# Behaviourally Adequate Software Testing

Gordon Fraser
*Saarland University – Computer Science*
*Saarbrücken, Germany*
*fraser@cs.uni-saarland.de*

Neil Walkinshaw
*Department of Computer Science*
*The University of Leicester*
*Leicester, UK*
*n.walkinshaw@leicester.ac.uk*

***Abstract*—Identifying a finite test set that adequately captures the essential behaviour of a program such that all faults are identified is a well-established problem. Traditional adequacy metrics can be impractical, and may be misleading even if they are satisfied. One intuitive notion of adequacy, which has been discussed in theoretical terms over the past three decades, is the idea of behavioural coverage; if it is possible to infer an accurate model of a system from its test executions, then the test set must be adequate. Despite its intuitive basis, it has remained almost entirely in the theoretical domain because inferred models have been expected to be exact (generally an infeasible task), and have not allowed for any pragmatic interim measures of adequacy to guide test set generation. In this work we present a new test generation technique that is founded on behavioural adequacy, which combines a model evaluation framework from the domain of statistical learning theory with search-based white-box test generation strategies. Experiments with our BESTEST prototype indicate that such test sets not only come with a statistically valid measurement of adequacy, but also detect significantly more defects.**

***Keywords*-test case generation; search-based testing; test adequacy; search-based software engineering**

## I. INTRODUCTION

To test a software system it is necessary to (a) determine the properties that constitute an adequate test set and (b) identify a finite test set that fulfils these adequacy criteria. These two questions have featured prominently in software testing research since they were first posed by Goodenough and Gerhart in 1975 [1]. They define an adequate test set to be one that *implies no errors in the program if it executes correctly*. In the absence of a complete and trustworthy specification or model, adequacy is conventionally measured according to proxy-measures of actual program behaviour. The most popular measures are rooted in the source code – these include branch, path, and mutation coverage.

Such measures are hampered because there is a fundamental leap between source code syntax and observable program behaviour. Ultimately, test sets that fulfil these criteria can omit crucial test cases, and quantitative assessments can give a misleading account of the extent to which program behaviour has really been explored.

In this paper, we take an alternative view of test set adequacy, following an idea first proposed by Weyuker in 1983 [2]: If we can correctly infer the general behaviour of a system by observing the behaviour elicited by a test set, then it can be deemed to be adequate. The appeal of this approach lies in the fact that it is concerned with *observable program behaviour*, as opposed to some proxy source-code approximation. However, despite this intuitive appeal, widespread adoption of this approach has been hampered by the dual problems that (a) the capability to infer accurate models has been limited, and (b) establishing the equivalence between a model and a program is generally undecidable.

The challenge of assessing the equivalence of inferred models with their hidden counterparts is one of the major avenues of research in the field of Machine Learning. In 1984 this gave rise to Valiant's Probably Approximately Correct (PAC) learning framework [3] – an evaluation framework for model inference techniques that can be used to attribute a statistically valid assessment of the accuracy to an inferred model. This ability to quantify model accuracy in a justifiable way presents an opportunity to make Weyuker's idea of inference-driven test adequacy a practical reality.

In this paper we introduce the BESTEST (BEhavioural Software TESTing) approach, which applies the ideas behind PAC to the challenge of assessing the adequacy of test sets and generating adequate test sets. It builds upon our earlier work on behavioural adequacy [4] by making the following contributions:

- It presents an enhanced measure of adequacy that is founded upon program behaviour, but also incorporates complementary code coverage information.
- It shows how the inference-based adequacy measure can be used to assess test sets for systems that take (potentially complex) data inputs and produce a data output (Section III).
- It presents a search-based test generation technique that will automatically generate test sets that are optimised with respect to this criterion (Section IV).
- It includes an empirical study on several Java units, demonstrating that the technique is practical for a wide range of systems and is effective at producing rigorous, truly adequate test sets (Section V).

## II. Syntax-Based Test Set Adequacy

When reduced to reasoning about program behaviour in terms of source code alone, it is generally impossible to predict with any confidence how the system is going to behave [5]. Despite this disconnect between code and behaviour, test adequacy is still commonly assessed purely in terms of syntactic constructs. Branch coverage measures the proportion of branches executed, path coverage measures the proportion of paths, mutation coverage measures the proportion of syntax mutations that are discovered.

These approaches are appealing because they are based on concepts every programmer understands; for example, it is straightforward to add new tests to improve branch coverage. However, the validity of these approaches is dubious because the precise relationship between a syntactic construct and its effect on the input/output behaviour of a program is generally impossible to ascertain. Branches and paths may or may not be feasible. Mutations may or may not change program behaviour. Loops may or may not terminate.

Even if these undecidability problems are set aside and one temporarily accepts that it *is* possible to cover all branches and paths, and that there are no equivalent mutants, there still remains the problem that these measures remain difficult to justify. There is at best a tenuous link between coverage of code and coverage of observable program behaviour (and the likelihood of exposing any faults). These measures become even more problematic when used as a basis for measuring *how adequate* a test set is. It is generally impossible to tell whether covering 75% of the branches, paths or mutants implies the exploration of 75% of the observable program behaviour; depending on the data-driven dynamics of the program it could just as well be 15% or 5%.

Some of these problems are illustrated with the bmiCategory example in Figure II. The test set in the table achieves branch and path coverage, but fails to highlight the bug in line 5; the inputs do not produce a BMI greater than 21 and smaller than 25 that would erroneously output "overweight" instead of "normal". Although mutation testing is capable in principle of highlighting this specific inadequacy, this depends on the selection of mutation operators and their quasi-random placement within the code—there is no means by which to establish that a given set of mutations collectively characterises what should be a truly adequate test set.

The fact that the given test set is unable to find this specific fault is merely illustrative. There is a broader point; *source code coverage does not imply behavioural coverage*, and is not in itself a justifiable adequacy criterion. If a test set claims to fully cover the behaviour of a system, it ought to be possible to derive an accurate picture of system behaviour from the test inputs and outputs alone [2], [6], [7]. A manual inspection of only the inputs and outputs of the BMI example tells us virtually nothing about the BMI system; one could guess that increasing the height can lead to a change in output category. However, it is impossible to accurately infer the relationship between height, weight, and category from these five examples. Despite being nominally adequate, they fail to sufficiently explore the behaviour of the system.

```
1  public String bmiCategory(double height, double weight){
2        double bmi = weight / (height*height);
3        if(bmi < 18.5)
4              return ''underweight'';
5        else if(bmi<21) //bug − should be (bmi<25)
6              return ''normal'';
7        else if(bmi<30)
8              return ''overweight'';
9        else if(bmi < 40)
10             return ''obese'';
11       else return ''very obese'';
12 }
```

| height | weight | *bmi* | output |
|---|---|---|---|
| 2 | 70 | *17.5* | "underweight" |
| 1.9 | 75 | *20.776* | "normal" |
| 1.8 | 85 | *26.23* | "overweight" |
| 1.7 | 90 | *31.14* | "obese" |
| 1.6 | 110 | *42.97* | "very obese" |

Figure 1. bmiCategory example that calculates the body mass index (BMI), and a test set for the BMI example that achieves branch and path coverage.

## III. Behavioural Test Set Adequacy

The idea of *behavioural adequacy* is founded on the idea that, if a test set is to be deemed adequate, it should contain enough information to capture the full range of program behaviour. In other words, it should be possible to infer the program behaviour from the test set. In this context, the term *behaviour* refers to the relationship between the possible inputs and outputs of a program. The concrete representation of this will vary depending on the nature of the program; a sequential control driven system could be modelled as a Finite State Machine, a data function might be represented by a differential equation, or a decision tree.

### A. Current Approaches to Behavioural Adequacy and their Limitations

The idea of adopting this perspective to assess test adequacy was first proposed by Weyuker [2], who developed a proof-of-concept system that inferred LISP programs to fit input / output data. Since then, the idea of combining model inference with software testing has been comprehensively explored in several theoretical and practical contexts [8], [6], [7], [9], [10], [11], [12], [13]. Much of this work has focussed on the appealing, complementary relationship between program testing and machine learning. The former is concerned with finding suitable inputs and outputs to exercise a given model of some hidden system, and the latter infers models from observed inputs and outputs. Together, the two disciplines can be combined to form a 'virtuous loop' where (at least in principle) it would be possible

to fully automate the complete exploration of software behaviour.

A key factor that has prevented the widespread use of behavioural adequacy has been its practicality. So far, approaches have sought to make an adequacy *decision*, rather than a obtain a quantitative *measurement*. Models are deemed either accurate or inaccurate, accordingly test sets must either be adequate or inadequate. This is problematic because the tasks of inferring an exact model and testing a model against a system are often either NP complete or NP hard. In practice, this means that the combined processes of inference and testing tend to require infeasibly large numbers of test cases to converge upon the final adequate test set. If on the other hand a cheaper inference process is adopted that allows for an inexact model (c.f. previous work by Walkinshaw *et al.* [13]), there has been no reliable means by which to gauge the accuracy of the final model, and to assess the adequacy of the final test set.

### B. The Probably Approximately Correct (PAC) Framework

The above problems of expense and accuracy have formed the basis for a substantial body of research in the Machine Learning community. Much of this research has been carried out under the heading of *Statistical* or *Computational Learning Theory* [14]. In this context, Valiant's popular *Probably Approximately Correct* (PAC) framework [3] has been used extensively to reason in statistical terms about the learnability of various types of concept, or to provide a sound basis for drawing statistically justified conclusions about the accuracy of inferred models. PAC describes a basic learning setting, where the key factors that determine the success of a learning outcome are characterised in probabilistic terms. As a consequence, if it can be shown that a specific type of learner fits this setting, important characteristics such as its accuracy and expense with respect to different sample sizes can be reasoned about probabilistically. Much of the notation used here to describe the key PAC concepts stems from Mitchell's introduction to PAC [14].

The PAC setting assumes that there is some *instance space* $X$. For a software system this would be the infinite set of all (possible and impossible) combinations of inputs and outputs. A *concept class* $C$ is a set of concepts over $X$, or the set of all possible models that consume the inputs and produce outputs in $X$. The nature of these models depends on the software system; for sequential input/output processors $C$ might refer to the set of all possible finite state machines over $X$. For systems such as the BMI example, $C$ might refer to the set of all possible decision trees [14].

A *concept* $c \subset X$ corresponds to a specific target within $C$ to be inferred (we want to find a specific subset of relationships between inputs and outputs that characterise our software system). Given some element $x$ (a given combination of inputs and outputs), $c(x) = 0$ or 1, depending on whether it belongs to the target concept (conforms to the behaviour of the software system or not). The conventional assumption in PAC is that there exists some selection procedure $EX(c,\mathcal{D})$ that randomly selects elements in $X$ following some static distribution $\mathcal{D}$ (we do not need to know this distribution, but it must not change).

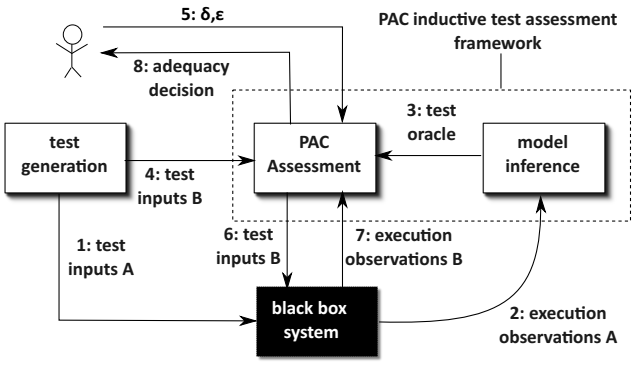


Figure 2. PAC-driven test adequacy assessment [4]

The basic learning scenario is that some learner is given a set of examples as selected by $EX(c,D)$. After a while it will produce a hypothesis $h$. The error rate of $h$ subject to distribution $\mathcal{D}$ ($error_{\mathcal{D}}(h)$) can be established with respect to a further 'test' sample from $EX(c,\mathcal{D})$. This represents the probability that $h$ will misclassify one of the test samples, i.e. $error_{\mathcal{D}}(h) \equiv Pr_{x \in \mathcal{D}}[c(x) \neq h(x)]$.

In most practical circumstances, a learner that has to guess a model given only a finite set of samples is susceptible to making a mistake. Furthermore, given that the samples are selected randomly, its performance might not always be consistent; certain input samples could happen to suffice for it to arrive at an accurate model, whereas others could miss out the crucial information required for it to do so. To account for this, the PAC framework enables us to explicitly specify a limit on (a) the extent to which an inferred model is allowed to be erroneous to still be considered approximately accurate, and (b) the probability with which it will infer an approximate model. The error parameter $\epsilon$ that puts an upper limit on the probability that an inferred model may mis-classify a given input. The $\delta$ parameter denotes an upper bound on the probability of a failure to infer a model (within the error bounds).

### C. Using PAC to Quantify Behavioural Adequacy

The PAC framework presents an intuitive basis for reasoning about test adequacy. Several authors have attempted to use it in a purely theoretical setting to reason about "testability", or to reformulate syntax-based adequacy axioms [6], [7], [10]. More recently, Walkinshaw [4] showed how the PAC framework could be incorporated into a more general, statistically sound basis for assessing test set adequacy and to place bounds on the number of tests required to produce an adequate test set.

In Figure 2 the arcs are numbered to indicate the flow of events. The test generator produces tests according to some fixed distribution $\mathcal{D}$ that are executed on the SUT $c$. With respect to the conventional PAC framework they combine to perform the function of $EX(c,\mathcal{D})$. The process starts with the generation of a test set $A$ by the test generator (this is what we are assessing for adequacy). These are executed on the SUT, the executions are recorded and supplied to the inference tool. This infers a hypothetical test oracle. Now, the test generator supplies a further test set $B$. The user may supply the acceptable error bounds $\epsilon$ and $\delta$ (without these the testing process can still operate, but without conditions for what constitutes an adequate test set). The observations of test set $B$ are then compared against the expected observations from the model to compute $error_{\mathcal{D}}(h)$. If this is smaller than $\epsilon$, the model inferred by test set $A$ can be deemed to be *approximately accurate* (i.e. the test set can be deemed to be *approximately adequate*).

The $\delta$ parameter is of use if we want to make broader statements about the effectiveness of the combination of learner and test generator. By running multiple experiments, we can count the proportion of times that the test set is approximately adequate for the given SUT. If, over a number of experiments, this proportion is greater than or equal to $1-\delta$, it becomes possible to state that, in general, the test generator produces test sets that are *probably approximately adequate* (to paraphrase the term 'probably approximately correct', that would apply to the models inferred by the inference technique in a traditional PAC setting).

### D. Behavioural Adequacy for White-box Testing

The PAC process described above will produce a test set that is adequate in an empirical sense. It treats the software as a black box, collects a sample of observations, produces a hypothesis, and validates the hypothesis on an independent set of further observations. Although this is statistically valid, there remains the danger that the sampling process that produced the two sets misses out test cases that happen to exercise a crucial facet of program behaviour. This is particularly problematic if the SUT has a wide range of outputs that depend a highly specific set of inputs, which are unlikely to be triggered without prior knowledge.

Previous work [4] thus assumed sufficiently large test sets, such that there is a high probability that all behaviour is triggered. However, when we are testing without a formal specification we are interested in producing *small* test sets, such that the developer can manually assess test outcomes, or can add test oracles in terms of test assertions.

To overcome this issue, we take a white-box view on the traditional black-box PAC setting: If we know that some part of the code has not even been executed, then clearly the adequacy measurement cannot relate to the behaviour related to that code. Consequently, a minimal requirement to properly assess adequacy is that all of the code is executed–in other words, statement or branch coverage are a prerequisite for adequacy.

## IV. Generating Adequate Test Sets

Based on the notion of test set adequacy, we now turn to the question of how to produce such test sets. In this section, we describe a search-based technique to automatically generate adequate test sets.

### A. Search-based Testing

The use of search techniques to produce test cases is commonly referred to as search-based testing. A popular meta-heuristic applied in this context is a genetic algorithm, which imitates the processes of natural evolution in order to produce solutions to an optimization problem. A population of candidate solutions (chromosomes) is evolved by selecting individuals for reproduction based on their fitness for the given problem. For example, in the context of software testing the individuals of the population are candidate test cases or test suites, and the fitness value estimates the suitability with respect to a coverage criterion. Selected individuals reproduce using crossover and mutation operators, and as the evolution procedes the population gradually evolves towards better fitness values, until a solution is found or a predetermined resource limit has been reached.

### B. Problem Representation

In our context, we are aiming to produce adequate test sets. To determine adequacy of a given test set, we need a second test set which we can check against a model inferred from the first test set. Thus, the chromosomes in our search are pairs of test sets $\langle T_1,T_2\rangle$. A test set $T$ is a set of test cases $t_i$, and a test case is a value assignment for the input parameters of the target function. The number of tests in a test set is not fixed, such that it can vary through the application of search operators, but has an upper bound $B_T$. Based on this representation, we need to define the search operators of crossover and mutation, and we need to define how the initial population is derived.

*1) Crossover:* Crossover between two parent individuals $P_1$ and $P_2$ produces two offspring $O_1$ and $O_2$, such that genetic material is exchanged between the parents. However, in our context it is not desirable that genetic material is exchanged between the first (test set) and the second (validation set). Therefore, crossover of $P_1 = \langle T_1,T_2\rangle$ and $P_2 = \langle T_1',T_2'\rangle$ results in offspring $O_1 = \langle T_1,T_2'\rangle$ and $O_2 = \langle T_1',T_2\rangle$.

*2) Mutation:* The aim of mutation is to introduce new genetic material in the search, in order to support the exploration aspect of the search. When mutating a pair of test sets $\langle T_1,T_2\rangle$ we always mutate both test sets $T_1$ and $T_2$. When mutating a test set, we can either remove, add, or mutate individual tests. When inserting tests, we insert a new random test case to $T_i$ with probability $\sigma$. If it is added,

then a second test case is added with probability $\sigma^2$, and so on, until the $i$th test case is not added. This type of mutation respects the upper bound on the number of test cases in a set, so once a test set has reached its maximum bound $B_T$, then no new tests are added. When deleting tests, each test is deleted with probability $1/n$. Finally, each test in a set of $n$ tests is mutated with probability $1/n$.

When mutating a test case $t = \langle p_1, \ldots p_n \rangle$, each parameter is mutated with probability $1/n$. The mutation of the value depends on its type: For example, numeric parameters are mutated by adding a delta using a Gaussian distribution around 0, while string values are mutated by randomly inserting, deleting, or changing characters.

*3) Initial population:* The initial population of the search consists of randomly generated chromosomes. For each test set in a pair, we select a random number $n = [T_{min}, T_{max}]$, and then generate $n$ test cases. For each test case, we assign random values to the parameters of the method under test.

### C. Fitness Function

A fitness function determines how good a given chromosome is with respect to the optimization goal, and thus guides the search by providing feedback which operations lead to better individuals. In our context, the goal is to produce a test set that adequately exercises the target method; these are actually two distinct objectives: first, to execute *all* the code of the method, and second, to *adequately* do so.

*1) Code coverage:* The first objective is a traditional goal in test generation, as a prerequisite to find errors in a piece of code is that this piece of code is actually executed in the first place. This is articulated in the traditional *statement* and *branch* coverage metrics: Statement coverage requires that all statements of a program are executed, while branch coverage additionally requires that all predicates in the program evaluate to true and to false.

We assume that a minimum requirement for any adequate test set is that all feasible branches in the program have been executed. Achieving branch coverage is a classical objective in search-based testing, and the literature has treated this problem sufficiently. A common metric to estimate how close a logical predicate in the program was to evaluating to true or to false is the *branch distance* [15]: The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see [15] for details). For example, for predicate $x \geq 10$ and $x$ having the value 5, the branch distance to the true branch is $10 - 5 + k$, with $k \geq 1$. When targeting individual branches, one can further use the *approach level* [15], which counts how many control dependencies need to be satisfied additionally before the target branch is reached.

In contrast, when targeting all branches at the same time (e.g., [16]) it is sufficient to simply sum up the individual branch distances. In this case, we also require that each branch has been executed twice, such that it is possible to cover both, the true and the false branch. This results in the following definition of the branch distance $d(b,T)$ for branch $b$ for a given test set $T$:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

Here, $\nu(x)$ is a normalizing function in $[0,1]$, e.g. ; we use the normalization function [17]: $\nu(x) = x/(x+1)$. When generating adequate test sets, we require that the branch coverage for all branches in the set of target branches $B$ is maximized, i.e., the branch distance is minimized:

$$\text{cov}(T) = \sum_{b_k \in B} (1 - d(b_k,T))$$

*2) Behavioural adequacy:* Once all the code has been reached, the second objective expresses how thoroughly this code has to be exercised with respect to its externally observable behaviour. This part of the fitness function is based on the PAC process shown in Figure 2. We infer a model $M$ of the behaviour exhibited by test set $T_1$, and measure the extent to which $T_2$ agrees with $M$.

The specific nature of this comparison depends entirely on the type of SUT and the type of model used to represent its behaviour. There exists a plethora of inference algorithms that are suited to specific types of system and input/output characteristics. The SUTs considered in the case studies in this paper all take numerical inputs and return either numerical or categorical outputs. Accordingly, we have selected two well-established inference algorithms that suit such systems. Both infer models in the form of decision trees (nested if-then-else statements). For SUTs that take numerical inputs and produce categorical outputs (e.g. the BMI example), we use the C4.5 algorithm [18]. For SUTs that take numerical inputs and return numerical outputs, we selected the M5 algorithm [19].

The task of measuring the overlap between $M$ and $T_2$ also depends on the type of model $M$. If $M$ produces categorical outputs, the overlap between the outputs produced by $M$ and $T_2$ is be measured with standard binary classification assessment measures such as Precision and Recall or the kappa statistic (used for the results in this paper). For numerical outputs it is similarly possible to use standard statistical measures such as the correlation coefficient, which is used in this paper.

*3) Adequacy measurement:* This results in an adequacy measurement $A$ in the range $[0,1]$, where adequacy 0 denotes an inadequate test set. Often, an adequacy value of 0 simply means that the number of samples is too small to draw any conclusions. Therefore, whenever adequacy is 0 we include the size the test set in the fitness function, such that the

growth of test sets is favoured until adequacy can be reliably determined. We do this by normalizing the size of $T_1$ in the range [0,1] if adequacy is 0, and by adding 1 if adequacy is $> 0$ such that now growth is required:

$$\mathcal{A}(\langle T_1,T_2\rangle) = \begin{cases} 1 + A & \text{if } A > 0, \\ \nu(|T_1|) & \text{otherwise} \end{cases}$$

The $\mathcal{A}$ measurement can also take into account how much the test sets $T_1$ and $T_2$ differ, penalizing a difference in size beyond a given threshold.

Finally, this results in the following fitness function:

$$\text{fitness}(\langle T_1,T_2\rangle) = \alpha \cdot \mathcal{A}(T_1,T_2) + \beta \cdot \text{cov}(T_1) + \beta \cdot \text{cov}(T_2)$$

The values $\alpha$ and $\beta$ are used to weight coverage against adequacy; for example, in our experiments, we used $\beta = 1000$, and $\alpha = 1$.

### D. Handling Test Set Size

The number of tests in a test set is variable up to the upper bound $B_T$. The accuracy of inferred models tends to be greater if the given set of tests is broad and diverse. Accordingly, the fitness function will tend to favour large test sets. Ultimately the minimal size required to infer an accurate model depends on the behavioural characteristics of the SUT (as is the case with the minimal number of tests). This can however lead to issues of efficiency. If a test set grows too large it can hamper the search process, requiring too many resources. Furthermore, commonly the resulting test set is provided to the developer, who then has to provide test oracles (e.g., assertions), as automated oracles are seldom available in practice. For this task it is important to ensure that a test set is as small as possible.

To encourage this, we use a secondary search objective that seeks to reduce the size of test sets. We use rank selection, where test sets with identical fitness are ranked according to their size, such that the smaller test set will have a better rank. This way, even though the search can explore large test sets to achieve adequacy, as the search proceeds the size of the adequate sets is reduced to a minimum.

### E. The Independence of $T_1$ and $T_2$

The PAC evaluation process is based on the premise that the samples $T_1$ and $T_2$ are drawn independently from each other. If this is not the case, it threatens the reliability of the resulting adequacy measurement. In statistical terms this threat is referred to as *sampling bias*; a pair of samples is selected according to a biased strategy that tends to produce samples that are not truly representative, which leads to skewed conclusions.

In our case, both sets are evolved such that the model inferred by $T_1$ returns the greatest score with respect to $T_2$. The danger here is that, instead of favouring test sets that spur the inference and evaluation of accurate models, the test sets are merely encouraged to be similar to each other in a shallow sense without exercising the system. As a consequence, there is a danger that this leads to adequacy scores that are artificially high.

This risk is attenuated somewhat by the large degree of stochasticity in the test set generation process – the likelihood that two test sets end up with the same test cases is low for most systems. In our experiments (see following section), out of 40,728 tests, only 409 (1%) were identified as being equivalent. Breaking this possible source of bias is the subject of ongoing work.

Table I
STUDY SUBJECTS

| Name | Source | #Lines | #Branches |
|---|---|---|---|
| BMICalculator | [4] | 17 | 9 |
| CalDate | [21] | 25 | 7 |
| Evaluation | [20] | 33 | 3 |
| Expint | [22] | 51 | 31 |
| Fisher | [23] | 49 | 17 |
| Gammq | [22] | 71 | 27 |
| Middle | [21] | 19 | 29 |
| Remainder | [24] | 33 | 25 |
| TicTacToe | [4] | 69 | 45 |
| Triangle | [15] | 25 | 17 |
| TCAS | [25] | 99 | 78 |
| WrapRoundCounter | [21] | 9 | 3 |

## V. EVALUATION

To evaluate the effects and implications of behavioural coverage, we have implemented the BESTEST prototype based on the EVOSUITE [16] framework, using the WEKA [20] toolkit for model inference. BESTEST takes as input a Java class, and produces a behaviourally adequate test set for each of its methods. Currently, the inference step is limited to parameters of primitive type and primitive return values, though we will extend the implementation to general data structures and stateful types in future work.

### A. Experimental Setup

We identified suitable case study subjects from the testing literature, resulting in the set of classes summarized in Table I. Each class contains only one top level public method, but may include further code in private methods not directly callable.

During our experiments we encountered an interesting (albeit unintended) case demonstrating the usefulness of search-based testing. Our fitness function is based on the adequacy measurement, which internally is based on the calculation of a correlation coefficient in WEKA. Although the maximum value for a correlation coefficient should be 1, in several examples, the search would optimize such that, when evaluated by the PAC process would elicit a correlation coefficient much larger than 1. It turned out that our search had identified a pathological case in the correlation coefficient calculation in WEKA, where the value can become unstable for small sample sizes and would return

a large value (possibly $> 1$) instead of 0 or NaN. For the experiments a corrected correlation coefficient implementation has been used, and the version from WEKA has been included in our set of case study subjects (labelled "Evaluation").

For each of the case study subjects, we ran BESTEST for 10 minutes to produce a set as adequate as possible; to account for the randomness of the search algorithm all experiments were repeated 30 times. In addition, we also measured the adequacy of traditional branch coverage test sets by using BESTEST without factoring behavioural adequacy into the fitness function. Furthermore, as a sanity check for the search, we generated sets of random test sets of a similar size to those produced by the search.

### B. Adequacy of Branch Coverage Test Sets

First, let us consider what level of behavioural adequacy test sets produced using only branch coverage can achieve. For this, we used the BESTEST prototype to evolve pairs of test sets, but the fitness would only try to maximize the branch coverage of the individual test sets. Adequacy is then assessed by following the PAC approach – using the second test set of each pair as a validation set for the model inferred from the first set. The top section of Table II lists the achieved levels of coverage as well as the resulting adequacy values for branch coverage test sets. In four cases adequacy is 0 (the inferred model could not predict anything in $T_2$), and in most other cases the adequacy value is very low. This clearly indicates that merely covering structural aspects of the source code does not imply coverage of the behaviour.

*Branch coverage test sets have a low behavioural adequacy.*

Table II
TEST SET STATISTICS

| | Name | Tests | Branch Coverage | Behavioural Adequacy | Mutation Score |
|---|---|---|---|---|---|
| **Branch Coverage** | BMICalculator | 5.0 | 0.89 | 0.05 | 0.85 |
| | CalDate | 2.0 | 1.00 | 0.00 | 0.76 |
| | Evaluation | 2.0 | 1.00 | 0.00 | 0.93 |
| | Expint | 9.7 | 0.96 | 0.44 | 0.51 |
| | Fisher | 22.07 | 1.00 | 0.00 | 0.91 |
| | Gammq | 8.0 | 0.81 | 0.60 | 0.52 |
| | Middle | 9.0 | 1.00 | 0.40 | 0.86 |
| | Remainder | 17.3 | 0.94 | 0.23 | 0.49 |
| | TicTacToe | 4.0 | 0.58 | 0.39 | 0.21 |
| | Triangle | 5.0 | 1.00 | 0.20 | 0.78 |
| | TCAS | 11.43 | 0.87 | 0.11 | 0.42 |
| | WrapRoundCounter | 2.0 | 1.00 | 0.00 | 0.81 |
| **Behavioural Adequacy** | BMICalculator | 75.6 | 0.89 | 0.92 | 0.92 |
| | CalDate | 19.7 | 1.00 | 1.00 | 0.82 |
| | Evaluation | 12.8 | 1.00 | 1.00 | 0.98 |
| | Expint | 35.6 | 0.95 | 0.95 | 0.56 |
| | Fisher | 29.0 | 1.00 | 0.03 | 0.95 |
| | Gammq | 41.7 | 0.81 | 1.00 | 0.61 |
| | Middle | 25.57 | 1.00 | 1.00 | 0.88 |
| | Remainder | 25.6 | 0.94 | 0.77 | 0.55 |
| | TicTacToe | 26.56 | 0.58 | 1.00 | 0.30 |
| | Triangle | 39.55 | 1.00 | 0.99 | 0.84 |
| | TCAS | 52.4 | 0.79 | 0.52 | 0.29 |
| | WrapRoundCounter | 24.3 | 1.00 | 1.00 | 0.87 |
| **Random Testing** | BMICalculator | 76.0 | 0.62 | 0.50 | 0.69 |
| | CalDate | 20.0 | 0.97 | 0.43 | 0.76 |
| | Evaluation | 13.0 | 1.00 | 0.12 | 0.98 |
| | Expint | 36.0 | 0.76 | 0.57 | 0.46 |
| | Fisher | 29.0 | 0.95 | 0.00 | 0.95 |
| | Gammq | 42.0 | 0.75 | 0.69 | 0.60 |
| | Middle | 26.0 | 0.86 | 0.38 | 0.69 |
| | Remainder | 26.0 | 0.67 | 0.25 | 0.37 |
| | TicTacToe | 27.0 | 0.56 | 0.27 | 0.29 |
| | Triangle | 40.0 | 0.74 | 0.06 | 0.57 |
| | TCAS | 52.0 | 1.00 | 0.00 | 0.09 |
| | WrapRoundCounter | 24.0 | 1.00 | 0.62 | 0.87 |

### C. Behaviourally Adequate Test Sets

Having seen that test sets optimized for branch coverage tend to be behaviourally inadequate, we would like to determine whether our BESTEST approach can lead to better test sets. The middle part of Table II summarizes the results of the experiments using BESTEST. In all cases the behavioural adequacy is significantly higher than in the case of the branch coverage test sets. This shows that the BESTEST test sets are *justifiable*; high adequacy values show that $T_1$ incorporates a sufficient amount of information about program behaviour to predict the output of test sets in $T_2$ that have not been observed yet.

It is interesting to note that the adequacy of the Fisher example is very low (0.03). This might have two reasons: First, the behaviour may be very complex, requiring more tests and more time to be fully explored. Alternatively, it might be that the combination of model and model inference technique used in the implementation BESTEST (i.e. using M5 to infer a decision tree) is not suited to infer the behaviour of this specific SUT. In this case, using a different combination might lead to better results. It is worth noting however that in such cases, the adequacy measure tends to be conservative; the PAC process will generally produce a low value as opposed to an artificially high one. Determining the best suited model for a given type of SUT is clearly part a future research direction.

*Search-based testing can produce test sets of high behavioural adequacy.*

### D. Effects of Adequacy on Test Set Size

We have seen that BESTEST produces tests of higher adequacy, but now the question is: At what price does adequacy come? To see the effects of adequacy on the costs of testing we look at the number of tests in the resulting test sets. Table II lists the average numbers of tests in the resulting test sets in our experiments. Averaged over all examples, branch coverage required 7.44 tests per set. In contrast, the test sets produced by BESTEST are significantly larger, with an average size of 34.69 tests. Thus, as expected

the increase in adequacy clearly comes with an increase in the number of tests.

*Higher adequacy requires significantly larger test sets than branch coverage.*

However, in practice one might not want to produce a test set that maximizes adequacy at any costs, but may have an upper bound on the number that is feasible. For example, if a tester has to manually assess correctness of test cases by adding assertions, the number of tests plays a crucial role. In this case, BESTEST could be used to produce test sets of a desired target level of adequacy (e.g., the objective could be to achieve 50% behavioural adequacy), or one could try to maximize the behavioural coverage of a test set of fixed size. In all cases a central benefit of the BESTEST approach is that whatever test set is produced, it comes with a statistically valid measurement of its adequacy.

### *E. Effects of Adequacy on Fault Detection Ability*

Of course, the question now is what the behavioural adequacy measurement really expresses. On an intuitive level, the idea of covering the behaviour of a program is easily understandable. This is essential, as for practicioners to adopt an adequacy measurement it needs to be intuitive – it is easy to understand a measurement that tells how many statements or branches have been taken, which is why these measurements are so popular in practice. In contrast, understanding mutation scores or dataflow coverage criteria is less intuitive, which probably contributes to their rare use. We argue that a measurement of the program behaviour is just as simple to grasp as simple code coverage metrics.

However, ease of understanding and intuition are difficult to evaluate without involving humans in experiments, and so for practical reasons we would like to measure whether higher adequacy actually means better ability to detect faults. For this, we resort to traditional mutation analysis: The more seeded defects (mutants) a test set can distinguish from the original program, the better it is at detecting real faults. For this reason, we calculated the mutation score for each of the produced test sets, using the mutation analysis component of the EVOSUITE testing tool. We consider a mutant to be killed when the output of the mutated function differs from the normal output. The results are presented in the last column of Table II.

It is important to bear in mind the discussion from Section II – syntax-based adequacy measures (such as mutation testing) are unreliable. Mutants can be equivalent and they might fail to capture the full range of program behaviour. We do not use the mutation score here as an absolute verdict on individual test sets, but merely to compare different test sets against each other – a test set that produces a greater mutation score is probably more rigorous.

To compare the mutation scores of behavioural adequacy, branch coverage and random tests, we measured statistical difference with the Mann-Whitney U test following the guidelines described by Arcuri and Briand [26]. To quantify the improvement in a standardized way, we used the Vargha-Delaney $\hat{A}_{12}$ effect size [27]. In our context, the $\hat{A}_{12}$ is an estimation of the probability that, if we use the test sets produced by BESTEST, we will obtain better mutation score than using the branch coverage test sets. When two randomized algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ means that, in *all* of the 30 runs of BESTEST, we obtained mutation scores higher than the ones obtained in *all* of the 30 runs of branch coverage. Table III lists the resulting $\hat{A}_{12}$ values for the comparison between branch coverage tests and the test sets with higher adequacy. The improvement in the mutation score is striking, as in all but one case the behaviourally adequate test sets are practically certain to achieve higher mutation scores (statistically significant with $\alpha < 0.05$).

Table III
$\hat{A}_{12}$ MEASURE VALUES IN THE MUTATION SCORE COMPARISONS: $\hat{A}_{12} < 0.5$ MEANS BESTEST ACHIEVED LOWER, $\hat{A}_{12} = 0.5$ EQUAL, AND $\hat{A}_{12} > 0.5$ HIGHER MUTATION SCORES THAN THE RANDOM / BRANCH COVERAGE TEST SUITES. BOLD FONTS REPRESENT A STATISTICAL SIGNIFICANCE WITH $\alpha < 0.05$.

| Case Study | Branch Coverage | Random |
|---|---|---|
| BMICalculator | **0.99** | **0.98** |
| CalDate | **0.85** | **0.65** |
| Evaluation | **1.00** | **0.25** |
| Expint | **1.00** | **1.00** |
| Fisher | **0.78** | **0.74** |
| Gammq | **0.94** | 0.45 |
| Middle | **0.97** | **0.98** |
| Remainder | **0.87** | **0.99** |
| TicTacToe | **0.99** | 0.63 |
| Triangle | **0.87** | **1.00** |
| TCAS | **0.06** | **0.96** |
| WrapRoundCounter | **0.84** | 0.47 |
| Mean: | 0.85 | 0.76 |

*Test sets optimized for behavioural adequacy detect more mutants than those optimized for branch coverage.*

The TCAS example is an interesting exception in this analysis, as in this case the probability of achieving a higher mutation score is very low, even though the behavioural adequacy of the test sets produced by BESTEST is higher. However, this can be explained by considering the level of code coverage: The branch coverage achieved by the test sets produced for branch coverage is higher than that of the test sets produced by BESTEST. This means that even though the adequacy of the behavior tested by the BESTEST test set is higher, there is a significant amount of behavior that is missed by this test set. Clearly, if more branches are executed this makes it possible to detect more mutants. This demonstrates the importance of the white-box perspective discussed in Section III-D. The lower branch coverage of the test set produced by BESTEST can be attributed to the fact

that more of the search effort is invested in the exploration of the adequacy aspect, rather than the branch coverage. It is likely that this would be substantially improved by allowing the search to proceed for longer than 10 minutes.

### F. Adequacy as Fitness Function for Search-based Testing

As the number of tests in the test sets produced by BESTEST is significantly higher than that of the branch coverage test sets, the question arises whether the improvement in adequacy and mutation score we observed is simply a consequence of the size increase, or whether the adequacy does actually guide towards better test sets. To establish whether this is the case we generated 30 pairs of random test sets for each of the case study examples, each of the same size as the mean size of the test sets produced by BESTEST. The results are summarized in the lower part of Table II. In most cases, the branch coverage is lower than in the other test sets, and the behavioural adequacy is significantly lower than in all sets produced by BESTEST. Not surprising, in some cases the behavioural adequacy is higher than that of the branch coverage test sets. The effectiveness in terms of detecting mutants is summarized in Table III (last column): In nine out of 12 cases the mutation score is higher for BESTEST (statistically significant in 8 cases), while in three cases it is lower. However, only in a single case is the lower mutation score statistically significant (Evaluation); however, looking at mean mutation scores, we see that this difference is 0.5% which, despite its statistical significance, is acceptably small.

*Test sets optimized for behavioural adequacy detect more mutants than random test sets of the same size.*

These results can only be treated as indicative; there remains the possibility that this increase in performance could be due to some unaccounted combination of branch coverage and size. Capturing the precise properties that contribute to behavioural adequacy underpins a significant part of our ongoing work.

### G. Threats to Validity

Threats to *construct validity* are on how the performance of a testing technique is defined. Traditionally, test generation techniques are compared in terms of the achieved code coverage; in our scenario, this is difficult as we are arguing about the inadequacy of such criteria. As a proxy measurement we use mutation testing. As mentioned previously, mutation testing is susceptible to the same problems as code coverage (how adequate is 80% mutation score?).

Threats to *internal validity* might arise from the method used for the empirical study. To reduce the probability of having faults in our testing framework, it has been carefully tested. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 30 times, and we followed rigorous statistical procedures to evaluate their results. We limited the search to 10 minutes when generating both, adequate and branch coverage test sets, which might lead to a biased results favoring the branch coverage test sets. However, it is difficult to determine what precisely would constitute a fair comparison.

Threats to *external validity* concern the generalization to other types of software, common for any empirical analysis. We have selected 12 different classes for evaluation, which arguably results in a small evaluation, such that the results might not generalize to all types of software. This was largely necessitated by the current limitations of the BESTEST prototype (e.g., limitation to primitive data types, model inference only for single outputs), which is the subject of ongoing work.

## VI. Conclusions and Consequences

Dijkstra famously stated that software testing can only show the presence of faults, but never their absence. General acceptance of this view has led to the conclusion that, if it is futile to attempt to demonstrate the absence of faults, then a test set should at least make sure that it achieves some notional level of "coverage". However, traditional structural criteria cannot serve as reliable gauge to tell the tester how rigorously a program has been tested.

In this paper we have argued for an intuitive idea dating back to Weyuker in 1983 [2]: If we can correctly infer the behaviour of a system from its test set, then we have tested the behaviour adequately. Exploiting Valiant's Probably Approximately Correct learning framework [3], we can put a statistically valid measurement to this adequacy. Given this approach, we can precisely estimate how much of the *behaviour* of a system is exercised by a test set, and we can use the measurement as guidance in automatically producing test sets of any desired level of adequacy.

Does 100% adequate mean 100% correct? Unfortunately, the answer to this question is no, for two reasons: The first reason is of course that behavioural adequacy only measures how much of the behaviour has been *executed*, but makes no statement about how much has actually been *checked*. The second reason is that, as discussed in Section III-D, there is the possibility that the sampling process misses out some aspects of the behaviour in all runs, leading to an overly optimistic adequacy measurement. We attenuate this risk by incorporating white-box techniques such as code coverage, but there remains a substantial scope for identifying further complementary code analysis techniques.

Our current BESTEST prototype serves as a proof-of-concept for the idea of using behavioural adequacy to drive test generation. Many aspects of this implementation need further consideration and evaluation, and in fact with this paper we open up a wealth of future research opportunities:

- The representation and search operators may be extended to better suite the problem and to increase independence between the individual test sets.

- A sensitivity analysis is necessary with respect to the parameters of the search (e.g. $\alpha$, $\beta$, $\delta$, $\epsilon$, etc.)
- The fitness function might be refined, e.g.to take the similarity between the individual sets into account.
- The adequacy measurement itself could be refined, e.g. by using more than one evaluation set at the same time.
- Our adequacy notion is based on branch coverage following the intuition that every branch needs to be executed as a minimal prerequisite to explore behaviour. However, other structural or data-flow criteria might better serve as baseline.
- Currently, we only use the size as a secondary objective during the search. However, as the size of test sets is important, dedicated minimization strategies for behaviourally adequate test sets need to be explored.
- Our current BESTEST prototype applies to methods that take primitive parameters as inputs and return a primitive value. We will extend this approach to apply to inputs and outputs of any complexity (e.g., instances of any class), which will require to use new model types and learning algorithms that can handle several outputs.

By replacing code coverage as a standard with behavioural adequacy, software testing has the potential for the first time to be driven by *meaningful* metrics. We believe that the availability of such metrics has the potential to profoundly change the landscape of software testing.

**Acknowledgments.** This project has been funded by Deutsche Forschungsgemeinschaft (DFG), grant Ze509/5-1, a Google Focused Research Award on "Test Amplification", and the DSTL-funded BATS project DSTLX1000062430.

## REFERENCES

[1] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*. ACM, 1975, pp. 493–510.

[2] E. Weyuker, "Assessing test data adequacy through program inference," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 641–655, 1983.

[3] L. Valiant, "A theory of the learnable," *Communications of the ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[4] N. Walkinshaw, "Assessing test adequacy for black-box systems without specifications," in *Proceedings of the International Conference on Testing Systems and Software (ICTSS'11)*, 2011.

[5] M. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[6] H. Zhu, P. Hall, and J. May, "Inductive inference and software testing," *Software Testing, Verification, and Reliability*, vol. 2, no. 2, pp. 69–81, 1992.

[7] H. Zhu, "A formal interpretation of software testing as inductive inference," *Software Testing, Verification and Reliability*, vol. 6, no. 1, pp. 3–31, 1996.

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

[9] F. Bergadano and D. Gunetti, "Testing by means of inductive program learning," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 119–145, 1996.

[10] K. Romanik, "Approximate testing and its relationship to learning," *Theoretical Computer Science*, vol. 188, no. 1-2, pp. 175–194, 1997.

[11] 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*, vol. 51, pp. 1551–1564, 2009.

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

[13] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris, "Increasing functional coverage by inductive testing: A case study," in *International Conference on Testing Software and Systems (ICTSS)*, ser. LNCS, 2010.

[14] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[15] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[16] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference On Quality Software (QSIC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40.

[17] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," 2011.

[18] J. R. Quinlan, *C4. 5: Programs for Machine Learning*. San Mateo, CA: MK, 1993.

[19] ——, "Learning with Continuous Classes," in *5th Australian Joint Conference on Artificial Intelligence*, 1992, pp. 343–348.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.

[21] K. Ghani and J. A. Clark, "Strengthening inferred specifications using search based testing," in *Proceedings of the International Workshop on Search-based Software Testing*. IEEE Computer Society, 2008, pp. 187–194.

[22] C. Schneckenburger and J. Mayer, "Towards the determination of typical failure patterns," in *4th International Workshop on Software Quality Assurance, co-located with ESEC/FSE'07 (SOQUA'07)*. ACM, 2007, pp. 90–93.

[23] E. Dorrer, "F-distribution," *Commun. ACM*, vol. 11, no. 2, pp. 116–117, 1968.

[24] H. Sthamer, "The automatic generation of software test data using genetic algorithms," Ph.D. dissertation, University of Glamorgan, Pontyprid, Wales, UK, April 1996.

[25] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, October 2005.

[26] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.

[27] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.