Uncertainty-Driven Black-Box Test Data Generation

Neil Walkinshaw
University of Leicester
Leicester, UK

Gordon Fraser
University of Sheffield
Sheffield, UK

***Abstract*—We can never be *certain* that a software system is correct simply by testing it, but with every additional successful test we become less *uncertain* about its correctness. In absence of source code or elaborate specifications and models, tests are usually generated or chosen randomly. However, rather than randomly choosing tests, it would be preferable to choose those tests that decrease our uncertainty about correctness the most. In order to guide test generation, we apply what is referred to in Machine Learning as "Query Strategy Framework": We infer a behavioural model of the system under test and select those tests which the inferred model is "least certain" about. Running these tests on the system under test thus directly targets those parts about which tests so far have failed to inform the model. We provide an implementation that uses a genetic programming engine for model inference in order to enable an uncertainty sampling technique known as "query by committee", and evaluate it on eight subject systems from the Apache Commons Math framework and JodaTime. The results indicate that test generation using uncertainty sampling outperforms conventional and Adaptive Random Testing.**

I. Introduction

Testing software components without access to source code or hand-crafted models is challenging because there is no guidance for the selection of test inputs. Selection is invariably guided by intuition or, if automated, by random or quasi-random input generation algorithms [8], [11], [16]. Left to chance alone, random test sets can easily fail to expose facets of software behaviour that depend upon specific input characteristics. Furthermore it can become exceedingly difficult to reason about the adequacy of a randomly-generated test set, especially for non-numerical programs without an operational profile [16].

Recently, several "Learning-Based Testing" (LBT) techniques have emerged [13], [14], [27], [33] that aim to address these limitations. LBT techniques are based on the idea, first espoused by Weyuker [34] and Budd and Angluin [7], that there is a natural duality between inductive model inference and software testing. The former seeks to infer a general model of behaviour for a system from an incomplete sample of observations of its behaviour. The latter seeks to identify the smallest possible set of observations that are required to expose the full range of behaviour. Although the ultimate purposes are different, both are bound by an intrinsic challenge: establishing the link between the often infinite range of observable behaviour of a system and a finite sample of observations (or vice versa).

LBT techniques seek to exploit this duality by using Machine Learning algorithms to infer input/output models from test executions. These models can then be used to derive new test cases. The rationale is that this ought to form a virtuous loop (or, to adopt Popper's terminology, a cycle of "conjecture and refutation" [26]) where the inferred models become increasingly detailed and accurate, and thereby drive the test generation to produce increasingly rigorous test sets.

The step of generating new test inputs from an inferred model is especially important. New test inputs ought ideally to expose 'new' aspects of software behaviour that have not featured in previous test executions. Intuitively, the test generation approach tends to be closely tied to the type of inferred model (e.g., if state machines are inferred, then likely state machine testing algorithms are used to derive new tests [22], [33]).

Unfortunately, there are two barriers that currently restrict LBT approaches to relatively specific classes of relatively small-scale software systems:

1) The dependence between the type of inferred model and the test generation approach can be highly limiting. Whole families of Machine Learning algorithms have to be excluded as they do not produce 'testable' models.
2) The application of model-based test generation approaches to inferred models can yield large numbers of test cases, which hampers scalability. Many of the generated tests are of little *utility* to the learner. Whereas the goal is to find 'counter-examples' to the inferred models, the majority of tests merely ends up corroborating what is already known.

In this paper we investigate the possibility of using an Active Learning query strategy framework [29], [30] to circumvent these limitations. In Machine Learning, query strategy frameworks provide a means by which to use an existing inferred model (or set of models) to select further samples that are most likely to be of "high utility" to the learner – i.e. provide information that is not already contained within the training set. These tend to be based on the principle that the best samples are those whose prediction elicits the highest degree of *uncertainty* with respect to the current model. In the context of LBT, if one accepts the existence of a relationship between the adequacy of a test set and the accuracy of a model inferred from it, then it should follow that test cases selected by an effective uncertainty sampling technique should form an effective basis for test case selection.

In detail, the contributions of this paper are as follows:

- We introduce the first application of query strategy frameworks to test generation (Section IV).
- We present an implementation of a query strategy framework for test generation using Query By Committee [30] on inferred models (Section IV).

- We propose the use of Genetic Programming [21] as a basis for model inference, as it directly enables Query By Committee (Section IV).
- We present an implementation of an LBT-based testing using query strategy frameworks, based on Genetic Programming and Query By Committee (Section IV).
- We present an empirical evaluation on eight functions provided within the Apache Commons Math and Joda-Time frameworks, using mutation testing to assess the effectiveness of the generated test cases (Section VI).

Our experiments demonstrate that uncertainty sampling leads to a higher mean number of mutants detected than random or adaptive random testing (the baseline techniques we use in this paper). It also tends to require fewer test executions to detect higher numbers of mutants. This is especially valuable for test-scenarios where there is a non-trivial cost associated with test execution (e.g. tests take a prohibitive amount of time, or their outputs need to be checked by a human test-oracle).

II. Automated Black-box Testing

Black-box testing in general refers to the concept of testing a software system without access to its source code. Ideally, black-box testing is driven by detailed formal specifications or test models, which enable techniques to automatically generate tests, and act as a test oracle that decides whether a given test execution revealed a fault or not. In practice, such specifications are not always available, in which case automated generation of tests is limited to few options.

A. Random Testing

The most common approach to test automation in the absence of formal specifications and source code is to randomly select tests, for example using a uniform distribution on the input space or an operational profile [16]. The effectiveness of random testing highly depends on the specifics of the system under test: Random testing is generally unlikely to find specific input values [3], and may perform poorly at covering the underlying behaviour of the program.

Adaptive Random Testing (ART) [8] aims to alleviate these problems by ensuring that tests are spread across the input space as much as possible. In general, ART works iteratively by repeatedly sampling a set of random inputs, and out of this set selecting the input that is most different to previously executed tests as the next test to run on the system under test. While there is evidence that this approach makes the selected tests more effective than a completely random selection, every test input adds to the complexity of generating the next test input, because there is an additional point in euclidean space against which to measure the next group of random inputs.

If running a test on a system under test is cheap, then pure random testing may be more effective than ART [2] as it can simply execute significantly more tests in the same time as ART. However, in practice test execution can often take a long time, and the absence of an automated oracle (e.g., a formal specification) may make it necessary to manually investigate every single test outcome. Thus, we assume that it is desirable to generate the most effective set of tests, rather than relying on the ability to run large sets of potentially redundant tests.

Algorithm 1: Generic LBT procedure

```
Input: $SUT$, $TestInputs$
Uses: $terminate$, $execute$, $selectInputs$, $inferModel$
Result: $TestInputs$
$hyp \leftarrow \varnothing$ ;
$Executions \leftarrow \varnothing$ ;
for ($input \leftarrow TestInputs$) do
    $Executions \leftarrow Executions \cup execute(input)$;
while ($\neg$ $terminate(Executions,hyp,SUT)$) do
    $hyp \leftarrow inferModel(Executions)$;
    $NewInputs \leftarrow selectInputs(hyp,SUT)$;
    $Executions \leftarrow Executions \cup execute(SUT, NewInputs)$;
    $TestInputs \leftarrow TestInputs \cup NewInputs$;
return $TestInputs$;
```

B. Learning-Based Testing

We use the term 'Learning-Based Testing' (LBT) to refer to the (now relatively broad) family of techniques that seek to use Machine Learning to support the generation of test cases. The idea was first explored by Weyuker [34] and Budd and Angluin [7] in the early eighties. For the subsequent 15 years it was the subject of some predominantly theoretical research [9], [28], [35]. However, over the subsequent 15 years it adopted a more practical bent, with several authors developing accompanying proof-of-concept tools [4], [13], [14], [22], [24], [27], [33].

Algorithm 1 shows the main generic LBT steps:

- The algorithm starts with an initial set *TestInputs* of inputs to the program. This may be empty, but it may also be an established test set that we wish to improve.
- The loop of model inference and test generation is executed until a stopping criterion *terminate(Executions,hyp)* evaluates to true. For example, it might attempt to establish the equivalence between the inferred model *hyp* and the system under test *SUT*, and return true if the model is sufficiently similar in some sense [27]. It might alternatively simply terminate after a fixed number of iterations, if *Tests* reaches a particular size, or there has been no change to *hyp* after a certain number of iterations.
- In this loop, the first step is to infer a predictive input/output model *hyp* of the program using the function *inferModel(Executions)*. The type of the model can vary, and depends on the nature of the system under test. Proposed techniques have adopted state machines [27], [33], decision trees [6], [14] and Daikon invariants [15].
- The input to the *inferModel* function are the executions, i.e., the input/output pairs resulting from executing the test inputs *TestInputs* on the system under test *SUT* using function *execute(SUT,Inputs)*.
- Finally, *selectInputs(hyp,SUT)* selects new inputs. The test generation strategy might be random [24], driven by source code coverage [14], or using a model-based test algorithm with respect to *Mod* [27].

Much of the research on combining inference and testing

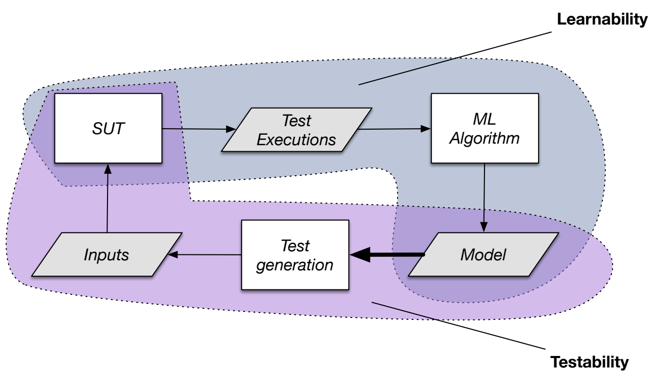

Fig. 1. The relationship between 'learnability' and 'testability' in LBT.

has focussed on the interplay between the *terminate* and *inferModel* functions — on the ability to leverage inference mechanisms to provide more meaningful adequacy criteria. This is what motivated most of the early research into the area as well [9], [34], [35]. Recent inference and test generation techniques have been combined to guarantee that the behaviour of the system has been exercised to a certain extent. For example, several researchers have combined Angluin's $L*$ inference technique [1] with established state machine testing techniques [13], [27] and showed that these lead to strong guarantees that the inferred model accurately represents what has been explored.

C. Limitations of Learning-Based Testing

LBT techniques tend to be limited in their practical applicability because they rely on the inference of models that not only approximate the behaviour of the SUT, but are *also* usable as a basis for automated test generation. As illustrated in Figure 1, the processes of inference and testing are highly interdependent; the model has to be *learnable* from the SUT [31], but also has to be *testable*, in the sense that it can provide a suitable basis for the generation of new test cases. This explains why LBT techniques so far have been largely restricted to state machines [13], [27], [33], decision trees [6], [24], and invariants [15]. As a consequence, entire families of Machine Learning algorithms that infer models that are harder to subject to symbolic reasoning are excluded, even if they could potentially infer more accurate models from a broader class of SUTs. These include kernel-method techniques such as SVM [32], Neural Net learners [5], and Genetic Programming [21].

Aside from the constraints mentioned above, the use of 'off-the-shelf' test generation techniques, coupled with the iterative nature of the approach, can lead to scalability problems. Test generation algorithms generate test cases without considering the test cases that have been generated in previous iterations. This can produce very large test sets, especially if the testing algorithm in question produces large numbers of test cases anyway (e.g., the popular W-Method [10] for state machines is a good example of an algorithm that does not scale well, and has often been used for test-driven model inference [27]).

The primary challenge addressed in this paper is to find a means by which to remove the constraints on the classes of Machine Learners that can be applied to LBT. Although we want to remove the constraints on the types of models that are inferred, it is crucial that they can still guide test generation, and do so in a scalable way that does not needlessly re-test behaviour that has already been tested.

III. Query Strategy Frameworks

In this paper we will show how the above problem can be solved by the use of query strategy frameworks, a core facet of active learning techniques [29] in Machine Learning. Precisely how this is achieved will be described in the next section. Here we provide a generic introduction to query strategy frameworks and Query by Committee.

A. Query Strategy Frameworks

Machine Learning algorithms can be broadly categorised into conventional (*passive*) Machine Learners and *active* Machine Learners. Passive learners infer models from data that is given to them before learning commences. Active learners might start from some given data, but crucially are also imbued with the ability to obtain further data. The learner might surmise that the inferred model is incomplete because the initial sample lacked data of a certain characteristic, so the learner can obtain more relevant data, which it can use to refine its model.

The active learning setting gives rise to the *query strategy problem* [29]. The process of obtaining a sample might be expensive, so it is consequently important to keep the number of queries (additional samples) down to a minimum. However, any additional data that *is* sampled must be of a *high utility* — i.e., it must lead to improvements in the model inferred by the learner. This problem has been the subject of a large amount of research over the past two decades (a good overview is provided by Settles [29]). The essential goal is to avoid selecting a query that fails to add new information that is of value to the learner. Any new data should ideally confound the predictions of the current model.

One factor that plays a key role in selecting queries is the notion of *uncertainty*. Given a data-point that was not part of the original training set (referred to as a 'query'), the degree of uncertainty exhibited by the current hypothesis model as to how it should be classified can provide an indication of how useful it would be to obtain a real sample. The goal is thus to identify queries for which the level of confidence in the corresponding output is at its lowest, with the aim of eliciting aspects of behaviour that were perhaps under-represented in the training sample.

One key challenge is to find a suitable metric that can be used to assess this "uncertainty" for a given model prediction. For statistical Machine Learners, where the output is often in the form of a probability distribution, numerous uncertainty sampling techniques have been developed [29]. However, in the context of LBT, models such as inferred state machines tend not to be probabilistic.

Algorithm 2: Query By Committee

```
Input: $Train$, $i$, $s$,$comitteeSize$,$randomPoolSize$
Uses: $learnMultiple$,$best$, $computeUtility$, $randomPoints$
Result: $Hyp$
$Hyp \leftarrow \varnothing$ ;
for $i$ iterations do
    $Hyp \leftarrow learnMultiple(Train, comitteeSize)$;
    $U \leftarrow randomPoints(randomPoolSize)$;
    for $s$ iterations do
        // Pick a point $u \in U$ with max utility
        $u = argmax_{x \in U} \mid\mid computeUtility(Hyp, x) \mid\mid$;
        $l = label(u)$;
        $Train \leftarrow Train \cup \{l\}$;
        $U \leftarrow U \setminus \{u\}$;
$Hyp \leftarrow learnMultiple(Train)$;
return $best(Hyp)$;
```

B. Query By Committee

There is a 'trick' that enables the application of uncertainty sampling even when the inferred models are themselves not probabilistic. If one can, from a given sample, infer *multiple* different models, then it becomes possible to use their mutual agreement / disagreement to estimate a level of uncertainty and use this as a basis for uncertainty sampling [30]. Algorithm 2 shows the Query By Committee (QBC) approach [30].

- The entire process iterates a fixed number of times (i).
- At each iteration, the *learnMultiple* function produces a "committee" of hypothesis models. This is conventionally achieved by Ensemble Methods [23], which produce different hypotheses by inferring models from different samples of the training set (in this paper we will illustrate an alternative approach of using the population generated by a Genetic Programming algorithm).
- Once the models have been inferred, the *randomPoints* function generates a set of random 'inputs' U – in Machine Learning terms this is a set of *unlabelled* data points. The size of U is determined by the *randomPoolSize* parameter.
- The nested for-loop then essentially picks a subset of s points in U. These are selected by evaluating each point in U to determine those points about which the inferred models *Hyp* are *least* in agreement (as computed by the *computeUtility* function). In other words, these points would be of most *utility* to the learner.
- The selected points are labelled with the *label* function, added to the training set, and the process iterates.
- After the final iteration, a set of models is inferred from the aggregate training set, and a model is selected to be returned by the *best* function. The selection criteria can vary depending on the inference approach – one straightforward option (adopted in this paper) is to return the model that best predicts the outputs (or 'labels') produced by *Train*.

There is a clear similarity between the QBC algorithm and the LBT algorithm in Algorithm 1. Both involve loops, where models are inferred at each iteration. In both cases, the models are used as a basis for selecting more data (test inputs in the testing context, unlabelled data points in the Machine Learning case). There are also two significant differences. In the case of QBC, the output is the final inferred model, whereas in LBT the output is the data that was used to infer the model (the test set with its outputs). In LBT, there is no fixed approach to generate test data – it could be random, or adopt a model-based testing algorithm. In QBC, there is only one approach; regardless of the type of model or system, a random pool of unlabelled data points are generated, and the best s points are chosen based on the 'uncertainty' that they elicit from the inferred committee of models.

Algorithm 3: Testing By Committee

```
Input: $SUT$,$Tests$,$s$,$i$,$comitteeSize$,$randomPoolSize$
Uses: $execute$,$learnMultiple$, $randomInputs$,$computeUtility$
Result: $Tests$
$Hyp \leftarrow \varnothing$ ;
for $i$ iterations do
    $Hyp \leftarrow learnMultiple(Tests, comitteeSize)$;
    $U \leftarrow randomInputs(SUT, randomPoolSize)$;
    for $s$ iterations do
        $u = argmax_{x \in U} \mid\mid computeUtility(Hyp, x) \mid\mid$;
        $l = execute(u)$;
        $Tests \leftarrow Tests \cup \{l\}$;
        $U \leftarrow U \setminus \{u\}$;
return $Tests$;
```

IV. Applying QBC to Test Generation

In the context of Machine Learning, QBC enables uncertainty-based sampling to occur, regardless of the type of model that is inferred. In this paper we produce the Testing By Committee approach, which applies QBC to LBT to circumvent the dependence between the model inference algorithm and the test-generation algorithm. In principle this enables LBT to use *any* model inference algorithm, and to select test cases based on the combined uncertainty of the inferred models.

In this section we first set out our *Test By Committee* algorithm, which combines LBT with QBC. We then provide a technique that implements this approach using Genetic Programming as a basis for the model inference.

A. Test By Committee

Our proposed 'Test By Committee' (TBC) algorithm is shown in Algorithm 3. It clearly combines Algorithms 1 and 2. The key similarities and dissimilarities are as follows:

- As with QBC, we limit the number of iterations to a fixed number i (though it would certainly be possible to integrate something more elaborate, along the lines of the *terminate* function in Algorithm 1).
- The step of *learnMultiple* is the same as in Algorithm 2; a population of models are inferred using either ensemble methods or, as we will demonstrate, population-based learners such as Genetic Programming.
- To generate the candidate test inputs, we introduce a

new function *randomInputs*. The *SUT* is only used to gain information about its interface. Once the types of the interface are known, inputs are formulated as combinations of random values of the appropriate types.

- The process of adding new tests to the test set is the same as in Algorithm 2. For s iterations, the best candidate is selected from U by seeing which candidate test case causes the most disagreement amongst models in *Hyp*. The chosen test is then executed to identify its actual output, and this is then added to *Tests* (it is also removed from U to avoid re-selection).

Many of the steps are in effect the same as they are in conventional LBT. However, two steps are very different, and therefore require a more in-depth discussion. The model inference step (*learnMultiple*) requires multiple models. The process of selecting the best candidate test case (*computeUtility*) is also new in the context of testing. In both cases, there are many possible ways in which they could be implemented. In the following two subsections, we describe how we have chosen to implement them for our proof of concept.

B. Learning Multiple Models by Genetic Programming

To produce the models required for Query-by-Committee it is possible to use a Genetic Programming (GP) inference engine [25]. A GP evolves programs of a given target language towards an optimisation goal, specified by a fitness function. As mentioned previously, in principle any inference technique could be applied (underpinned by Ensemble Methods [12]). However, (a) the intrinsic population-based nature of GPs renders them suitable for QBC, and (b) GPs can easily be adapted to different types of languages, making them well suited for modelling programs in different domains.

For space reasons, we only provide the essential details of GPs here, and refer the reader to Poli *et al.*'s GP field guide [25], along with our source code[1] for further details. In (tree-based) GPs, candidate programs are 'evolved' as abstract syntax trees, where branch nodes correspond to 'non-terminals' representing functions, and leaf-nodes represent atomic values or variables (terminals). The basic loop is as follows (details on the terms in italics will be elaborated in the next section):

1) Generate an initial population of programs as random compositions of non-terminals and terminals.
2) Execute each evolved program and evaluate it according to some *fitness function*.
3) *Select* the best programs from the population.
4) Create a new population using *crossover* and *mutation*.
5) Repeat from step 2 until some stopping criterion is met.

In its traditional application, the result of the GP is the program with the best fitness value, which represents the best solution. In our case, we can exploit the population-based nature of the GP: At the end of the search, the population consists of a range of slightly varied candidate solutions optimised for the problem at hand.

[1]https://bitbucket.org/nwalkinshaw/efsminferencetool

C. Generating Test Cases by QBC

The first step to applying QBC is to select the committee. For this we select the fittest set of chromosomes *Hyp*. The size of this set is determined by the parameter *committeeSize*. The query generation step involves generating a pool of random inputs U, and then assessing every $u \in U$ to find the one that creates most 'uncertainty' according to the set of inferred models in *Hyp* (in our case the set of chromosomes inferred by the GP). Every potential test input u is executed on every model $h \in Hyp$, and the outputs are recorded. The input that produces the greatest spread of predictions is then chosen to be executed on the real SUT.

V. Implementation

As a proof-of-concept, we have implemented the approach described in the previous section for side-effect free numeric programs returning single outputs. This section provides details of this implementation.

A. A GP for Programs with Primitive Types

In this section we elaborate the detailed aspects of the generic GP algorithm shown in Section IV-B.

Fitness function: The fitness function provides a metric for the accuracy of the inferred program to predict the SUT. Fitness is evaluated by executing a candidate program on all existing test inputs, and comparing the outputs to those that were actually observed in the trace data.

Selection: Step 3 selects good candidates from the population to be fed into the next generation. A popular approach, which we adopt here, is Tournament Selection [25]. In our case the selection process is *elitist*; i.e., the best individual from one generation is always preserved for the next one.

Crossover and Mutation: The candidates that were selected in step 3 are subjected to a mixture of crossover and mutation (the frequency at which they occur is given in probabilistic terms). We choose to use the most common form crossover called *subtree-crossover* [25]: A pair of candidates is chosen, in each candidate tree a random node is selected, then the sub-tree rooted at the node selected in the first parent is substituted by the subtree rooted at the node selected in the second parent. Mutation is carried out by selecting a random node in a tree and changing it. If the selected node happens to be a terminal, its value is simply changed. If it is a non-terminal, we replace its subtree with a randomly generated one. Arbitrary crossover or mutation can easily lead to nonsensical programs, e.g., by using String terminals with a function that expects integer parameters. Strongly-typed GP [25] prevents this from happening by ensuring that every terminal and non-terminal has a declared type.

Termination and result: The loop terminates once a candidate has been identified that cannot improve in terms of fitness, or once the number of iterations hits a given limit.

Terminals and Non-Terminals: The choices of terminals and non-terminals are shown in Table I. In general, of course, the choice of GP operators is flexible, and is ideally informed by knowledge about the system being inferred. In our case, we

TABLE I
NON-TERMINALS AND TERMINALS CHOSEN FOR OUR EXPERIMENTS

Non-Terminals	
Double (D)	add(x:D,y:D), subtract(x:D,y:D), multiply(x:D,y:D), divide(x:D,y:D), power(x:D,y:D), root(x:D, y:D), cast(x:I), if(x:B,y:D,z:D), cos(x:D), exp(x:D),log(x:D)
Integer (I)	cast(x:D)
Boolean (B)	and(x:B,y:B), or(x:B, y:B), LT(x:D,y:D), GT(x:D,y:D), EQ(x:B,y:B), EQArith(x:D,y:D),EQString(x:S,y:S)
Logic (all)	if-then-else(a:B,b:D,c:D),if-then-else(a:B,b:I,c:I),if-then-else(a:B,b:S,c:S),if-then-else(a:B,b:B,c:B)
Terminals	
Double (D)	all variable names in *Vars* of type double, one free variable limited to the interval $[-2, 2]$, -1.0
Integer (I)	all variable names in *Vars* of type integer, one free variable limited to the interval $[-2, 2]$, 0
Boolean (B)	All variable names in *Vars* of type Boolean, `true`, `false`.
String (S)	All variable names in *Vars* of type String, any customised pre-defined String values.

sought a reasonably general set that can be applied across a range of programs. The question of how to refine the selection of terminals and non-terminals to best suit an SUT is part of our ongoing work.

B. Generating Test Cases by QBC

For our proof of concept, we are restricting ourselves to a particular class of system that produces single numerical outputs (either integers or numbers with decimal places). Our initial use of standard deviation to quantify uncertainty proved to be problematic, as it could often produce a misleadingly high value for the situation where most of the models were in fact in agreement, but one “rogue” model had produced an extreme value. To address this problem, we instead opted for the Mean Absolute Deviation (MAD) value [20], which is less vulnerable to data-spikes. For a set of values $X = \{x_1, \ldots, x_n\}$, $MAD(X) = \frac{1}{n}\sum_{i=1}^{n} \mid x_i - m(X) \mid$, where $m(X)$ calculates the mean of X.

It is necessary to select a value to accommodate the situation where an inferred model returns either infinity or Not a Number (e.g., because an inferred model divides by zero), but the SUT returns a valid value. The value should be high, to indicate that the model is incorrect, but cannot be too high (e.g., Double.MAX_VALUE), because this prevents the calculation of an accurate mean over multiple outputs. In this case, we substitute the result with a value of 10,000,000 (this was a somewhat ad-hoc choice, and establishing a more justified value is part of our future work).

C. Example – The BMI Calculator

This section contains a brief walk-through of TBC. As an SUT we choose a simple BMI calculator. This takes as input two numbers (height in meters and weight in kilograms), and returns a “Body Mass Index” value, calculated as $\frac{weight}{height^2}$. For our technique to operate, we do not need to be able to look at the internal implementation, but only need to know the interface. However, to provide a complete overview, let us assume that the calculator is implemented as a bash script, with the following source code:

```bash
#!/bin/bash
awk "BEGIN {print $2 / ($1 * $1)}"
```

Our implementation accepts a specification of the interface in the following self-explanatory JSON format.

```
{  "command": "bmi.sh",
   "parameters":[
      {  "name": "height",
         "type": "double",
         "max": "100",
         "min": "-100" },
      {  "name": "weight",
         "type": "double",
         "max": "100",
         "min": "-100"}
      ],
   "output":[
      {  "name": "output",
         "type": "double" }
      ] }
```

Finally, we provide an existing basic test set that we wish to improve upon. Our implementation accepts a space-separated text file, where the order of values is taken to be the order of parameters in the specification file (height followed by weight):

```
1.7 50
1.8 70
1.9 100
1.7 110
0.0 5
5.0 0
```

With reference to the TBC process in Algorithm 3, the BMI represents the *SUT*, and the above list of test sets represents *TestInputs*. For the sake of illustration, we will only show one iteration ($i = 1$), and we will only add a single test set in this iteration $s = 1$. To illustrate how new test cases are selected, we set *randomPoolSize* to 3, although this would usually be much higher (in the evaluation we will set it to 1,000).

The TBC algorithm begins by inferring the “committee” *Hyp* via *learnMultiple*. In our case, this produces the top 10 chromosomes. To give an idea of what is inferred, two of the fittest GP programs after the first iteration is as follows:

```
gp1: Mult(weight,Exp(-1.1518922634307343)))
gp2: Div(height,Exp(height-Log(weight))
```

Although they are clearly inaccurate, we can assume that (as the fittest members of their pool of solutions), they at least *approximate* the output. This is illustrated in Figure 2, which plots outputs (the dashed and dotted lines) against the expected output (the plain line), for all test inputs.

As the next step, *randomInputs* produces a set of *randomPoolSize* inputs (in this case three, see left-hand side of Table II). For each input, the disagreement between the models is calculated as the MAD, shown in the right-hand column. From this, it is clear that the second input produced a huge divergence between the two inferred models.

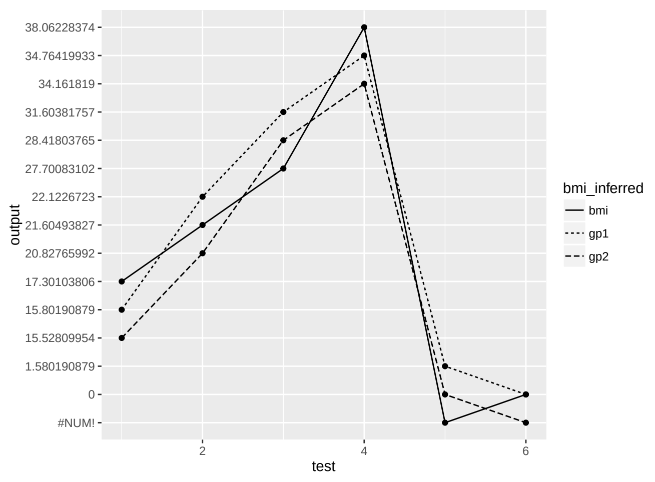

Fig. 2. Comparison of expected versus inferred outputs after one iteration wrt. BMI example.

TABLE II
PROPOSED INPUTS AND MAD CALCULATION

height	weight	MAD
87.95	50.49	3.99
-62.41	91.14	**1.80E+30**
26.44	56.65	4.48

The input with the highest MAD value is thus added to the test set, and the TBC process moves to the next iteration. This time, thanks to the new test execution, the inferred models ought to be more precise, and lead to test cases that explore new aspects of the input domain.

VI. EVALUATION

In this section we seek to assess the effectiveness of TBC at generating rigorous test sets. Of primary concern is the question of whether TBC can detect more faults than baseline testing techniques. In this evaluation we use random testing and Adaptive Random Testing [8] as the baseline. Accordingly, the first research question is as follows:

RQ1: Do TBC-generated test sets expose more faults than random and ART-generated test sets?

One further question is concerned with the *efficiency*. In the event that TBC does not ultimately expose a larger number of errors than other techniques, it might still expose the same number of faults, but after executing fewer tests, which would render it more efficient:

RQ2: Are TBC-generated test sets more efficient at exposing faults than random and ART-generated test sets?

A. Subjects

We chose six units within the Apache Commons Math framework (version 3.6)[2] and two units within JodaTime (version 2.9.3)[3] using the following criteria:

- It must accept a single (set of) input parameters – i.e., it must not require sequences of method calls (apart from the call to the constructor).
- It must produce a single output value.
- The parameters accepted by the unit under test (and its output value) must either be primitive data types that are supported by our GP implementation, or be complex objects where the constructor accepts primitive data types.
- The unit in question must be invoked by one of the Apache Commons Math or JodaTime test sets (so that we can use these tests to infer the first model).

[2]https://commons.apache.org/proper/commons-math/
[3]http://www.joda.org/joda-time/

TABLE III
SUBJECT SYSTEMS.

Component	Functionality	Exec. LOC	Tests
`BesselJ`	`value`	1,211	699
`Binomial`	`binomialCoefficientDouble`	501	3,000*
`DerivativeStructure`	`asinh`	360	3,000*
`Gamma`	`regularizedGammaQ`	783	4
`Erf`	`erf`	763	116
`RombergIntegrator`	`RombergIntegrator`	735	4
`Period`	`toStandardWeeks`	1,128	5
`Days`	`daysBetween`	1,251	8

TABLE IV
MEAN NUMBER OF MUTANTS KILLED AFTER 60 ITERATIONS. HIGHEST VALUES ARE IN BOLD. THE SIGNIFICANCE OF THE MANN-WHITNEY TEST IS INDICATED IN PARENTHESES. NO SIGNIFICANCE - $p > 0.05$ IS (-) , $p < 0.05$ IS (*), AND $p < 0.001$ IS (***).

SUT	TBC	Random		ART	
BesselJ	**447.50**	442.83	(***)	442.93	(*)
Binomial	**30.53**	29.03	(***)	29.20	(***)
DerivativeSin	**55.93**	51.20	(***)	50.07	(***)
Erf	**190.52**	188.62	(***)	189.33	(***)
Gamma	**208.23**	206.90	(-)	205.60	(-)
Romberg Integrator	**87.77**	87.63	(-)	87.46	(-)
periodToWeeks	**304.95**	249.52	(*)	271.58	(-)
daysBetween	**72.13**	50	(-)	49.53	(*)

The eight units in Table III represent the first units that were encountered in each system. Where a package contained a large number of possible varieties (e.g., calculations of derivatives), we chose one at random, and avoided choosing multiple units in the same collection. Where an initial test set was particularly large (some contained $>$ 20,000 executions of the SUT), we sampled 3,000 executions at random to ensure that the fitness functions in the GP could be evaluated in a reasonable amount of time. These are marked with a '*' in Table III.

Apache Commons Math and JodaTime were chosen because they are written in Java, which enables us to use the Major mutation framework [19], and because they have a reasonably extensive set of unit tests (enabling us to use these as a starting point for the learning-based testing). Their details are shown in Table III. The sizes of the various functionalities have to be treated as approximate. As the LOC of the entire libraries would be an overestimation and the LOC for a single class would be a gross underestimation (especially in the case of Apache Commons Math, where a large portion of the functionality is contained within the very large `FastMath` class), we provide the total LOC within the library tracked (using IntelliJ) when executing all generated test sets for a given SUT.

It is important to note that these selection criteria are in part so restrictive for the sake of control in our experiment. In practice, if we wanted to test a system for which our current GP was not sufficient, we would resort to a different Machine Learner, or add the requisite terminals and non-terminals to the GP. However, in our case, this special treatment would obviously bias the results. To avoid bias, we thus restrict ourselves to a subset of systems that are at least compatible with our choice of GP.

B. Methodology

To gauge the performance of TBC in comparison with the 'state of the art', we compared the mutation scores for its test sets against randomly generated test inputs, and test sets generated by Adaptive Random Testing (ART) [8]. For ART, an important factor is the choice of distance function to distinguish test sets. In our case, since most of the inputs were numerical, we chose the Euclidean distance function, which tends to be the distance measure of choice.

All of the techniques were provided with an interface specification file, which contained the various parameters, and the ranges for any numerical parameters. If parameters were strings, the potential value-selections were explicitly enumerated. To avoid biasing results, we did not use any domain to set numerical variable range boundaries, and adopted a conservative approach; we looked at the ranges in the given test sets, and expanded these ranges with a substantial buffer in either direction (e.g., if the range of the test cases was from 0 to 10, we would set the range from -100 to 100). The full configuration files, along with all other materials used for this experiment are available online[4].

To gauge how effective a test set is at exposing faults, we employed mutation testing [17]. We used the Major Java mutation testing framework (version 1.6, with all mutants) [19]. We seeded mutants conservatively, by selecting any classes that were executed by the initial set of tests (we could not seed mutants in every class in the system because of the resource constraints of mutation testing). It does not make sense to measure the mutation score as the proportion of mutants killed, because the conservative seeding strategy will invariably mean that this proportion is liable to be very small (for example, all of the units use a fraction of the `org.apache.commons.math4.util.FastMath` class). Instead, we simply compare the absolute numbers of mutants detected, which suffices to provide valid answers to our two research questions.

To prevent any bias arising from configurations, we used the same configuration for TBC across all experiments. For the GP configuration we used the set of terminals and non-terminals detailed in Table I. We used a population size of 800, with a crossover-rate of 0.9, a mutation rate of 0.1, a maximum term-depth of 10 and a tournament size of 6 [25]. We set the number of tests generated per iteration to 1,000 (this same number was used for the randomly generated tests and ART), and the number selected for addition to the test set to 5.

To answer RQ1, we analysed the mutation scores that were computed after 60 iterations, grouped according to the technique (TBC, ART, and Random). We chose 60 iterations as a cut-off with a view to gathering sufficient data to highlight trends in performance, whilst also ensuring that the experiments did not require too much time. To compare the results we carried out two (non-parametric) Wilcoxon Rank Sum tests per SUT (having confirmed that the distributions are not normally distributed according to the Shapiro Wilks test). The first null-hypothesis was that the mutation scores for TBC are smaller than those for random tests. The second null-hypothesis was that the mutation scores for TBC are smaller than those for ART tests. The distributions were also visualised as box-plots.

To answer RQ2 (how much more effective is TBC?), we recorded the last iteration at which TBC produced the highest mutation score (versus ART and Random). We also plotted the trajectories of the means to show how the trajectories differed over the course of the 60 iterations.

C. Results for RQ1: Effectiveness

The mean numbers of mutants killed for each system are shown in Table IV. The distributions are also visualised as box-plots in Figure 3. The table shows that, after 60 iterations, TBC has killed the highest mean number of mutants for every program. The improvement over ART and random testing varies substantially between the systems. For BesselJ, Binomial, Derivative Sinh, and ERF, the difference is statistically significant; this is corroborated in the box plots. In three of these systems (Binomial, Derivative Sinh and ERF), difference is so marked that the lower quartile for TBC is higher than the upper quartile for ART and Random.

For Gamma, Romberg, PeriodToWeeks and DaysBetween although the mean is higher for TBC, the differences are not statistically significant (they are partially significant for PeriodToWeeks and DaysBetween). Looking at the box plots, in all cases apart from PeriodToWeeks the boxes for TBC are noticeably elevated. In the case of PeriodToWeeks, the median score for TBC is the same as ART (even thought the mean score is substantially higher). This is largely due to one particular execution that achieved a particularly large number of mutations. In all cases, the difference in distributions is particularly marked at the lower end; ART and Random have lower minimum scores, and lower lower-quartiles than TBC, which indicates that TBC is more consistent.

RQ1: In our experiments, TBC was more effective than random testing and ART. In all cases there was a higher mean number of mutants killed, and the difference in distributions was significant in 4/8 SUTs.

D. Results for RQ2: Efficiency

We discuss the relative efficiency of TBC versus ART and random testing by looking at how rapidly TBC out-performs the other approaches (by achieving a higher mean number of mutation faults without being overtaken in subsequent iterations). Figure 4 shows the average mutation scores and their standard deviations throughout the 60 iterations. It is important to note the differences in scales; the different SUTs give rise to markedly different numbers of mutants. This means

[4]http://staffwww.dcs.shef.ac.uk/people/G.Fraser/ICST2017/

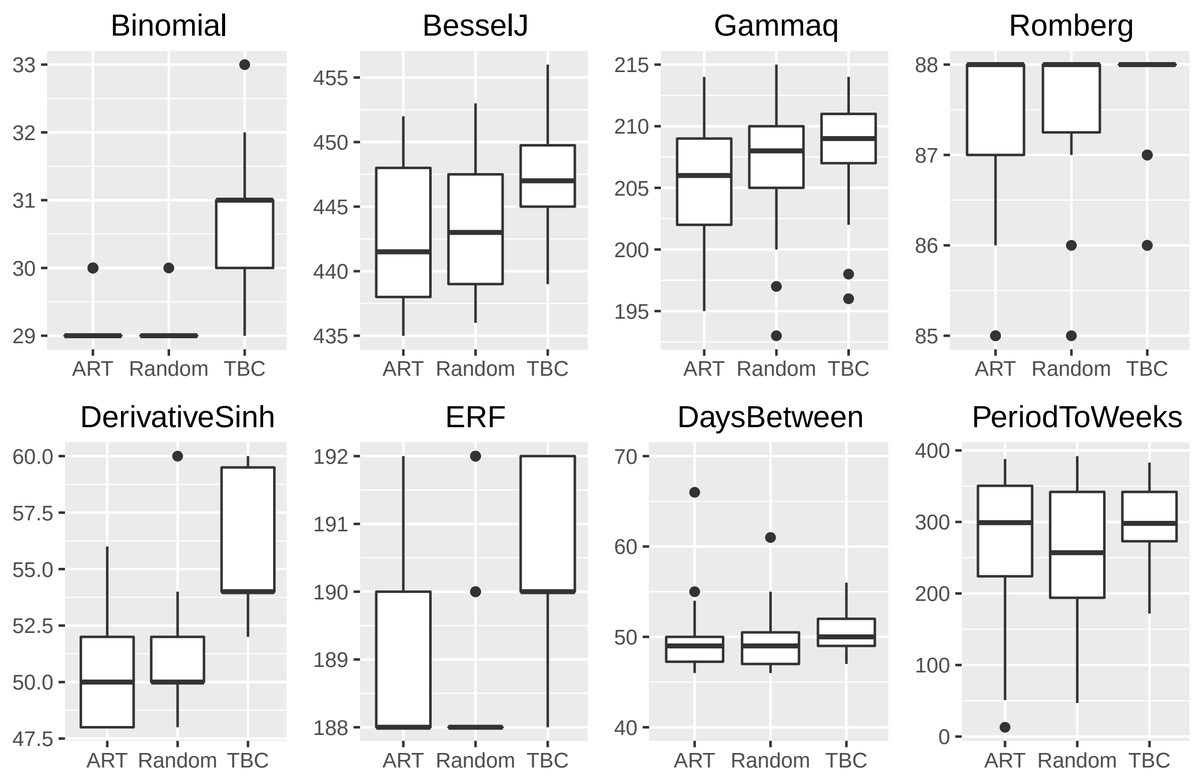

Fig. 3. Mutation Scores after 60 iterations, starting from given test sets.

that similar differentials in the mean numbers of mutants on different plots can appear markedly different. We discuss the various trajectories by starting with the systems where the performances are most similar.

In all of the studied systems, TBC eventually kills more mutants on average than random and ART testing. In some systems the numbers of faults detected remain similar throughout, whereas in others TBC significantly outperforms ART and Random from the start. These cases are discussed in more detail below.

As one might expect from the results for RQ1, the trajectories in the Romberg and Gammaq SUTs are visually similar; these are the systems where the relative performance between the techniques is at its closest. In the Romberg SUT, TBC is consistently better than ART from iteration 20 onwards, but only outperforms random testing after iteration 50. In Gammaq, TBC consistently outperforms ART and random from iteration 23 onwards, though only marginally.

Perhaps more surprisingly for both JodaTime systems PeriodToWeeks and DaysBetween the trajectory for TBC is noticeably higher than for ART and Random. For PeriodToWeeks, the number of mutants killed for TBC rapidly increases after 10 iterations to a level that ART and Random only start to approach after 40-50 iterations.

In ERF, both ART and TBC outperform random testing from the start. ART and TBC are similar up to iteration 40, where ART continues to plateau at 189 whilst the mean number of killed mutants for TBC rises to over 190.

In Binomial, BesselJ, and DerivativeSinh, the results for TBC are markedly better from the start. In the case of BesselJ the difference may look smaller, but this is because of the scale of the graphs. In BesselJ the mean TBC score after 60 iterations is 447.5, whereas for scores for ART and Random are approximately 443; this difference of 5 is in fact larger than the differences in the other systems.

RQ2: In our experiments, TBC was significantly more efficient at exposing faults than random testing and ART.

E. Threats to Validity

Threats to external validity: The answers to RQ1 and RQ2 can only validly be applied to systems of a similar character to those tested here. We have only tested eight systems from two frameworks. This means that they will often have shared developers, and they all deal with similar domains. We have additionally restricted ourselves to units that are functional, which do not accept sequential inputs (as discussed in Section VI-A). To attenuate this risk, we attempted to make the selection of SUTs as indiscriminate as possible within our broader selection constraints. The SUTs presented here are the first ones we encountered that fitted our criteria. However, a larger study on a more diverse range of SUTs is needed, which is what we will be doing in our future work.

As mentioned previously, the choice of value ranges for the parameters is important for all of the techniques. Our choice of ranges may not be ideal, given that we avoided using domain knowledge to avoid bias. It is possible that, for certain range limitations, the differences between the various techniques are reduced (i.e., if the value ranges are reduced). Investigating

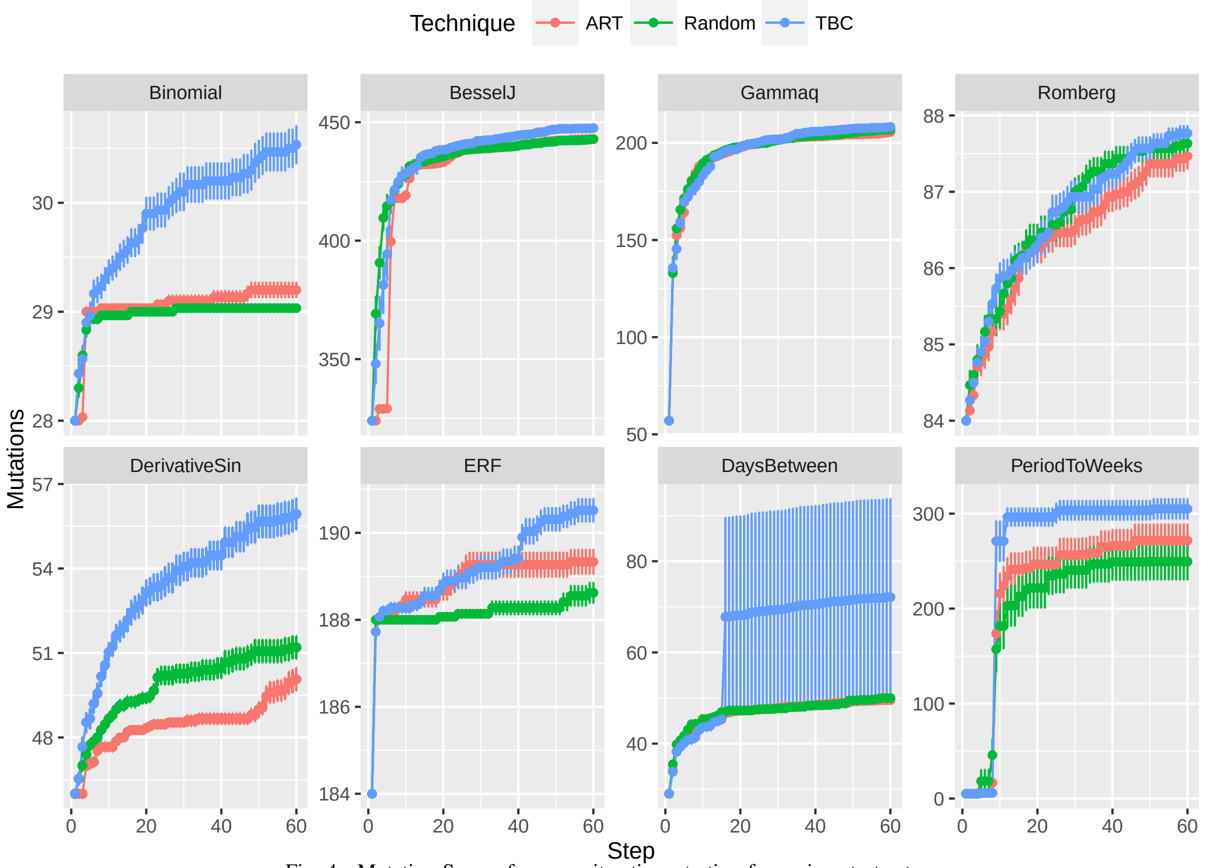

Fig. 4. Mutation Scores for every iteration, starting from given test sets.

the relationship between the selection of value ranges and the relative performance of these techniques is something that we are exploring as part of our ongoing and future work.

The effectiveness of our generated test sets is currently assessed by their capacity to detect seeded defects (mutants). It is assumed that this translates to an ability to expose real faults. This will need to be verified in future work, which will apply our experiments to fault repositories, such as Defects4J [18].

Threats to internal validity: The mutation score depends upon the seeding of mutants. It is possible that code was executed that was not seeded with mutants, thus skewing the results. We attempted to limit this possibility by tracking the execution of code with profiling tools.

F. Discussion

The results indicate that TBC tends to detect more faults *per test* than the baseline techniques. One factor, however, that was not an explicit consideration in our experiments was the question of *time*. For ART, Arcuri and Briand [2] have pointed out that if time is taken into account instead of the number of tests, then ART can be inferior to conventional random testing because it is able to execute many more tests in the same amount of time, which may compensate for any advantage gained by ART. Of course, it should also be noted that executing many more tests has its own cost, in the sense that these additional tests have to be checked by an oracle, which may not be automated.

This question of time is however a pertinent one for TBC (indeed, it applies to every LBT technique). Aside from the time taken to execute the tests, TBC also requires time to infer models (and the time taken for this can also increase with the number of tests). The time taken to execute a test is heavily dependent on variables that were not controlled in this experiment. The subject system is another influencing factor; for the sake of mutation testing, we have focussed on Java units that have a uniformly low execution time. It also crucially depends on the choice of model inference framework. Our choice of GP has offered a straightforward basis for computing uncertainty, but can be time-consuming. There are potentially speedier model inference alternatives (such as decision trees), where the use of ensemble techniques can also be used to derive uncertainty measures [29].

Finally, our empirical results have focussed entirely on the high-level question of how good the resulting test sets are at killing mutants. The relationship between this and the accuracy of the inferred model is currently presumed. Establishing whether this relationship indeed exists in reality would enable a more informed choice of model-inference technique. These questions are all the subject of a broader, more in-depth experiment which is part of our ongoing work (see Section VII).

VII. Conclusions and Future Work

In this paper we have made an explicit connection between the problems of test data generation in Software Engineering and sampling in active Machine Learning. Our solution proposes the use of uncertainty sampling as a means by which to generate suitable test data. We have provided a proof-of-concept implementation, along with the results of an empirical exploration using eight units within the Apache Commons Math and JodaTime frameworks. The initial results are encouraging. Our TBC approach outperforms regular and adaptive random testing.

Although promising, the approach has also given rise to several important questions, which were touched upon in the study. Our ongoing work is seeking to build upon our experiments in such a way that time is taken into account. To fully explore this question, we will seek to identify a broader range of software systems, including network protocols, web systems, and mobile apps for example, where the time taken to execute individual test cases can potentially be very high. We will also look at alternative model inference techniques (such as some of the baseline approaches used in our previous work [14]) and the potential for using ensemble-based approaches to derive uncertainty measures from them. A more extensive experiment will also seek to explore the specific relationship between the variable-range constraints, the initial amounts of test data, and the effectiveness of the generated tests.

In our ongoing and future work we will seek to explore these questions. We will carry out experiments to examine the effect of variable range on the number of mutants killed. We will look at the accuracy of the inferred model to see if, in this context, it leads to better test sets (building upon our previous work [14]), and we will explore the incorporation of multi-objective optimisation algorithms to ensure that all inferred models are good approximations of the whole set of observed test sets. We will also investigate the adoption of alternative Machine Learning algorithms that can model more sophisticated types of functionalities, such as complex data structures and sequential behaviour. This work on model-inference techniques will include an investigation of the relationship between model accuracy and the ability to kill mutants.

References

[1] D. Angluin. learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[2] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 265–275. ACM, 2011.

[3] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[4] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Transactions on Software Engineering and Methodology*, 5(2):119–145, 1996.

[5] C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[6] L. Briand, Y. Labiche, Z. Bawar, and N. Spido. Using machine learning to refine category-partition test specifications and test suites. *Information and Software Technology*, 51:1551–1564, 2009.

[7] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[8] T. Y. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer, 2005.

[9] J. Cherniavsky and C. Smith. A recursion theoretic approach to program testing. *IEEE Transactions on Software Engineering*, 13, 1987.

[10] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[11] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[12] T. G. Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[13] L. Feng, S. Lundmark, K. Meinke, F. Niu, M. A. Sindhu, and P. Y. Wong. Case studies in learning-based testing. In *IFIP International Conference on Testing Software and Systems*, pages 164–179. Springer, 2013.

[14] G. Fraser and N. Walkinshaw. Assessing and generating test sets in terms of behavioural adequacy. *Software Testing, Verification and Reliability*, 25(8):749–780, 2015.

[15] K. Ghani and J. Clark. Strengthening inferred specifications using search based testing. In *International Conference on Software Testing Workshops (ICSTW)*. IEEE, 2008.

[16] R. Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[18] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[19] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 612–615. IEEE Computer Society, 2011.

[20] G. D. Kader. Means and mads. *Mathematics Teaching in the Middle School*, 4(6):398, 1999.

[21] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[22] K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In *Tests and Proofs*, pages 134–151. Springer, 2011.

[23] P. Melville and R. J. Mooney. Diverse ensembles for active learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 74. ACM, 2004.

[24] P. Papadopoulos and N. Walkinshaw. Black-box test generation from inferred models. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2015 IEEE/ACM 4th International Workshop on*, pages 19–24. IEEE, 2015.

[25] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[26] K. R. Popper and G. E. Hudson. Conjectures and refutations, 1963.

[27] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *Formal Aspects of Software Engineering (FASE)*, LNCS, pages 377–380, 2006.

[28] K. Romanik. Approximate testing and its relationship to learning. *Theoretical Computer Science*, 188(1-2):175–194, 1997.

[29] B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. 2012.

[30] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 287–294. ACM, 1992.

[31] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[32] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[33] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In *Formal Methods (FM)*, LNCS, pages 305–320. Springer, 2009.

[34] E. J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):641–655, 1983.

[35] H. Zhu, P. Hall, and J. May. Inductive inference and software testing. *Software Testing, Verification, and Reliability*, 2(2):69–81, 1992.