\ c_{i+1}]$.

On the other hand two different types of postconditions could be generated: we call these *equational* and *inequational postconditions*.

For the same SUT S, its *equational postcondition* is a formula of the form:

$$eq\_post(S) \equiv \bigwedge_{i=1,\ldots,k} (\ c_i \leq x < c_{i+1} \ \Rightarrow \|f_i(x) - m_i(x)\| < \epsilon \ )$$

where $m_i(x)$ describes the mutation of $f_i(x)$ (if any) over the i-th subinterval. Intuitively, $eq\_post(S)$ asserts that the function of the mutated SUT is equal to the function of the original SUT S, to within an absolute tolerance ϵ.

The *inequational postcondition* for S is a formula of the form:

$$ineq\_post(S) \equiv \bigwedge_{i=1,\ldots,k} (\ lower\_bound < f_i(x) \ )$$

Intuitively, $ineq\_post(S)$ asserts that all values of each function $f_i(x)$ lie above a constant lower bound. The value of *lower_bound* is randomly chosen so that this postcondition is semantically correct for the unmutated (correct) program S.

These preconditions and two types of postcondition generalise in an obvious way to n-dimensional input for $n \geq 2$.

4.2 Results and Analysis

Having set up random generators for numerical programs and their specifications, we proceeded to generate a large number of SUT/specification pairs, and mutate these SUTs to achieve different percentage error sizes within the range 10% to 0.01%. We then measured the minimum number of test cases using LBT and IRT needed to find the first true negative in each mutated SUT. This measurement was chosen since for IRT it provides an estimate of the mean time to failure (MTF) of the mutated SUT under an equiprobable input distribution. (We can view IRT as a Monte Carlo algorithm to estimate MTF.) To deal with the stochastic behavior of IRT, this value was averaged out over many runs until a well converged mean value had emerged. We then compared the ratio of these two measurements, and averaged out this figure over many different SUTs and many different mutations all of the same percentage error size. The results of our experiments are given in Figure 3 which illustrates the relative performance of LBT and IRT as the percentage error size is reduced. The x-axis expresses the percentage error size (c.f. Section 4.1) on a logarithmic scale. The y-axis gives the ratio IRT/LBT of the average number of IRT test cases divided by the average number of LBT test cases. Note that Figure 3 shows two curves, one for testing SUTs against equational specifications and one for testing SUTs against inequational specifications. Also note that above an error size of 10%, both curves converge rapidly to $y = 1.0$, which explains our chosen range of error sizes.

The two distinct curves in Figure 3 clearly indicate that relative performance of LBT is influenced by the logical complexity of specifications, as we expected. However, the shapes of both curves are similar. Both curves show that as the

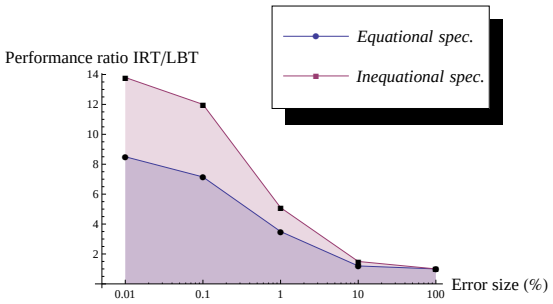


Fig. 3. Relative performance of IRT and LBT

percentage error size decreases (or equivalently the MTF of the mutated SUT increases) the efficiency of LBT over IRT increases. Since the x-axis is logarithmic, this improvement in relative performance seems approximately exponential in the percentage size of errors.

4.3 A Concrete Case Study: Bubblesort

The statistical results of Section 4.2 may appear somewhat abstract, since the thousands of randomly generated case studies do not correspond to specific well known algorithms. Therefore, we complement this statistical analysis with a concrete case study.

```
1  class BubbleSort {
2      public void sort(double[] a) {
3          for (int i = a.length; --i >= 0; ) {
4              boolean flipped = false;
5              for (int j = 0; j < i; j++ ) {
6                  // Mutated from "if (a[j] > a[j+1]) {"
7                  if (a[j] - N > a[j+1]) {
8                      double T = a[j];
9                      a[j] = a[j+1];
10                     a[j+1] = T;
11                     flipped = true;
12                 }
13             }
14             if (!flipped) {
15                 return;
16             }
17         }
18     }
19 }
```

Fig. 4. Bubblesort algorithm with mutation

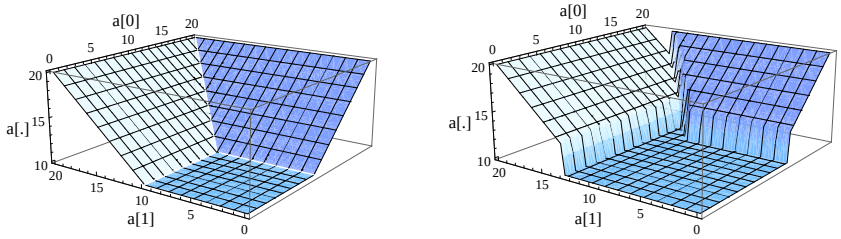
Figure 4 presents the familiar Bubblesort algorithm for an array of floating point numbers. This algorithm represents a typical high dimensional problem,

since the input (and hence output) array size is usually rather large. In line 7 we introduce a mutation into the code via a parameter N. This particular mutation was mainly chosen to evaluate the quality of test cases, since it allows us to control the percentage error size of the mutation. Despite the high dimension of this SUT computation, pairwise LBT testing can find the mutation error fairly quickly. Figure 5 illustrates[2] why this is so. Taking a 2-dimensional polynomial model on any output array value, Figure 5(a) shows that the SUT itself can be modeled quite easily. Furthermore Figure 5(b) shows that the mutated SUT can also be modeled with just a little more effort, since large regions of this model are again essentially simple. A suitable requirement specification for this code is just to assert that the output array is linearly ordered:

$$\{ \bigwedge_{i=0}^{a.length-1} MIN < a[i] < MAX \} \; BubbleSort \; \{ \bigwedge_{i=0}^{a.length-2} a[i] \leq a[i+1] \}$$

where MIN and MAX are simply the lower and upper bounds of input values.

As in Section 4.2, we can measure the minimum number of test cases required by LBT and IRT to find the first true negative in the mutated SUT, against the above requirement specification. Our results show that on average (since IRT has a stochastic performance) LBT is 10 times faster than IRT at uncovering the mutation error.



(a) Model segment of original SUT (b) Model segment of mutated SUT

Fig. 5. Modeling the Bubblesort algorithm with and without mutation

5 Conclusion

We have presented a systematic and powerful extension of the learning-based testing (LBT) approach to iterative ATCG introduced in [10]. We have compared the performance of LBT against the results of iterative random testing over a large number of case studies. Our results clearly demonstrate that LBT,

[2] Note that the grid structure in Figures 5(a) and 5(b) is an artifact of the Mathematica graphics package, the models themselves are non-gridded.

while never worse than iterative random testing, can be significantly faster at discovering errors. Future research will also consider non-linear models and learning algorithms for floating point data types. More generally, we also need to consider the problem of learned models, learning algorithms and satisfiability checkers for other data types besides floating point, in order to increase the range of applicability of our testing methods.

Acknowledgement

We gratefully acknowledge financial support for this research from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620.

References

1. B. F. Caviness and J. R. Johnson. *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer Verlag, 1998.
2. Pankaj Chauhan, Edmund M. Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Proc. 14th International Conference On Formal Methods in Computer-Aided Design (FMCAD02)*, 2002.
3. Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. Sat-based abstraction refinement using ilp and machine learning. In *Proc. 21st International Conference On Computer Aided Verification (CAV'02)*, 2002.
4. M.G. Cox, P.M. Harris, E.G. Johnson, P.D. Kenward, and G.I. Parkin. Testing the numerical correctness of software. Technical Report CMSC 34/04, National Physical Laboratory, Teddington, January 2004.
5. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.
6. L. Hatton and A. Roberts. How accurate is scientific software? *ACM Transactions on Software Engineering*, 20(10):786–797, 1994.
7. Les Hatton. The chimera of software quality. *Computer*, 40(8):104, 102–103, 2007.
8. P. Knupp and K. Salari. *Verification of Computer Codes in Computational Science and Engineering.* CRC Press, 2002.
9. Jacques Loeckx and Kurt Sieber. *The foundations of program verification (2nd ed.).* John Wiley & Sons, Inc., New York, NY, USA, 1987.
10. Karl Meinke. Automated black-box testing of functional correctness using function approximation. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–153, New York, NY, USA, 2004. ACM.
11. Robert M. Poston. *Automating Specification-Based Software Testing.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.
12. M. Reimer. *Multivariate Polynomial Approximation.* Birkhauser Basel, October 2003.
13. P.J. Roache. Building pde codes to be verifiable and validatable. *Computing in Science and Engineering*, pages 30–38, September/October 2004.
14. A. Tarski. *Decision Method for Elementary Algebra and Geometry.* Univ. of California Press, 1951.