SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* 2015; **25**:749–780
Published online 20 March 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1575

# Assessing and generating test sets in terms of behavioural adequacy

Gordon Fraser[1,*,†] and Neil Walkinshaw[2,*,†]

[1]*Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, S1 4DP, Sheffield, UK*
[2]*Department of Computer Science, University of Leicester, University Road, LE1 7RH, Leicester, UK*

SUMMARY

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. This is traditionally addressed with syntactic adequacy metrics (e.g. branch coverage), but these 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 can be deemed to 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. This paper presents a practical approach to incorporate behavioural coverage. Our BESTEST approach (1) enables the use of machine learning algorithms to augment standard syntactic testing approaches and (2) shows how search-based testing techniques can be applied to generate test sets with respect to this criterion. An empirical study on a selection of Java units demonstrates that test sets with higher behavioural coverage significantly outperform current baseline test criteria in terms of detected faults. 

Received 30 April 2014; Revised 23 February 2015; Accepted 23 February 2015

KEY WORDS: test generation; test adequacy; search-based software testing

## 1. 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 quantified 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 often a chasm between the static source code syntax and dynamic, observable program behaviour. Ultimately, test sets that fulfil source code-based 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.

*Correspondence to: Gordon Fraser, University of Sheffield, Department of Computer Science, Regent Court, 211 Portobello, S1 4DP, Sheffield, UK; Neil Walkinshaw, Department of Computer Science, University of Leicester, University Road, LE1 7RH, Leicester, UK.

†E-mail: gordon.fraser@sheffield.ac.uk; n.walkinshaw@leicester.ac.uk

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 infer a model of the behaviour of a system by observing its outputs during the execution of a test set, and we can show that the model is accurate, it follows that the test set can be deemed to be adequate. The approach is appealing because it is concerned with *observable program behaviour*, as opposed to some proxy source-code approximation. However, despite this intuitive appeal, widespread adoption has been hampered by the problems that (a) the capability to infer accurate models has been limited, (b) establishing the equivalence between a model and a program is generally undecidable, and (c) there are no systematic approaches to deriving test sets that cover a program's behaviour.

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. Since the publication of Valiant's paper on *probably approximately correct (PAC)* learning [3], numerous techniques have been developed that, instead of aiming to establish whether or not an inferred model is exactly correct, aim to provide a more quantitative measurement of *how accurate* it is. 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.

This paper shows how, by applying the principles that underlie PAC learning, it is possible to develop a practical and reliable basis for generating rigorous test sets. This paper extends an earlier paper [4] that strictly followed the PAC principles in the BESTEST$_{PAC}$ approach, but in doing so leads to problems of large test sets and potential bias. To overcome this problem, this paper makes the following contributions:

- It presents the refined BESTEST$_{CV}$ technique, which adopts *k-folds cross validation*—a more pragmatic substitute for PAC, which yields much smaller test sets.
- It shows how behavioural adequacy assessment approaches can be used to assess test sets for systems that take data inputs and produce a data output (Section 3) in terms of a *Behavioural Coverage* criterion.
- It presents a *search-based test generation technique* that extends standard syntactic test generation techniques by ensuring that test sets are optimized with respect to this criterion (Section 4).
- It presents an *empirical study* on 18 Java units, indicating that the technique produces test sets that explore a broader range of program behaviour and find more faults than similar test sets that meet the traditional, syntax-based adequacy metrics (Section 5).

## 2. BACKGROUND

The section begins with a discussion of the weaknesses of conventional syntactic coverage measures. This is followed by an introduction to the general notion of behavioural adequacy. Finally, the section introduces two notions from the domain of machine learning. The first is the probably approximately correct (PAC) framework, a theoretical framework for evaluating model accuracy that underpins our BESTEST$_{PAC}$ approach. The second introduces the $k$-folds cross validation, a more applied evaluation framework that underpins our BESTEST$_{CV}$ approach.

### 2.1. Source code-driven testing is inadequate

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 usually straightforward to add new tests to improve branch coverage. However, the validity of these approaches is dubious because the precise relationship between a



DOI: 10.1002/stvr

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 a commensurate exploration of 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 1. 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 body mass index (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 fault, 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 characterizes 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.

### 2.2. Behavioural test set adequacy

*Behavioural adequacy* is founded on the idea that, if a test set is to be deemed adequate, its tests 'cover all aspects of the actual computation performed by the program' [2]. In this context, the term *behaviour* refers to the relationship between the possible inputs and outputs of a program. In other words, it should be possible to infer a model from the program behaviour from the test set, which can accurately predict the outputs for inputs that have not necessarily been encountered. The concrete representation of this will vary depending on the nature of the program; a sequential control-driven

```
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.



DOI: 10.1002/stvr

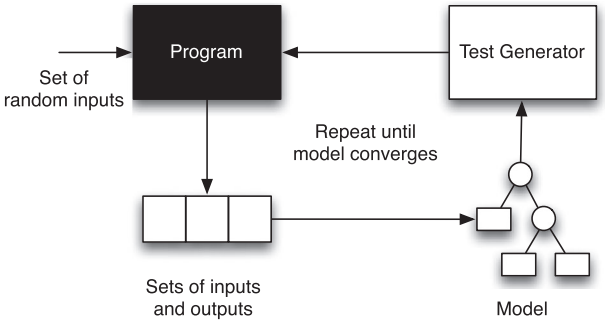



Figure 2. Basic 'virtuous loop' that combines testing with model inference.

system could be modelled as a finite-state machine; a data function might be represented by a differential equation, or a decision tree.

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 [6–14]. 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.

This loop is illustrated in Figure 2. Of course, there are variants; different types of models, feedback mechanisms and other sources of inputs can be fed in. Ultimately, the basic process is one of generating test sets, inferring models from their executions and (usually) using the models to identify new test cases. The point at which the process terminates is also variable. Some approaches will only terminate once a strict condition has been established (e.g. the model can be demonstrated to be exactly accurate [12]). Others will simply terminate if time runs out, or no further test cases can be found that contradict the model [14].

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 obtain a quantitative *measurement*. Models are deemed either accurate or inaccurate, accordingly test sets must be either adequate or inadequate. This is problematic because the tasks of inferring an exact model and testing a model against a system are practically infeasible. 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 (cf. previous work by Walkinshaw *et al.* [14]), 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.

### *2.3. The probably approximately correct (PAC) framework*

The aforementioned 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* [15]. In this context, Valiant's popular PAC framework [3] has been used extensively to reason in statistical terms about learnability—the notion that certain types of concepts can be inferred to a sufficient degree of accuracy within polynomial time. The PAC framework describes a basic learning setting, where the key factors that determine the success of a learning outcome are characterized 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.



DOI: 10.1002/stvr

Much of the notation used here to describe the key PAC concepts stems from Mitchell's introduction to PAC [15].

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 [15].

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 characterize 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).

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 model $h$. The error rate of $h$ subject to distribution $\mathcal{D}$ ($error_{\mathcal{D}}(h)$) can be established with respect to a further 'evaluation' sample from $EX(c, \mathcal{D})$. This represents the probability that $h$ will misclassify one of the test samples, that is, $error_{\mathcal{D}}(h) \equiv Pr_{x \in \mathcal{D}}[c(x) \neq h(x)]$.

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 the extent to which an inferred model is allowed to be erroneous to still be considered approximately accurate ($\epsilon$), and the probability with which it will infer an approximate model ($\delta$).

It is important to distinguish between the term 'correct' in this context of model inference and the context of software testing. A model is 'correct' in the PAC sense if, for every input, it will return the same output as the subject system from which it was inferred, regardless of whether these outputs conform to the output expected by the systems' developer. This is clearly different from the notion of functional correctness as applied to software. In this case, an output is only correct if it conforms to the developer's intentions, or some abstract specification.

### 2.4. *k-folds cross validation*

In the testing context considered in this paper, there are several practical barriers that undermine the validity of applying PAC. PAC presumes that there are two large samples, selected under identical circumstances. However, in a testing context, there may only be a small selection of test cases available, and partitioning this sample into a training and a test sample could produce two highly dissimilar sets of features. Secondly, the available sample of test cases is unlikely to have been randomly selected from some fixed distribution; they might have been handpicked to deliberately achieve branch coverage for example.

This problem is also well established in machine learning. A practical alternative that has become the de facto standard for evaluating inferred models in such a context is known as *k-folds cross validation* (CV) [16]. The basic idea is to, for some $k$, randomly divide the original set of examples into $k$ mutually exclusive subsets. Over $k$ iterations, a different subset of examples is selected as the 'evaluation set', whilst the rest are used to infer the model. The final accuracy value is taken to be the average accuracy score from all of the iterations. This has the benefit of producing an accuracy measure without requiring a second, external test set.

*2.4.1. Choosing k.* One key parameter with the use of CV is the choice of $k$. There is no firm advice on choosing this [16]. The choice depends on a combination of (a) the total number of examples available, (b) the extent to which these exercise the behaviour of the system in question and (c) the



DOI: 10.1002/stvr

amount of time available. For example, if the number of examples is low and $k$ is too high, the partition used for evaluation could be too small, yielding a misleading score. If there are lots of examples and $k$ is high, it could take too much time to iterate through all of the partitions.

A common choice for $k$ when CV is used to evaluate machine learning techniques is 10 (the CV technique as a whole is often referred to as '10-folds cross validation'). Alternatively, if the number of examples is not too high, it is possible to use 'leave-one-out cross validation', where $k = n - 1$, and the evaluation set always consists of just one example. Ultimately, however, given the lack of concrete guidance, the choice of $k$ is left to the user, and their judgement of the extent to which the set of examples is representative of the system in question.

*2.4.2. Choosing a scoring function.* Another important parameter is the choice of evaluation metric. In other words, given an inferred model and a sample of inputs and outputs that were not used for the inference, how can we use this sample to quantify the predictive accuracy of the model? To provide an answer, there are numerous approaches, the selection of which depends on several factors, including the type of the model and whether its output is numerical or categorical.

For models that produce a single numerical output, the challenge of comparing expected outputs to the outputs produced by a model is akin to the challenge of establishing a statistical correlation. Accordingly, standard correlation-coefficient computation techniques (Pearson, Kendall or Spearman rank) can be used.

For non-numerical outputs, assessing the accuracy of a model can be more challenging. Accuracy is often assessed by measures that build upon the notions of true and false positives and negatives. Popular measures include the F-measure (the harmonic mean of Precision and Recall [17]), the receiver operating characteristic (ROC) and Cohen's kappa measure on inter-rater agreement [18].

## 3. ASSESSING BEHAVIOURAL ADEQUACY

In this section, we show how the various notions presented in the previous section can be used to compute inference-based measures of test adequacy. We firstly present a PAC-based measure [4] in Section 3.1. This is then followed up by the $k$-folds CV measure in Section 3.2, which addresses some of the limitations of PAC.

### *3.1. 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', to reformulate syntax-based adequacy axioms [6, 7, 10] or to place bounds on the number of tests required to produce an adequate test set [19].

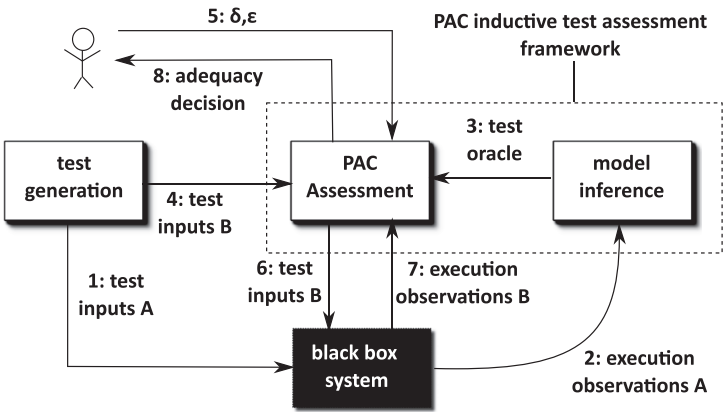



Figure 3. Probably approximately correct (PAC)-driven test adequacy assessment [19]



DOI: 10.1002/stvr

The basic approach (as presented in [4]) is shown in Figure 3 (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 system under test (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). This is executed on the SUT; the executions are recorded and supplied to the inference tool, which infers a hypothetical model that predicts outputs from inputs. 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, and the results are used 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).

*3.1.1. Limiting factors.* The PAC framework was developed as a purely theoretical concept. With respect to testing, this has facilitated several useful calculations, such as establishing a polynomial bound on the number of random tests required to ensure behavioural adequacy [19]. More fundamentally, it enables us to establish whether, at least in theory, certain systems are even 'testable' (i.e. whether or not there is a polynomial limit for a given system at all).

However, applying this framework in a testing context gives rise to several fundamental limitations (as mentioned in Section 2.4). The assumption that tests are selected at random from a fixed distribution is unrealistic. Effectively, this assumption would imply that there exists some fixed set of test inputs, from which both the test set and evaluation set are blindly selected. In reality, test sets are generated differently. For example, in attempting to ensure syntax coverage, one might select random inputs at first but then select further inputs that are influenced by the performance of the initial ones. One might also include particular test cases that are sanity-checks, and others that target aspects of behaviour that are known to be particularly vulnerable to faults. This is not random selection; the distribution is not fixed, and the tests are not selected independently.

If we simply ignore these problems and use PAC regardless, there is a danger that the adequacy score is invalid. Ultimately, PAC is a statistical framework; it makes a probabilistic assumption that the final model is 'approximately correct'. To be valid, statistical comparisons between groups rely on the presumption that the groups are sufficiently similar (to ensure that we are comparing 'like with like'). This assumption can be easily violated in a testing context.

Aside from the manner in which samples are selected, there is also the (implicit) assumption that there are sufficient test cases from which to form a sufficiently robust comparison. Given that the comparison between the test set and the evaluation set is statistical in nature, statistical observations can only be confirmed with any confidence if they are derived from a sufficiently large number of observations. This runs counter to the general aim of keeping test sets as small as possible. Test sets that contain fewer than 10 test cases are very frequent but cannot be reasonably used in a PAC setting, because any statistical conclusion would lack sufficient statistical power.

These limiting factors do not necessarily rule out the use of PAC; it is always possible to calculate a score. If there are not enough test cases to produce two sufficiently sized groups, the de facto alternative is to construct the evaluation set from a large number of random tests. Ultimately, however, such workarounds do place a question mark over the validity of the adequacy scores that are produced.



DOI: 10.1002/stvr

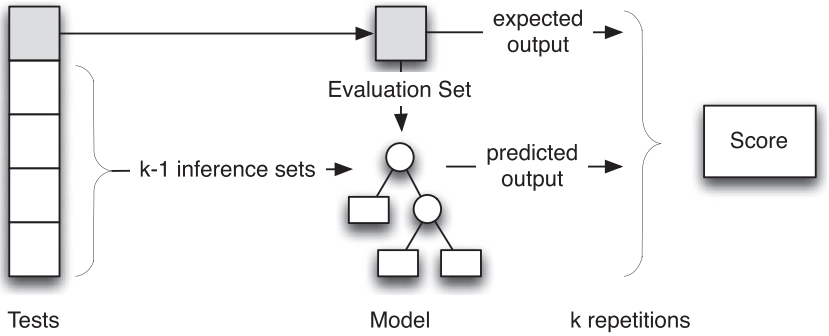



Figure 4. Illustration of $k$-folds cross validation applied to behavioural adequacy.

### 3.2. Using CV to quantify behavioural adequacy

This section shows how $k$-folds cross validation (introduced in Section 2.4) can be used instead of PAC. This enables the use of a single large test set instead of two separate ones, which attenuates the problem of bias that can arise with PAC and reduces the number of tests required.

The process is illustrated in Figure 4. The scoring process starts with a single test set. The test set is partitioned into $k$ sets of tests. Over $k$ iterations, $k-1$ of the sets are used to infer a model, whilst the remaining set is used for evaluation. The result is taken to be the average of the scores.

CV is commonly used under the assumption that the given set of examples collectively reflect the complete range of behaviour of the underlying system. If this is the case, the resulting average score can be taken as indicative of the accuracy of the model inferred from the complete set. This assumption of a collectively representative set of examples does of course not always hold. In the case of program executions, a truly representative set is notoriously difficult to collect [1]. This gives rise to the question of what a CV score means when the test set is not necessarily complete or representative.

In this scenario, a CV score has to be interpreted in a more nuanced manner. Although CV scores are always 'internally valid' (they are always valid with respect to the data they are given), they are not necessarily 'externally valid'; they can easily be misled by a poor sample. For example, looking forward to our inference of models from program executions, a set of examples that omits a prominent function in a program could yield models that all presume that no such function exists. Although the models may be very wrong, because they are all evaluated with respect to the same incomplete sample, they could still yield a very high CV score.

As a result, for scenarios where sets of examples fail to collectively expose the full range of behaviour of the system, there is the danger that the resulting score can be inaccurate. It is consequently necessary to interpret CV in a conservative light. If the score is high, it could well be due to a bias in the sample. A high CV score can at best corroborate the conclusion that an inference technique is accurate but cannot offer any form of guarantee. However, if the score is low, this is more reliably indicative of a problem, that is, with the sampling of the test set, or the inference of the model.

### 3.3. Combining code coverage with behavioural adequacy

As discussed in Section 2.1, source code coverage alone is insufficient when used alone as a basis for assessing test adequacy. Test sets that achieve code coverage often fail to expose crucial aspects of software behaviour. Capturing the set of executions that fully expose the relationship between the input to a program and its output generally entails more than simply executing every branch. It is this line of reasoning that underpins the PAC and CV-driven behavioural adequacy approaches.

However, as discussed in the previous text, PAC and CV are limited by one common factor: they will provide a misleading score if the test set(s) is incomplete. If a portion of a program that contributes to the output of the program is not executed, it cannot be factored into a behavioural model. In this respect, test adequacy is two-dimensional; it is necessary to make sure that all of



DOI: 10.1002/stvr

the code is executed, but it is equally necessary to ensure that this code is executed sufficiently rigorously so as to ensure that all of its possible contributions to the program output are exposed.

This rationale underpins the argument that code coverage and behavioural adequacy are complementary [2]. Code coverage can guide the test selection towards executing the full range of behavioural features but cannot ensure that these features are executed in a sufficiently comprehensive manner. This aspect can however be assessed in terms of behavioural adequacy. The fact that the two measures are complementary suggests that they should both be taken into account when assessing the adequacy of a test set. This gives rise to our behavioural coverage metric, defined as follows:

*Definition 1 (Behavioural coverage)*

$$BC_T = (Cov_T, BA_T)$$

For a given test set $T$, $Cov_T$ measures the code coverage (e.g. branch coverage) for $T$. $BA$ measures the behavioural adequacy (either by PAC or CV) for $T$.

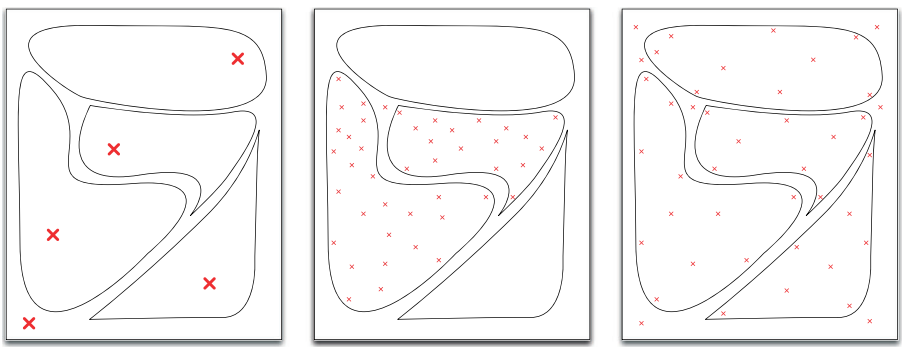
In order to impose an order on test sets in terms of their behavioural coverage, one could combine the two dimensions into a weighted average, similar to how the F-measure combines precision and recall. This is what we do in our optimisation (Section 4). Depending on the circumstances, however, it might be preferable to treat the two dimensions separately, for example, as part of a multi-objective optimisation.

This is not the first attempt to extend pure code coverage. However, past approaches to overcome the deficiencies of code coverage have lead to extended coverage metrics that consider code elements only covered if they influence observable outputs (e.g. OCCOM [20] or OMC/DC [21]), or if they propagate to test assertions [22]. Behavioural coverage is different in that syntactic coverage and behavioural adequacy cannot easily be coerced into a single value, they are rather two dimensions: Code coverage represents *which* parts of a program have been tested, and adequacy measures *how well* they have been tested.

### 3.4. An example of behavioural coverage in action

To provide an intuitive illustration of the rationale behind behavioural coverage, we consider the three diagrams shown in Figure 5. The area of each diagram is taken to represent the full range of input/output behaviour. Each zone within a diagram represents a distinctive feature of program behaviour.

Branch coverage of a program (Figure 5(a)) can often be achieved from a relatively small test set. In the BMI example, this ought to execute each behaviour of the program at least once. As one could easily conceive of a program involving conditional loops or complex data dependencies,

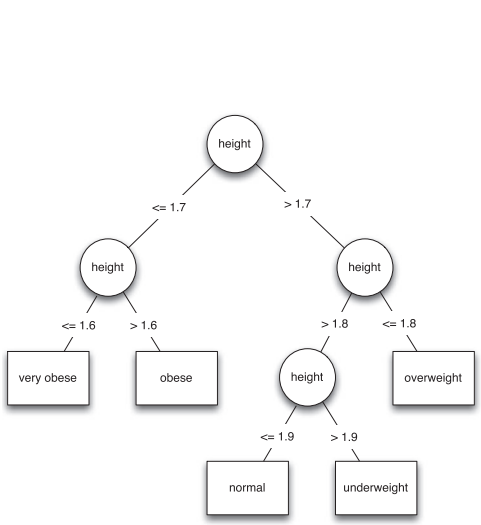


(a) Branch Coverage (b) Behavioural Adequacy (c) Behavioural Coverage

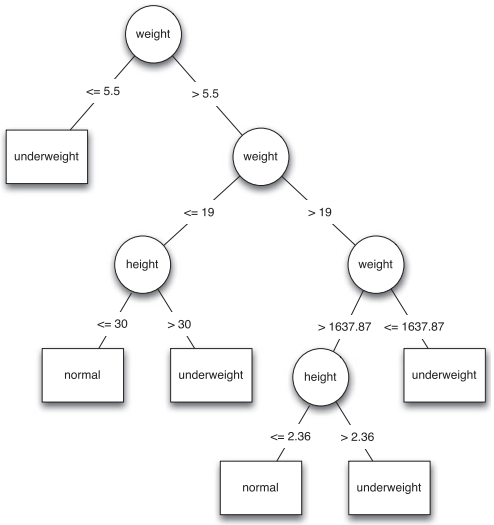
Figure 5. Illustration supporting the rationale for behavioural coverage.



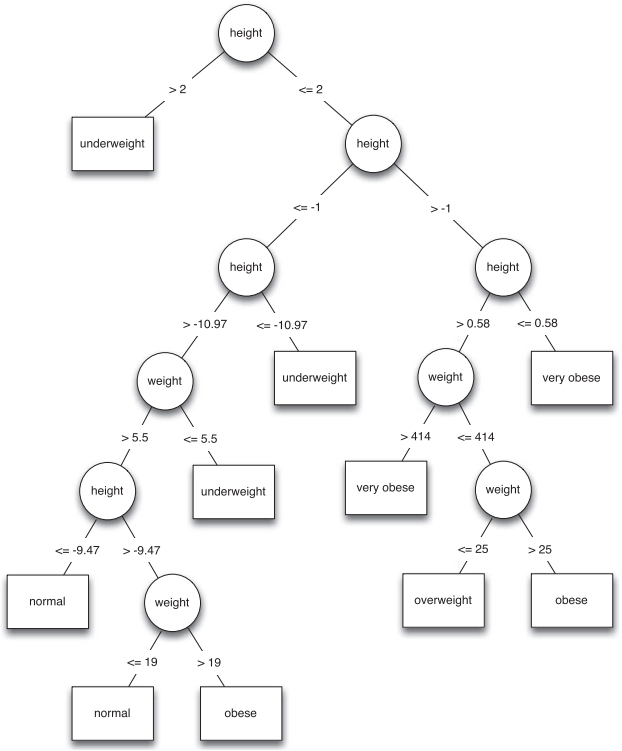
DOI: 10.1002/stvr




(a) Test set in Figure 1: $BC_T = (1.0,0.0)$




(b) Example test set of 76 tests: $BC_T = (0.4,0.97)$




(c) Example test set of 78 tests: $BC_T = (1.0,0.89)$

Figure 6. Decision trees inferred from different example test sets for the body mass index program.

it is therefore worth emphasizing that one could choose more rigorous code coverage criteria for behavioural coverage, such as Def-Use coverage [23] or MCDC [24]. However, merely executing a behaviour once is not necessarily sufficient by itself; as discussed in Section 2.1, a single execution of a feature is unlikely to expose any faulty behaviour.

Behavioural adequacy tells us how accurate the models are that we infer from the executed test cases. This, in turn, can be used to gauge how well our test cases delineate between the distinctive behaviour of different program features. For example, consider the decision tree in Figure 6(a), which is inferred from the branch coverage test set for the BMI example in Figure 1. This decision



DOI: 10.1002/stvr

tree is clearly incomplete; for example, none of the inferred decisions even depend on input 'weight'. This incompleteness is reflected with a behavioural adequacy measurement of 0.0.

However, as shown in Figure 5(b), behavioural adequacy alone can be misleading. If a feature is not represented in a test set at all, this will not be reflected in the resulting assessment. Figure 6(b) shows a decision tree inferred from a set of 76 tests (selected from a random set of tests for illustrative purposes) that cover only the cases of normal and underweight BMI values. Compared with Figure 6(a), the decisions are quite accurately reflecting the implementation with respect to the observed behaviour, which is reflected by a high adequacy score (0.97 using F-measure). However, the branch coverage value (40%) reveals that only two out of five cases are covered.

This leads us to the rationale for behavioural coverage, as shown in Figure 5(c). It seeks to combine the benefits of these two approaches. Accordingly, a test set should not only execute each individual feature but should do so sufficiently to achieve behavioural adequacy with respect to all of the features. An example of such a test set is used to infer the decision tree shown in Figure 6(c): All branches are covered, leading to 100% branch coverage. However, the adequacy score is now lower than that in Figure 6(b) (0.89), showing that although more behaviour has been covered, it has not been exercised with the same intensity as the subset of behaviour in Figure 6(b).

### *3.5. What about program correctness?*

In this paper, we are exclusively concerned with the adequacy problem; assessing the extent to which a test set covers the observable program behaviour. The question of whether the tests are producing the correct results (the *oracle problem*) remains an open one. Note that this is not a problem specific to the notion of behavioural adequacy but applies to any adequacy measurement not based on a representation of intended behaviour (e.g. specification). For now, we have to presume that faults will manifest themselves in obvious ways (e.g. as program failures) or will be detected with the aid of assertions that have been produced *a priori* (either within the code or expressed as test properties).

One further commonly suggested option is for the tester to manually inspect the test outputs. The problem here is that it can be a time-consuming, error-prone task. Potentially complex computations carried out by the SUT have to be replicated by the tester. Abstract functional requirements have to be reconciled with (possibly large numbers of) low-level inputs and outputs.

In this respect, inferring models from test executions can however provide assistance. Depending on the choice of inference algorithm, inferred models can provide an abstract, inspectable representation of what has been tested. What might be thousands of test executions can often be summarized into a much more succinct abstract model.

## 4. GENERATING ADEQUATE TEST SETS

Having defined how to measure test set adequacy, we now turn to the question of how to produce test sets that are optimized in this respect. Whilst some traditional test generation problems such as branch coverage can be nicely framed in a way that makes it possible to use symbolic techniques to calculate test data, this does not immediately hold for behavioural adequacy. Furthermore, because a test set that achieves *full* behavioural adequacy could imply an unreasonably large number of tests, one might be content to trade off the adequacy for a smaller, more practical test set. Ultimately, this is an optimization problem calling for search-based testing techniques.

### *4.1. Search-based testing*

The use of meta-heuristic search techniques to produce test cases is commonly referred to as search-based testing. Typically, meta-heuristic search algorithms are driven by fitness functions, which are heuristic functions to estimate the distance of a candidate solution to the optimal solution. In the context of software testing, an individual of the search is often a test case, and the fitness function estimates the distance towards reaching a particular point in the code. Alternatively, individuals can constitute entire test sets. This is necessary if the objective concerns the entire test set, such as achieving 100% branch coverage or, in our case, behavioural adequacy.



DOI: 10.1002/stvr

To develop a search-based test generator for behavioural adequacy, it is necessary to define a suitable fitness function. According to Section 3, and in particular Definition 1, achieving this goal involves two distinct objectives: first, to execute *all* the code of the method, and second, to *adequately* do so. In other words, the fitness function should assess a test set in terms of its code coverage, but also in terms of its ability to expose a sufficiently broad range of observable behaviour with respect to the executed code.

The tasks of fulfilling these two objectives are discussed in the following two subsections. This is followed by a more in-depth description of how the genetic algorithm applies the resulting fitness functions to home-in on suitable test sets.

### *4.2. Code coverage*

The first objective is a traditional goal in test generation; a prerequisite to finding errors in a piece of code is that the code is actually executed in the first place. A reasonable approximation is articulated in the traditional *branch* coverage metric: The proportion of possible true and false predicate outcomes that have been executed in a program (where 100% implies that all edges of the control flow graph are traversed). In principle, the approach could also be used in conjunction with more rigorous criteria such as Modified Condition/Decision Coverage (MCDC) [24], which may have a higher chance of revealing additional behaviour. Theoretical coverage criteria such as path coverage may help to expose more behaviour, but the path explosion problem caused by constructs such as by loops means that such criteria cannot generally be applied in practice. Furthermore, the link between behaviour and coverage remains tenuous: Executing loops with different numbers of iterations will not necessarily lead to relevant new behaviour.

We thus assume that a minimum requirement for any adequate test set is that all feasible (atomic) 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. The fitness of a test set with respect to covering all branches is based on the *branch distance* metric, which estimates how close a logical predicate in the program was to evaluating to true or to false [25]. The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see [25] for details). For example, for predicate $x \geq 10$ and $x$ having the value 5, the branch distance to the true branch is $max(0, 10 - x) = 5$.

A simple fitness function [26] is to sum up the individual branch distances, such that an optimal test suite would have a fitness value of 0. Some details need to be taken into account: First, branch distances need to be normalized to prevent any one branch from dominating the search. Second, we need to require that each branching predicate is executed at least twice, to avoid the search from oscillating between the true and false evaluation of the predicate (i.e. if the predicate is executed only once and evaluates to true, the search would optimize it towards evaluating to false instead, and once that is achieved, it would optimize it towards evaluating true again). Let $d_{min}(b, T)$ denote the minimum branch distance of branch $b$ when executing test set $T$, and let $\nu(x)$ be a normalizing function in $[0, 1]$ (e.g. we use the normalization function [27]: $\nu(x) = x/(x + 1)$), we defined $d(b, T)$ as follows:

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

When generating adequate test sets, we require that the branch coverage for all branches in the set of target branches $B$ is maximized, that is, the branch distance is minimized. Thus, the overall branch fitness is defined as follows:

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



DOI: 10.1002/stvr

### 4.3. Behavioural adequacy

The second objective expresses how thoroughly this code has to be exercised with respect to its externally observable behaviour.

*4.3.1. Using PAC and a validation set.* To optimize behavioural adequacy using the PAC approach (Section 3.1), we require two test sets. Consequently, our initial approach to compute behavioural adequacy [4] was to evolve pairs of test sets, where one will ultimately end up being the final test set, and the other one is the 'evaluation' set that is used to infer a model with WEKA [28], which is a widely used collection of machine learning algorithms. For a given pair of test sets $(T_1, T_2)$, the adequacy $A$ can be measured by inferring a model from $T_1$, and comparing the predicted outputs from this model against the actual outputs from the program for the tests in set $T_2$.

Let $A(T_1, T_2, S)$ be a function that calculates the adequacy as described in Section 3.1 using a scoring function $S$ (Section 2.4.2). 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 of the test set in the fitness function, to spur the growth of test sets until adequacy can be reliably determined. We do this by including the size of test set $T_1$ in the fitness function if adequacy is 0, such that larger test sets will be favoured by the search. To ensure that any adequacy value $> 0$ has a higher fitness value, we normalize the size of $T_1$ in the range $[0, 1]$ using the normalization function $\nu$ [27] (see previous text), and otherwise add 1 if adequacy is $> 0$:

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

A combined fitness function to guide test generation for adequacy can thus be defined as the weighted sum of $\mathcal{A}_{PAC}$ and the branch coverage values $\text{cov}(T_1)$ and $\text{cov}(T_2)$ (coverage of both test sets needs to be optimized):

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

The values $\alpha$ and $\beta$ are used to weigh coverage against adequacy. For example, in our experiments, we used $\beta = 1000$ and $\alpha = 1$, which would cause the search to prefer even small improvements in coverage to large increases in behavioural adequacy. The reasoning for this choice of weighting is that a higher adequacy value may be misleading if it is measured on an incomplete sample of behaviour. The search will thus initially be dominated by code coverage, and once the search converges on code coverage, exploration will focus more on behaviour. However, covering the existing behaviour to a higher level of adequacy might in turn lead to coverage of additional syntactic elements, which the search would embrace and then explore to higher adequacy. Note that this weighting is an implementation choice we made for our experiments; in principle, it would also be possible to use multi-objective optimization to generate the test suites [29]. In this case, the result of the optimisation would be a Pareto-front of test sets, and the user would be able to choose between test sets with higher coverage and lower adequacy, or test sets with lower coverage but higher adequacy of the covered code.

As discussed in Section 3.1, there is a threat of sampling bias as $T_1$ and $T_2$ are not drawn independently from each other. The consequences of this are twofold. Firstly, this interdependence means that the final test sets are not as effective as they could be. Secondly, it can undermine the reliability of the final 'behavioural adequacy' score. A test set that is accompanied by an artificially high adequacy score that cannot be trusted by the developer undermines its value.

*4.3.2. Using $k$-folds cross validation.* To overcome this drawback of the initial approach, we introduced the $k$-folds cross validation approach described in Section 3. The adequacy of a test set $T$ is calculated using CV by deciding on a $k$, partitioning $T$ into $k$ subsets. In turn, each of the $k$ subsets is set aside for evaluation, whilst the others are used for inferring the model. The adequacy measurement is the average of the individual scores.



DOI: 10.1002/stvr

Let $A(T, S, k)$ be the adequacy calculated as described in Section 3.2, where $S$ is a scoring function (Section 2.4.2) and $k$ is chosen as described in Section 2.4.1. Again, an adequacy value of 0 may indicate a too small sample size, such that the fitness function favours growth in that case:

$$\mathcal{A}_{CV}(T) = \begin{cases} 1 + A(T, S, k) & \text{if } A(T, S, k) > 0, \\ \nu(|T|) & \text{otherwise} \end{cases}$$

Expressed as a weighted combination with the branch coverage value $\text{cov}(T)$, the fitness function to guide test generation for adequacy is thus defined as follows:

$$\text{fitness}(T) = \alpha \cdot \mathcal{A}_{CV}(T) + \beta \cdot \text{cov}(T)$$

### *4.4. Evolving adequate test sets with a genetic algorithm*

The optimization goal is to produce an adequate test set. 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 an adequate test set is usually not known beforehand, so we assume that the size is variable and needs to be searched for but has an upper bound $B_T$. The neighbourhood of test sets is potentially very large, such that we aim for global search algorithms such as a *genetic algorithm* (GA), where individuals of the search population are referred to as *chromosomes*.

We generate the initial population of test sets randomly, where a random test set is generated by selecting a random number $n = [T_{min}, T_{max}]$, and then generating $n$ test cases randomly. A test case is generated randomly by assigning random values to the parameters of the method under test. Note that this is not a requirement; the initial population could also be based on an existing test set (e.g. [30]), for which one may intend to improve behavioural adequacy.

Out of this population, the GA selects individuals for reproduction, where individuals with better fitness values have a higher probability of being selected. To avoid undesired growth of the population (bloat [31]), individuals with identical fitness are ranked by size, such that smaller test sets are more likely to be selected. Crossover and mutation are applied with given probabilities, where crossover exchanges individual tests between two parent test sets, and mutation changes existing tests or adds new tests to a test set. These search operators have been explored in detail in the literature, and we refer to [32] for further details.

When using the $\mathcal{A}_{PAC}$ fitness function, the use of a validation set requires some adjustment: Here, a chromosome is a pair of test sets $\langle T_1, T_2 \rangle$, in contrast to the single test set $T$ used for the approach based on $\mathcal{A}_{CV}$. In general, it is not desirable that genetic material is exchanged between the first (test set) and the second (validation set). Therefore, in this case, crossover and mutation are applied to both test sets individually.

If the search is terminated before a minimum has been found, post-processing can be applied to reduce the test set size further. Traditional test set minimization uses heuristics that select subsets based on coverage information; in the case of adequacy, it is not easy to determine how an individual test contributes to the overall behavioural exploration. Therefore, in our experiments, we minimized test sets using a simple, but potentially inefficient, approach where we attempt to delete each test in a set and check whether this has an impact on the fitness.

## 5. EVALUATION

To evaluate the effects and implications of behavioural coverage, we have implemented a prototype providing the functionality of the BESTEST$_{PAC}$ and BESTEST$_{CV}$ approaches. The prototype takes as input a Java class and attempts to produce a behaviourally adequate test set for each of its methods (according to either of the PAC or $k$-folds CV heuristics). Currently, systems are limited to parameters of primitive input and return values, although this will be extended to general data structures and stateful types in future work.

In this paper, we proposed BESTEST$_{CV}$ as a means to overcome limitations of BESTEST$_{PAC}$; therefore, the first three research questions focus on gathering a deeper understanding of the factors



DOI: 10.1002/stvr

Table I. Study subjects.

| Name | Source | No. of lines | No. of branches | Output |
|---|---|---|---|---|
| Bessj | [33] | 80 | 29 | Numeric |
| Binomial | [34] | 92 | 69 | Numeric |
| BMICalculator | [19] | 17 | 9 | Discrete |
| CalDate | [35] | 25 | 7 | Numeric |
| ColorHelper | [36] | 19 | 7 | Numeric |
| Evaluation | [28] | 33 | 3 | Numeric |
| Expint | [33] | 51 | 31 | Numeric |
| Fisher | [37] | 49 | 17 | Numeric |
| Gammq | [33] | 71 | 27 | Numeric |
| Luhn | [38] | 42 | 57 | Discrete |
| Middle | [35] | 19 | 29 | Numeric |
| MulAndCheck | [34] | 31 | 17 | Numeric |
| Remainder | [39] | 33 | 25 | Numeric |
| TicTacToe | [40] | 69 | 45 | Numeric |
| Triangle | [25] | 25 | 17 | Discrete |
| TCAS | [41] | 99 | 78 | Discrete |
| WBS | [42] | 168 | 93 | Discrete |
| WrapRoundCounter | [35] | 9 | 3 | Numeric |

influencing BESTEST$_{CV}$, whereas the fourth research question compares both approaches against a broad range of typical testing criteria (thus subsuming the original experiments in [4]). The specific research questions are as follows:

**RQ1:** What are the effects of learners, the value $k$ and evaluation metrics on the final behavioural adequacy score?
**RQ2:** Which configuration leads to test sets with the highest fault-detection ability?
**RQ3:** What is the relationship between behavioural adequacy and fault-detection ability?
**RQ4:** How does behavioural adequacy compare with traditional syntactical adequacy criteria?

### *5.1. Experimental setup*

*5.1.1. Subject systems.* The set of classes selected for our experiments is shown in Table I. These are selected (indiscriminately) from existing testing literature (referenced where possible). Because the purpose is to focus on particular units of functionality, we identify the particular method within each system that accepts the input parameters and returns the output value. Although an arbitrary number of other methods may be involved in the computation, our current proof-of-concept system requires the specification of a single point for providing input and reading output. As will be discussed in Section 6, our ongoing work is transferring this technique to other systems (e.g. abstract data types) that have multiple methods for providing input and output.

Our use of off-the-shelf machine learning algorithms imposes another restriction. We only selected systems that return primitive inputs and outputs (numbers, strings and Booleans). As discussed in Section 2.4.2, these have been grouped into two categories: those that return a numerical or a discrete value. This has a bearing on the model inference and scoring techniques that can be applied.

*5.1.2. Experimental variables.* The experimental variables that were identified, along with the values that were assigned to them, are presented below.

- The SUT (Table I).
- The machine learning algorithm. To establish the effect of this, we identified a broad range of algorithms that have been well established in machine learning literature and have been shown to excel for a wide range of different types of system. The selected algorithms are as follows:
  - The C4.5 Decision Tree learner (discrete systems) [43]
  - A Naive Bayesian Network learner (discrete systems) [28]



DOI: 10.1002/stvr

  - The M5 learner and the M5Rules variant (numeric systems) [28]
  - The AdaBoost learner (discrete systems) [44]
  - Multilayer Perceptron (neural net) learners (both numeric and discrete systems) [28]
  - Additive Regression (numeric systems) [28]

- The parameters of the BESTEST$_{CV}$ approach mentioned in Section 3:
  - The value $k$. This was chosen from one of the four most commonly used values: 2, 5, 10 or $n-1$ (leave one out (LOOCV)).
  - The evaluation function (to measure how accurate the inferred model is). We selected from F-Measure, kappa and area under the ROC curve and the correlation coefficient (all as computed by WEKA [28]).

*5.1.3. Data collection.* The described approaches have been implemented using EVOSUITE [32] as framework for evolutionary search and the WEKA model inference framework [28]. Each learner was executed with its default WEKA parameter settings.

The experiments were systematically executed, using every possible combination of variables on every subject. Because the EVOSUITE evolutionary algorithms and some of the WEKA inference algorithms include a degree of stochasticity, there is a danger that this can lead to particularly lucky or unlucky results. To avoid any skew from this effect, every experiment was repeated 30 times with different random seeds, and results are statistically analysed. EVOSUITE was configured to run for 10 min per run for each class and configuration; all other parameters were set to default values [45].

After the 10-min run, resulting test sets were minimized using a simple heuristic. For each test $t$ in the resulting test set $T$, the fitness value was calculated for the test set without the test ($T' = T \setminus \{t\}$). If the fitness value of $T'$ is worse than that of $T$, the test is retained in $T$; otherwise, it is removed from $T$ (and the adequacy calculation for the next test $t'$ would then be based on $T'$, rather than $T$).

For each execution of a test set, its behavioural adequacy and mutation score were recorded. The data collection was particularly time-consuming. For RQ1–RQ3, BESTEST$_{CV}$ had to be executed and assessed for every possible combination of $k$, evaluation function and learner. For RQ4, both BESTEST$_{CV}$ and BESTEST$_{PAC}$ had to be executed and compared against a host of baseline test generation techniques (carried out by configuring alternative fitness functions in EVOSUITE). Again, to avoid accidental bias, each experiment for the baseline approaches was repeated 30 times with different random seeds. For each subject, configuration and seed, EVOSUITE was run for 10 min. This should lead to a fair comparison, as the overhead of inferring models for BESTEST$_{PAC}$ and BESTEST$_{CV}$ is included in these 10 min. The following baseline test adequacy measures were used to drive test generation:

- A test set optimized for branch coverage, where test sets are built to cover every logic branch in every method.
- A test set optimized for branch coverage but expanded with random tests to match the average size of the BESTEST$_{CV}$ test sets (to investigate the influence of test set size).
- A random test set that was generated to match the average size of the BESTEST$_{CV}$ tests.
- A test set optimizing the weak mutation score [46], where test sets were generated to expose mutants.
- A test set optimizing dataflow coverage, where test sets are optimized to cover as many as possible inter-method, intra-method, and intra-class definition-use pairs in the target class [47].
- A test set generated using the original BESTEST$_{PAC}$ approach [4] (denoted 'PAC' in the plots). This was run with the full set of machine learning algorithms and evaluation metrics.

This culminated in a total of 286 202 experimental configurations, totalling 286 202 $\times$ 10 min $=$ 5.4 years of computational time. These experiments were executed on the University of Sheffield Iceberg HPC cluster[‡]. The full dataset is available online[§].

[‡]http://www.shef.ac.uk/wrgrid/iceberg.
[§]http://www.evosuite.org/bestest/.



DOI: 10.1002/stvr

*5.1.4. Analysis techniques.* To measure the effect of different factors (e.g. the choice of learner), or combinations of effects (required for RQ1–RQ3), we carry out grouped statistical tests. Because Shapiro–Wilks tests indicate that the data is non-normal, we cannot use analyses of variance and related measures such as partial eta-squared to assess effect sizes. Instead, we resort to non-parametric measures that do not presume normality.

To measure the effect of different factors (e.g. the choice of learner) on the adequacy and mutation scores, we use Cliff's $\delta$ [48]. This was primarily chosen because it is (a) well established and (b) simpler to interpret than other comparable measures. For more details about non-parametric effect size measurements, we refer to Peng and Chen [49]. Given a pair of groups $A$ and $B$ (e.g. the group of mutation scores for two different classes), Cliff's $\delta$ gives the probability that individual scores for $A$ are greater than those for $B$. The $\delta$ score lies in the interval $[-1 : 1]$, where negative numbers indicate the probability that all of the scores for $A$ are smaller than $B$, and positive numbers indicate the probability that all of the scores for $A$ are greater than $B$. A score of 0 indicates that the two distributions are overlapping.

Cliff's $\delta$ is a pairwise test. To investigate the effect of a particular factor (e.g. learner), we carry out every possible pairwise test for that factor (we compare the mutation scores for every pair of learners). The systematic approach used to run the experiments (Section 4) means that other factors are evenly represented in these groups. Because we only care about the relative distance between factors (and not which one was greater), we take the mean absolute value for all deltas, which gives us an overall effect size for the factor as a whole.

### 5.2. Results

*5.2.1. RQ1—What is the effect of learners, k and evaluation metrics on the final behavioural adequacy score.* The aim of this research question is to investigate how the various factors affect the resulting adequacy score. Table II shows the Cliff's $\delta$ values, illustrating the extent to which different factors contribute to the variance of the adequacy scores. To further provide an overview of the data, two box plots that summarize the interplay between the choice of learner and subject class are shown in Figure 7, and Figure 8 illustrates the effect of the evaluation metric for each of the discrete systems.

**Learner:** Taking the choice of class into account, the choice of a *suitable* learner is important, with an average $|\delta|$ of 0.12 for both numerical and discrete systems. This is corroborated by the box plots in Figure 7; different learners can yield completely different adequacy scores. For example, under TCAS for discrete systems, the MultiLayerPerceptron neural net learner tends to produce low adequacy scores (mean of 0.51), whereas the NaiveBayes learner measures a mean score of 0.93.

In general, we observe that for discrete systems, the J48 (C4.5) and Naive Bayes learners consistently lead to higher adequacy scores than the AdaBoost and MultiLayerPerceptron learners. For numerical systems, depending on the system, either the AdditiveRegression or M5 algorithms lead to highest adequacy scores.

**System under test (class):** The Cliff's $\delta$ values show that adequacy scores depend primarily on the system under test (the class). This is especially pronounced with the discrete systems, where the choice of class leads to an average $|\delta|$ of 0.58. This is to be expected; systems can vary substantially in terms of complexity, making it much harder to infer suitable models and derive useful test sets

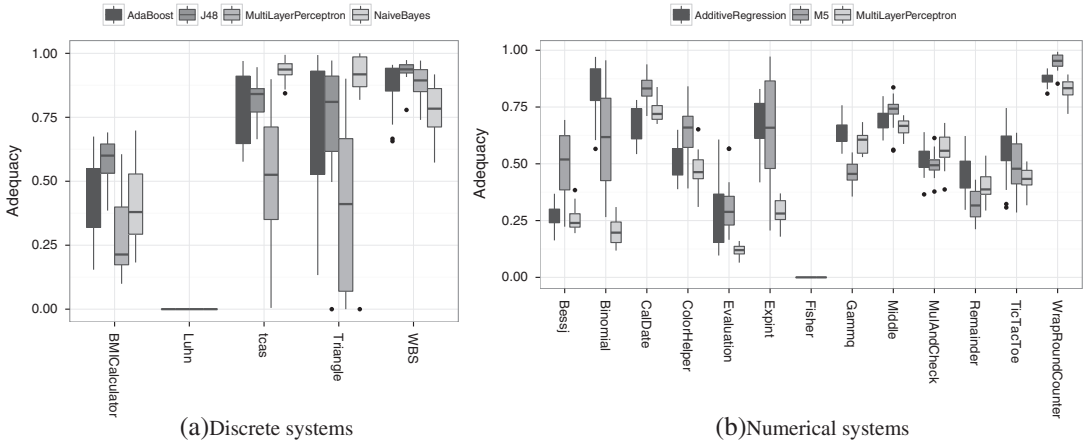
Table II. Cliff's delta with respect to adequacy scores.

| | **Numerical systems** | | **Discrete systems** | |
|---|---|---|---|---|
| **Factors** | Mean $\|\delta\|$ | SD | Mean $\|\delta\|$ | SD |
| SUT | 0.406 | 0.315 | 0.583 | 0.318 |
| Evaluation function | 0.006 | 0.004 | 0.125 | 0.077 |
| K | 0.1 | 0.05 | 0.025 | 0.016 |
| Learner | 0.121 | 0.081 | 0.125 | 0.085 |

SD, standard deviation.



DOI: 10.1002/stvr




(a)Discrete systems (b)Numerical systems

Figure 7. Adequacy scores by class and learner.

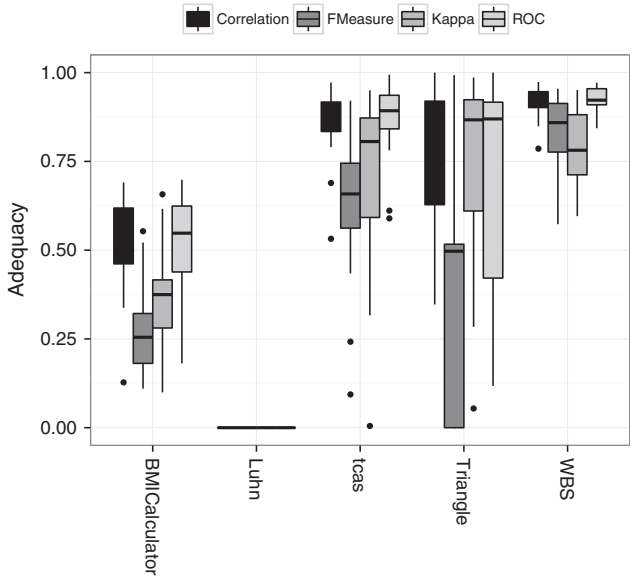



Figure 8. Adequacy scores for discrete systems by class and evaluation metric.

for some than others. This is also nicely demonstrated by the large variation between the individual classes as shown in Figure 7.

This gives rise to the question of which features of a system make it particularly easy or hard to infer an accurate model from the system. As we saw in the previous text, the learner does have an influence, but clearly less than the actual system under test. We conjecture that the main factor influencing this phenomenon is the *testability* of the system. In particular, Figure 7 shows two special cases: For the discrete systems, *no* configuration managed to achieve a positive adequacy score on the Luhn system. Similarly, for the numerical systems, *no* configuration managed to achieve a positive adequacy score for the Fisher class. We revisit Luhn and Fisher again in detail in Section 5.3.2.

**Evaluation metric:** The choice of evaluation metric has a negligible impact on the adequacy score achieved for numerical systems. Evaluation metrics have a more significant impact for discrete systems, as illustrated in Figure 8. The choice of evaluation metric for discrete systems leads to an average $|\delta|$ of 0.12. The box plots indicate that the F-measure tends to yield the lowest adequacy scores, whereas ROC and Correlation metrics tend to yield the highest.



DOI: 10.1002/stvr

**Value of $k$:** The choice of $k$ has a very slight impact for both numerical and discrete systems. The average $|\delta|$ is 0.01 for numerical systems, and 0.03 for discrete systems.

For numerical systems, higher values of $k$ tend to lead to higher values of adequacy. When all numerical configurations are ranked according to adequacy score, the top eight configurations have $k$ as LOOCV (the highest possible value). Of the bottom 10 configurations, six have $k = 2$, whereas none of the top 25 configurations have $k = 2$. The full table is provided in the accompanying dataset for this paper[§].

Finally, there is a remark about the interpretation of these scores. The purpose of this research question is to investigate the relationship between the various factors (learner, $k$ and evaluation metric) and the adequacy score. There is a temptation to conclude that those configurations that yield the highest scores are also the 'best' ones. This is however potentially misleading; a false high score could easily arise from the use of unsuitable configurations (Section 3.3). The next research question will therefore attempt to draw a link between adequacy and the actual performance: the ability to detect faults.

*5.2.2. RQ2—Which configurations lead to test sets with highest fault-detection ability.* To assess fault-detection ability, we resort to mutation analysis as a proxy measurement [50]. We calculated mutation scores using EvoSuite's built-in support for mutation analysis, which uses the same mutation operators as implemented in Javalanche [51] and the aforementioned study [50]. The mutation scores are averaged over 30 runs, to account for randomness in the evolutionary algorithm. The average $|\delta|$ results, shown in Table III, assess the extent to which different factors affect the mutation scores. The top 10 configurations according to average rankings are displayed in Table IV.

Before discussing the individual findings, it is important to discuss an intrinsic characteristic of the mutation data. As shown by the data in Table III, the choice of class has by far the largest effect on the eventual mutation score. In both numerical and discrete systems, the choice of system

Table III. Cliff's delta with respect to mutation scores.

| | **Numerical systems** | | **Discrete systems** | |
|---|---|---|---|---|
| Factors | Mean $\|\delta\|$ | SD | Mean $\|\delta\|$ | SD |
| SUT | 0.849 | 0.277 | 0.86 | 0.291 |
| Evaluation function | 0.003 | 0.002 | 0.04 | 0.03 |
| K | 0.01 | 0.006 | 0.03 | 0.015 |
| Learner | 0.03 | 0.02 | 0.08 | 0.07 |

SD, standard deviation; SUT, system under test.

Table IV. Average rankings in terms of mutation score for different configurations of learner, $k$ and evaluation function.

| Discrete systems | | | | Numerical systems | | | |
|---|---|---|---|---|---|---|---|
| Eval | $k$ | Learner | Av. rank | Eval | $k$ | Learner | Av. rank |
| Kappa | 5 | Bayes | 5.7 | ROC | 2 | Regression | 15.5 |
| Kappa | 2 | J48 | 13.2 | Kappa | 2 | Perceptron | 16.4 |
| Kappa | LOOCV | J48 | 13.5 | FMeasure | 2 | Perceptron | 18.1 |
| ROC | 2 | Bayes | 14.5 | FMeasure | 5 | Regression | 18.3 |
| FMeasure | 2 | J48 | 17.0 | ROC | 2 | Perceptron | 18.5 |
| Kappa | 10 | J48 | 19.7 | Kappa | 5 | Perceptron | 19.3 |
| Corr. | 2 | J48 | 20.0 | ROC | 5 | Regression | 20.0 |
| Kappa | 2 | Bayes | 20.2 | Corr. | 10 | Perceptron | 20.0 |
| FMeasure | LOOCV | Bayes | 20.8 | ROC | LOOCV | Perceptron | 21.0 |
| ROC | 10 | J48 | 21.9 | Kappa | 2 | M5 | 21.2 |

Due to the different numbers of learners for discrete and numeric systems, there are a total of 64 configurations per class for discrete systems and 48 possible configurations per class for numerical ones.



DOI: 10.1002/stvr

accounts for the biggest differences in score. This is explained by the fact that the average mutation scores were primarily stratified according to classes, where specific configuration options would only lead to relatively small deviations from the average mutation score for a given class. This is due to the fact that for a given class, the majority of mutants could be exposed trivially, by *any* test. Only a small fraction of remaining mutants could serve to distinguish the truly rigorous test cases; a relatively small improvement in the mutation score could indicate a significant increase in the adequacy of the test set.

Despite the overwhelming variations in mutation score according to class, the results show that the other factors nonetheless have a significant impact of their own. The key findings are as follows:

- For numerical systems, the choice of a suitable learner is the only factor to have a slight influence on the mutation score, with an average $|\delta|$ of 0.03.
- For discrete systems, all other factors have a non-trivial effect on the mutation score. Eval and $k$ have average $|\delta|$s of 0.03 and 0.04, respectively. The choice of learner has a comparatively much more significant effect, with an average $|\delta|$ of 0.08.

Looking at the specific configurations (the top 10 of which are shown in Table IV), the significant role of the learner is clear. The Naive Bayes and J48 (C4.5) learners consistently lead to the highest mutation scores for discrete systems (they dominate the highest-scoring 10 configurations but do not appear at all in the bottom 10). For numerical systems, it is a similar case for Additive Regression and MultiLayer Perceptron learners.

*5.2.3. RQ3—What is the relationship between the adequacy score and the ability to detect defects.* Whilst investigating RQ1 and RQ2, it became apparent that the range of adequacy and mutation scores is sensitive to the configuration of $\text{BESTEST}_{CV}$. An unsuitable configuration will overestimate adequacy scores. Accordingly, to answer this question, we assume that we are starting from a configuration that is capable of effectively exposing faults. So, given such a configuration, why is it effective? How and to what extent does adequacy contribute?

To establish this relationship, we therefore choose those configurations that led to the highest mutation scores in RQ2, that is, those ranked highest for numerical and discrete systems in Table IV. Note that the earlier criticism on source code-based criteria holds also for mutation analysis (Section 2.1). That is, a high mutation score is merely a rough indicator of adequacy. Nevertheless, a sufficient exploration of a program's behaviour should generally lead to high mutation scores. The chosen configurations are thus those that performed best across all numerical and discrete systems (in some cases, certain configurations might achieve higher mutation scores on individual classes).

First of all, we want to establish whether there is a correlation between adequacy and mutation score; does an increase in adequacy imply a corresponding increase in mutation score? Calculating the Pearson correlation coefficient indicates a coefficient of $-0.18$ for discrete systems and 0.02 for numerical systems. Clearly then, at face value, there is no positive correlation at all.

For a closer inspection, Figure 9 shows scatter plots relating the mean adequacy score to the mean mutation scores. It is important to bear in mind that each coordinate summarizes 30 actual recordings, which are often unevenly spread. Because there are too many points to plot in a useful way, the concentration of these points is indicated by the contour lines (the results of a 2D Kernel density estimation[¶]).

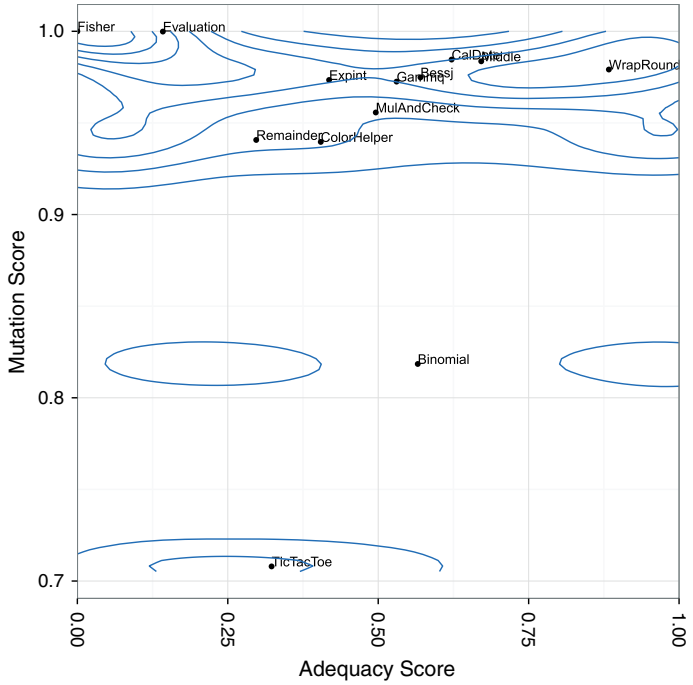
This points towards a more subtle relationship between the two factors that differ between the discrete and numerical systems. In the numerical systems, the mutation scores tend to be high, but there is a high variance in adequacy score (as indicated by the fact that the contours run across the plot in a relatively narrow band). However, for the discrete systems, both dimensions tend to be spread more evenly around particular coordinates for each system (as indicated by the more conical contours).

For Fisher, Evaluation (numerical) and BMICalculator (discrete), the results stand out; the mutation score is very high, despite a very low adequacy score. These programs are interesting because

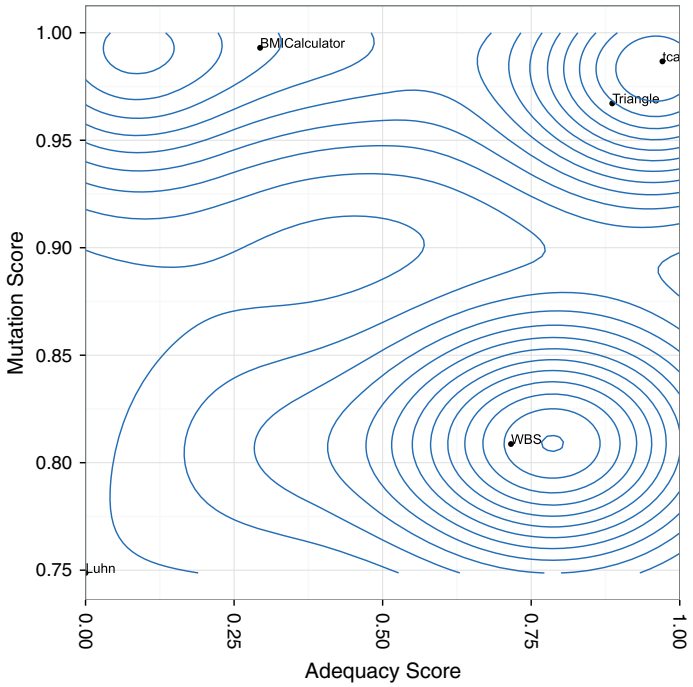
[¶]http://docs.ggplot2.org/0.9.3.1/geom_density2d.html.



DOI: 10.1002/stvr




(a) Numerical systems: Eval=ROC, $k$=2, learner = Additive Regression




(b) Discrete systems: Eval=Kappa, $k$=5, learner = Naive Bayes

Figure 9. Scatter plots relating adequacy scores to the normalized mutation scores for the two top configurations in Table IV.

their source code is relatively easy to cover by standard branch coverage techniques, making it relatively straightforward to expose the mutants. However, their underlying functional behaviour is exceedingly difficult to fully capture and infer. The difficulty of doing so for Fisher was discussed



DOI: 10.1002/stvr

for RQ1. Evaluation calculates a statistical correlation, where a test set has to contain pairs of number sets as inputs that fulfil particular correlation properties. BMICalculator carries out a non-linear calculation from two numbers to compute a category; although its branches can be covered easily, this would not suffice to expose the underlying non-linear relationship between the input parameters and the output category.

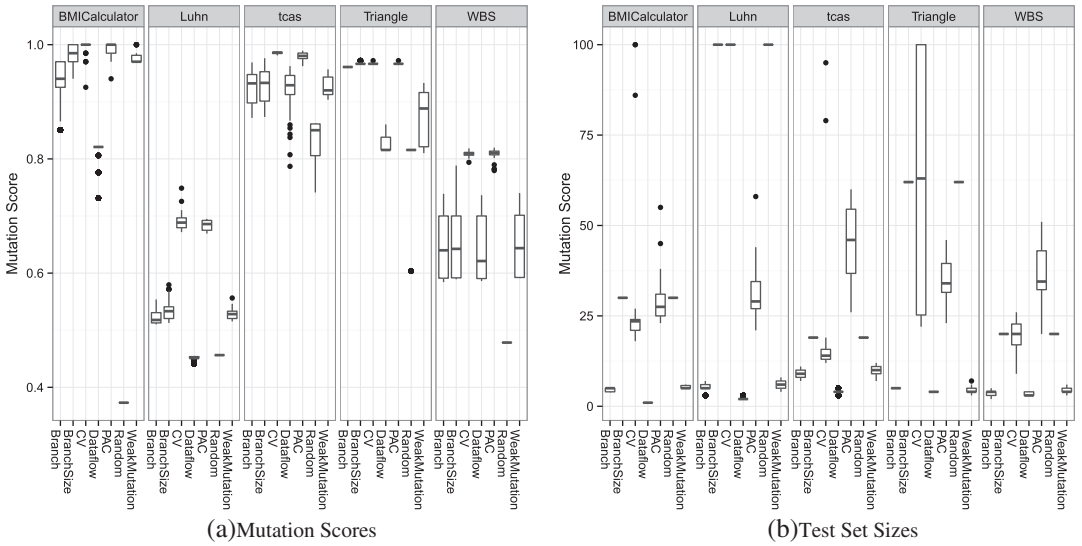
If we restrict ourselves to programs for which the mutants cannot be trivially exposed (by leaving out Fisher, Evaluation and BMICalculator), effects on the correlation coefficients are observable. For discrete systems, the Pearson correlation rises from −0.18 to 0.59, and for numerical systems, the correlation rises from 0 to 0.21. In summary, for programs where mutants are not trivially covered, a correlation emerges between adequacy and the mutation score. An increase in adequacy indicates an increase in the number of faults that are exposed by the test set.

Branch coverage remains the primary driver for mutation coverage. For numerical systems, the correlation between branch and mutation coverage is 0.59 (0.7 for the subset excluding Fisher, Evaluation and BMICalculator), and for discrete systems, it is 0.93.

Despite the fact that they are both positively correlated with mutation score, adequacy and branch coverage are not correlated with each other (Pearson correlations of −0.02 for both numerical and discrete systems). This corroborates their complementary nature, as discussed in Section 3.3. The branch coverage objective increases the diversity of the test set to execute a larger proportion of program features. The extent to which these features are fully explored is in turn maximized by the adequacy score, which also tends to spur the exploration of additional branches that are particularly hard to reach. Ultimately, it is this synergy between the two that yields the higher mutation scores.

*5.2.4. RQ4—How does* BESTEST$_{CV}$ *compare with existing adequacy criteria.* To establish the comparative performance of test sets generated using the BESTEST$_{CV}$ approach, we compared them against test sets generated to fulfil a selection of traditional, syntax-based criteria, as described in Section 5.1.3. All of the resulting test sets were compared in terms of their mutation score and size. The results for discrete systems are shown in Figure 10, and the results for numerical systems are shown in Figure 11.

To make such a comparison meaningful, it is necessary to pick a specific configuration for BESTEST$_{CV}$ . For this, it makes sense to select a configuration that is known to perform reasonably well (previous RQs have after all shown that there are several choices of model inference algorithm that perform very poorly). Accordingly, we selected the two configurations that had the highest correlations for discrete and numerical systems (as used in RQ3, those ranked highest for numerical

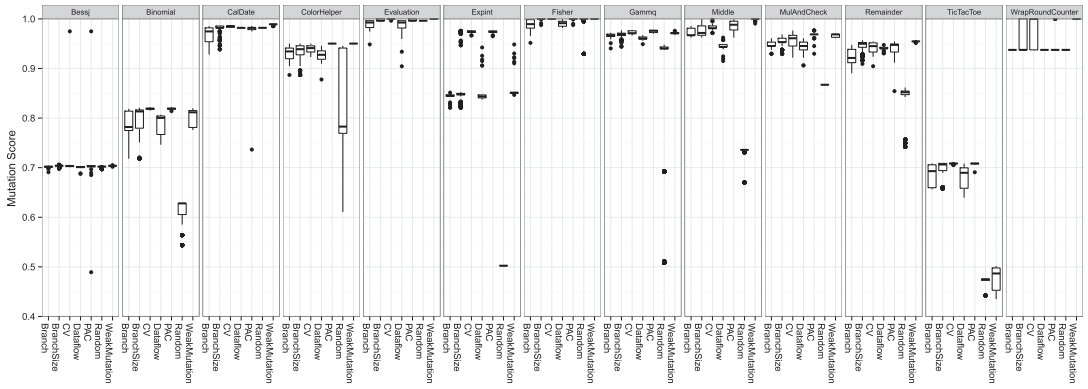



(a)Mutation Scores (b)Test Set Sizes

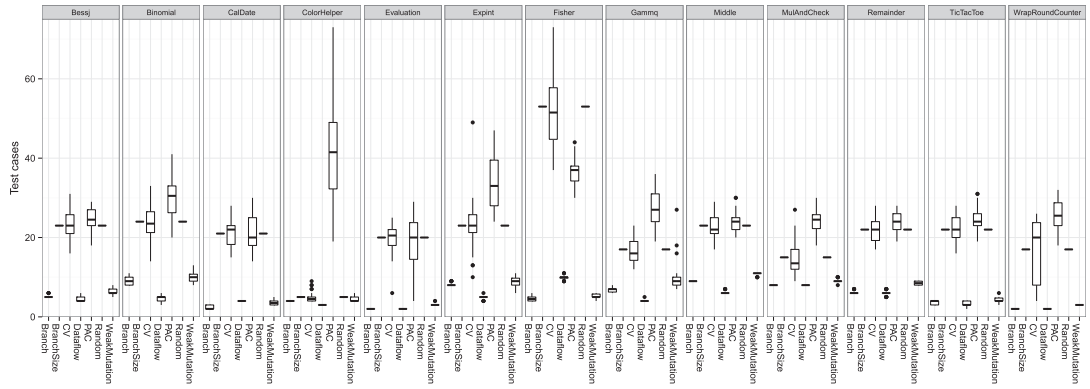
Figure 10. Comparison between BESTEST$_{CV}$ and alternative criteria for discrete systems. PAC, probably approximately correct; CV, cross validation.



DOI: 10.1002/stvr




(a)Mutation Scores




(b)Test Set Sizes

Figure 11. Comparison between BESTEST$_{CV}$ and alternative criteria for numerical systems. PAC, probably approximately correct; CV, cross validation.

Table V. Cliff's delta for comparison of mutation scores against BESTEST$_{CV}$.

| | D. flow | Branch | PAC | Mut. | Branch+ | Rnd. |
|---|---|---|---|---|---|---|
| | | | Numerical systems | | | |
| Expint | −1.000 | −1.000 | 0.034 | −1.000 | −0.952 | −1.000 |
| Remainder | −0.133 | −0.656 | −0.003 | **0.915** | **0.308** | −1.000 |
| TicTacToe | −0.949 | −0.882 | −0.037 | −1.000 | −0.510 | −1.000 |
| Gammq | −0.989 | −0.734 | **0.448** | −0.162 | −0.505 | −1.000 |
| Fisher | −0.900 | −0.833 | −0.100 | 0.000 | −0.002 | −0.167 |
| Middle | −1.000 | −0.570 | **0.175** | **0.944** | −0.486 | −1.000 |
| Mul&Check | −0.457 | −0.438 | **0.506** | **0.517** | −0.114 | −1.000 |
| W.R.Counter | −0.667 | −0.667 | −0.467 | **0.333** | −0.386 | −0.667 |
| Col.Hlp | −0.669 | −0.463 | **0.867** | **0.867** | −0.241 | −0.480 |
| Bessj | −0.728 | −0.460 | −0.092 | **0.374** | **0.442** | −0.539 |
| Binomial | −1.000 | −0.929 | 0.101 | −0.834 | −0.800 | −1.000 |
| Eval. | −0.984 | −0.986 | −0.500 | 0.033 | −0.500 | −0.967 |
| CalDate | −0.733 | −0.817 | −0.721 | **0.829** | −0.528 | −0.733 |
| | | | Discrete systems | | | |
| Luhn | −1.000 | −1.000 | −0.246 | −1.000 | −1.000 | −1.000 |
| BMI | −1.000 | −0.911 | −0.148 | −0.579 | −0.416 | −1.000 |
| tcas | −1.000 | −1.000 | −0.553 | −1.000 | −1.000 | −1.000 |
| WBS | −1.000 | −1.000 | 0.131 | −1.000 | −1.000 | −1.000 |
| Triangle | −1.000 | −1.000 | −0.082 | −1.000 | −0.070 | −1.000 |

PAC, probably approximately correct.



DOI: 10.1002/stvr

and discrete systems in Table IV). Of course, this introduces a bias, and the results should be interpreted accordingly; this will show how other techniques compare against BESTEST$_{CV}$ , *assuming that it is suitably configured*.

Because the differences between mutation scores are often difficult to discern, we provide in Table V Cliff's delta scores that compare, for each SUT, the mutation score distribution against BESTEST$_{CV}$. To recap, a negative delta value indicates the mutation scores for the other technique tend to be smaller than those for BESTEST$_{CV}$. A greater value indicates that the mutation scores tend to be greater than for BESTEST$_{CV}$. The magnitude of the delta conveys the *probability* of one population being greater or smaller than the other; it does not convey the magnitude. Romano *et al.* [52] suggest that a value $\leq$0.147 should be treated as negligible. Any delta greater than this, favouring a different technique, is highlighted in bold in the table. The delta is often relatively small for mutation scores; the box plots serve as a useful basis for assessing this.

For numerical systems, the box plots show that in many cases, there is little to distinguish the performance of different techniques.

Table V shows that, for numerical systems, there are seven cases in which Mutation Testing achieves higher mutation scores than BESTEST$_{CV}$. Although the distances are significant from a Cliff's delta perspective, the box plots for these systems show that the distances in scores are very small indeed. This is possibly due to the fact that the systems for which Mutation produces higher mutation scores than BESTEST$_{CV}$ are relatively small systems, with less scope for hard-to-hit mutants, and a naturally high number of easily killed mutants.

Looking at the four largest numerical systems in terms of lines (Bessj, Binomial, Expint and Gammq), the situation is different. On Bessj, there is a negligible difference in mutation score between all approaches (apart from notable outliers for BESTEST$_{CV}$ and BESTEST$_{PAC}$). For the other three systems, BESTEST$_{PAC}$ and BESTEST$_{CV}$ significantly outperform the other systems. The difference in mutation scores is especially marked for Binomial and Expint.

For discrete systems, results are more marked. For all systems apart from Triangle, the difference in mutation score can be discerned visually from the box plots, indicating that BESTEST$_{CV}$ and BESTEST$_{PAC}$ clearly outperform the other criteria. The Cliff's delta scores indicate that, for Triangle, there is a negligible difference in performance between BESTEST$_{CV}$ and BranchSize.

Looking at the test sizes (Figures 10(b) and 11(b)), the numbers of test cases generated for both BESTEST$_{PAC}$ and BESTEST$_{CV}$ tend to be significantly larger than for the other criteria[‖]. In most cases, the number of tests ranges between 20–35, whereas other approaches are minimized to between 5 and 15 tests. In most cases, BESTEST$_{CV}$ requires fewer test cases than BESTEST$_{PAC}$. For two systems (TCAS and ColorHelper), BESTEST$_{CV}$ produces a more comparable number of test cases to alternative criteria (a mean of 19 and 5 tests, respectively).

The figures for Branch+ (shorthand for BranchSize) merit further discussion. These test sets were optimized for branch coverage, but expanded with random tests to match the size of those produced by BESTEST$_{CV}$. Only in two instances (Remainder and Bessj) did the test sets marginally outperform BESTEST$_{CV}$. This shows that the improvement over branch coverage is not merely due to the larger sizes.

It is not surprising that higher behavioural adequacy comes at the cost of larger test suites. When resources available for testing are limited, does this mean that behavioural adequacy is not an option? The answer is no—although in our experiments, we aimed to maximize behavioural adequacy as much as possible within the 10-min bound, this may not reflect practical usage. For example, one might first decide on the number of test cases to generate and then use the BESTEST$_{CV}$ approach to find the best test set of that size. Furthermore, clearly 100% adequacy is an unrealistic goal in most cases, so one might determine a given threshold of a minimum acceptable behavioural adequacy and then generate only as many tests as necessary for this threshold. Unlike traditional syntactic coverage criteria, the resulting test set comes with a more reliable measurement of the covered program behaviour.

[‖]We ignore figures for Random and BranchSize, because these are calibrated based on the numbers of tests for BESTEST$_{CV}$.



DOI: 10.1002/stvr

*5.2.5. Threats to validity.* In this paper, we argue that the link between structural coverage criteria and behaviour is tenuous, yet as a proxy measurement for our analysis, we had to resort to mutation analysis. This is a threat to *construct validity*, that is, how the performance of a testing technique is defined. Traditionally, test generation techniques are compared in terms of the achieved code coverage or mutation scores. However, as discussed at length in Section 2, these are at best indicative of the true adequacy of the test sets and are at worst misleading. Given that the systems used in this study are tested at a unit level, the scope for subtly mutating the source code is often restricted; most mutants are exposed by the mere execution of *any* code. It is well known that mutation scores are inflated by trivial mutants, yet at the same time, also too strict because of equivalent mutants. Consequently, our results cannot be interpreted as a quantification of the relation between mutation score and behavioural adequacy. However, considering that mutants are generally assumed to be similar to real faults [53], the mutation analysis at least allowed us to establish that higher adequacy leads to higher fault-detection ability, as indicated by mutation scores.

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 min when generating test sets, which may be more time than necessary to satisfy simple criteria like branch coverage. However, it is difficult to determine what precisely would constitute a fair comparison, as this varies greatly between individual subjects and criteria.

Threats to *external validity* concern the generalization to other types of software, common for any empirical analysis. We have selected 18 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$_{PAC}$ prototype (e.g. limitation to primitive data types and model inference only for single outputs), which is the subject of ongoing work. However, to reduce sampling bias, the evaluation set represents all cases satisfying the previous constraints on which we have applied our experiments, and it is based on an extensive literature review. The chosen systems are mainly small, and this was influenced by the use of off-the-shelf machine learners. However, in principle, the techniques presented apply to any system for which a suitable machine learning algorithm is available, and our results indicate that the benefit of using behavioural adequacy is higher for larger systems (cf. results on WBS and TCAS, which are based on industrial systems). We discuss scalability issues in detail in the succeeding text.

### *5.3. Discussion*

*5.3.1. Using behavioural coverage to assess test criteria.* The research questions have provided insights into the general capability of behavioural adequacy to assess test sets and to generate test cases that expose faults. However, there has been no explicit consideration of its value as a metric in its own right. We now show how the notion of behavioural coverage (as defined in Section 3.3) can serve as a useful performance indicator for different test set criteria.

In RQ4, a particular BESTEST$_{CV}$ configuration was compared against standard testing criteria, and the resulting test sets were compared in terms of mutation coverage (as well as test set size). If our behavioural coverage measure is to be of value, then one should be able to use it as a suitable indicator of the effectiveness of the test set.

As an illustration of this, Figure 12 plots the behavioural coverage in its two dimensions for each type of criterion. Given that we have multiple data points, we plot each coordinate by calculating the harmonic mean for each dimension[**].

When compared with the mutation score box plots from RQ4, the behavioural coverage plots indicate that this two-dimensional metric tends to be a good indicator of test set quality. If we focus on the systems for which there were clear ‘winners’ and ‘losers’ in terms of mutation scores, these

[**]This is merely an illustrative example of how the metric could be used and is not meant to be prescriptive; as with other two-dimensional metrics (such as Precision and Recall [17]), the relationship between the two dimensions can be plotted and visualized in numerous ways.



DOI: 10.1002/stvr

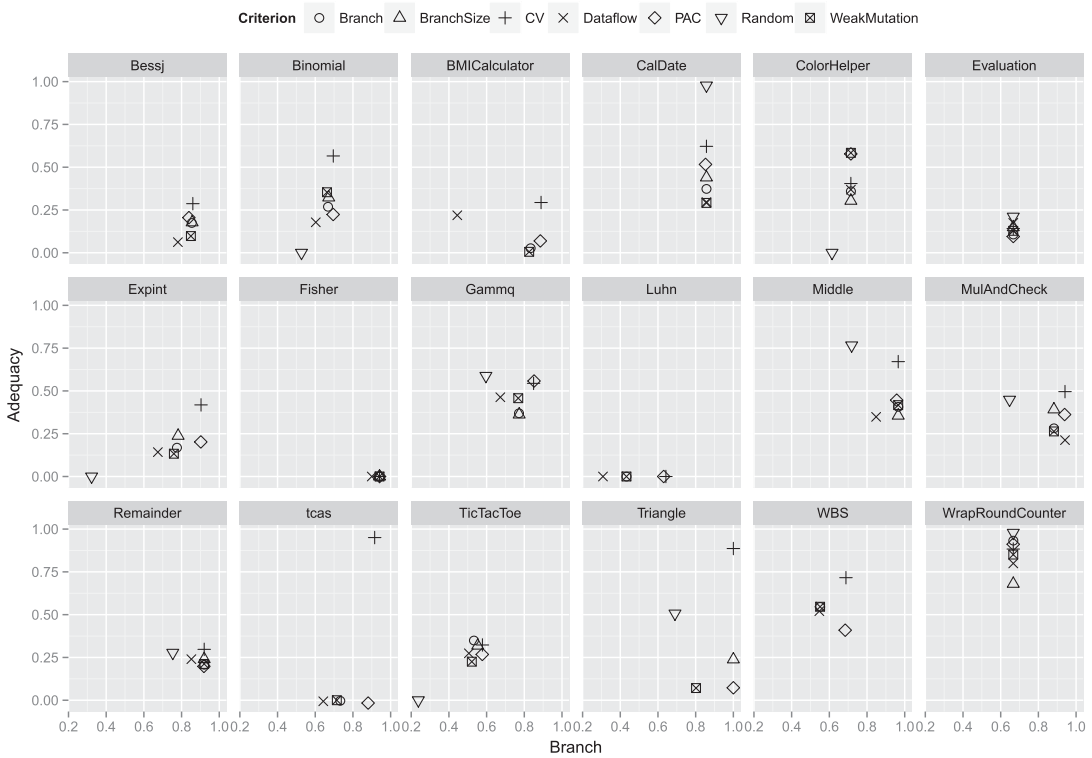



Figure 12. Comparison of different test criteria in terms of behavioural coverage. PAC, probably approximately correct; CV, cross validation.

relationships are also replicated in the scatter plot. If a criterion dominates other criteria in terms of both branch coverage and adequacy, it will tend to yield a higher mutation score.

For example, the Expint example demonstrates this relationship particularly clearly. PAC and CV both achieve the highest mutation scores. In the chart, CV dominates on both dimensions; PAC matches CV on the branch-coverage front (dominating the rest) but is approximately matched by BranchSize in terms of adequacy. However, looking at the mutation scores, BranchSize is the only other technique to contain test cases (represented as outliers) that occasionally match PAC and CV. The rest of the criteria are clustered together (and produce similar mutation scores). Finally, the Random test sets produce very poor mutation scores, and similarly, the coordinate in the graph is located firmly in the bottom left-hand corner of the chart.

As is to be expected, the specific BESTEST$_{CV}$ configuration we have chosen here does not *always* produce the strongest behavioural coverage scores. The configuration is never bested in terms of branch coverage but fails to produce test sets that yield the most accurate models in four cases (CalDate, ColorHelper, Evaluation and WrapRoundCounter[††]). Looking back to the mutation scores (Figures 10 and 11), the coordinates tend to agree with the mutation scores. In every case (including the four mentioned in the previous text), the best-performing techniques are amongst the top performers for mutation scores. Conversely, techniques that score poorly in behavioural coverage (e.g. Random in Expint or Binomial) perform just as badly in the mutation scores.

An interesting observation in Figure 12 is that there are several cases where BESTEST$_{CV}$ leads to higher branch coverage than optimizing branch coverage directly (e.g. Binomial, BMICalculator, Expint, Gammq, Luhn, MulAndCheck, TCAS, TicTacToe, Triangle and WBS); in the remaining

[††]There are cases where a technique produces lower branch coverage and a higher adequacy score (e.g. TicTacToe or Middle). However, as discussed in Section 3.3, this tends to be due to the fact that the test set has failed to capture enough of the program behaviour, so although the model is accurate, this is with respect to a narrower range of behaviour.



DOI: 10.1002/stvr

cases, the branch coverage is the same. This suggests that the exploration of additional behaviour helps in reaching additional code.

The fact that a single configuration of BESTEST$_{CV}$ does not always outperform all other techniques on all subject systems is due to the fact that it is ultimately founded on the capability of a model inference algorithm. Model inference algorithms are subject to Wolpert and Macready's 'no free lunch' theorem [54]: *No learning algorithm achieves consistently better generalization performance than any other over all possible target functions*. In other words, given a particular BESTEST$_{CV}$ configuration, some systems under test will be particularly 'learnable' and thus yield strong test sets, others will not. Thus, choosing a suitable learning configuration ultimately relies upon human judgement, factoring in the various characteristics of the learning problem (i.e. the subject system), and the learning algorithm.

*5.3.2. Luhn and Fisher.* From the results, there are two systems that would seem to be problematic for our approach. None of the test sets generated for Luhn or Fisher gave rise to accurate models. In fact, no single run for either system yielded an adequacy greater than 0. In other words, at no point was any of the model inference algorithms able to produce a model that reliably predicted the outputs given the inputs. Furthermore, in both cases, the mutation scores tended to be relatively high (in the case of Fisher, the mean mutation score was almost perfect at 0.997).

At first glance, the results would seem to indicate that our technique for assessing behavioural adequacy has simply failed. However, a closer investigation of these cases in fact reveals the contrary. It turns out that Luhn offers a perfect example of the problems discussed in Section 2.1, where syntax-driven coverage measures such as mutation coverage can be dangerously misleading.

In the case of Fisher, the function computes the Fisher–Snedecor distribution [37]. It takes as input three doubles: the value $x$ and two parameters $a$ and $b$. It returns a NaN if any of the parameters is negative and only provides a value between 0 and 1 (of interest to the learner) for a very small window in the input domain. We observed that it was the presence of NaNs that tended to lead to these poor results from WEKA. We devised a filtering procedure to remove the NaNs from the training sets, whereupon adequacy values rose to the region of 0.4. This did not produce an increase in the mutation score or branch coverage, but simply because these were already at extremely high levels.

In the case of Luhn, the problem comes down to a limitation of the test generator. Luhn is a procedure by which to validate credit card numbers for different card providers. The constraints that govern whether a particular type of card number is valid are quite 'narrow'; the number has to be of the correct length, and different intervals of the number have to fall into particular ranges, depending on the card provider. Furthermore, the input is actually a string, which is converted into a sequence of digits. EVOSUITE attempts numbers for every type of card but fails to ever produce a valid card number, which means that the output is always the same (invalid)—making it impossible to infer a sensible model. To validate that this is indeed a testability problem, we added five valid credit card numbers (one for each card type) to the seeding used by EVOSUITE (by adding public static fields to the class). Doing so immediately increases the achieved branch coverage to around 73%, with adequacy and mutation score both going up to around 79%.

These are two examples of cases where our BESTEST$_{CV}$ implementation did not work 'out of the box'. To maintain validity, we resisted the urge to manipulate the experimental setup. However, here, we have shown that such problems are in fact relatively straightforward to address in practice. If the set of test observations contains values that are incompatible with the learner, they can be recoded. If the system suffers from testability problems, often strategies such as seeding can be used to guide the test generation towards better inputs.

*5.3.3. Scalability.* The use of a genetic algorithm, coupled with the model inference and cross validation, all indicate that this approach could be very expensive. Genetic algorithms rely on the ability to execute large numbers of tests. Cross validation again requires the repeated inference of a model for each chromosome.

To assess the scalability of our approach, we re-ran our experiments for the two top-performing configurations from RQ3, whilst varying the time limit. We ran 10 batches of experiments with 30



DOI: 10.1002/stvr

repetitions each, ranging from 1 to 10 min, tracking intermediate values. In these runs, we observed that the total number of tests executed per minute increases at an approximately constant rate, although this rate varies from program to program.

The time spent on model inference/adequacy assessment tends to be relatively modest. For our experiments, on average, 7.6% of time was spent on model inference and evaluation, whereas 70% was spent on the actual execution of tests. These 7.6% represents an overhead that is not present when optimizing for simpler criteria such as branch coverage. However, even so, the dominating factor remains test execution, as is the case with any search-based test generation approach. Consequently, we do not anticipate the use of model inference to be a major scalability issue. However, it should be noted that these particular statistics are specific to the configurations considered. Some model inference algorithms scale better than others and could lead to different balances in terms of how time is used. For example, inferring models for larger state-based systems is likely to increase the share of time that is spent on model inference.

Although test execution is in general the dominating factor in search-based testing, there is evidence that search-based test generation approaches also scale to large industrial systems [55–58]; thus, we also expect our approach to scale to larger systems. However, scalability might also be affected by the larger test sets that optimization for behavioural adequacy leads to. The larger the test sets in the search population, the more of the search budget is spent on evaluating the fitness of each individual, and that has an impact on the exploration the search can perform within a fixed search budget. However, we observed that the size of the test sets in the search population remains more or less stable throughout the search. The largest growth happens during phases where there is no adequacy (e.g. in the case of Fisher, this was the case throughout the search). However, the implementation of the GA in the underlying EvoSuite ranks two individuals by their size if their fitness values are equal, such that smaller test sets have a higher probability of being selected for reproduction. Consequently, during phases of the search where there is no exploration of new coverage or adequacy, redundant test cases are removed from the test sets, and thus behavioural adequacy does not lead to bloat [31].

*5.3.4. The limits of model inference.* The work presented in this paper relies on the notion that the input/output behaviour of a program is approximately learnable by a machine learning algorithm. We have seen (in RQ1) how the accuracy of an inferred model can vary significantly, depending on a range of factors.

The question of how different factors lead to the successful inference of a model (irrespective of whether the system in question is a software function) is almost impossible to answer in the general case. We present some of the key factors (specifically with respect to software systems) in the succeeding text:

1. **Trace diversity and representativeness:** The supply of a representative set of examples from which to learn is key; if there is zero evidence of the potential for a particular facet of behaviour, then no algorithm will produce a model that contains it. This is the problem that prevented the inference of useful models for Luhn and Fisher. This problem is typical to dynamic analysis and was discussed at length by Ernst [5].
2. **Number of input variables:** A large number of input parameters can give rise to the phenomenon known as *over-fitting* [15], where the model wrongly incorporates variables that in fact have no effect on the output. Although we did not do so for our experiments, over-fitting can be avoided by eliding unnecessary variables from the execution traces.
3. **Non-determinism/noisy output:** The model inference algorithms in this paper are statistical at heart; they draw probabilistic relationships between input variable values, and the corresponding outputs. For systems where the output is non-deterministic, care has to be taken. It is still of course possible to infer a statistical model of a non-deterministic system, but this is contingent upon the fact that the effect of the non-determinism on the output is sufficiently captured in the traces (see point 1).
4. **Types:** The types of inputs and outputs are key to the selection of model inference algorithm. Our examples all deal with some combination of numerical or string inputs and outputs. How-



DOI: 10.1002/stvr

ever, software systems tend to operate on complex types: objects, trees, nested lists and so on. At the moment, our approach has to adopt the same approach as other inference-based systems such as Daikon [59]. Where possible, we flatten complex types into their primitive parts, and lists are summarized in terms of their size.

5. **Non-linearity:** Certain behavioural characteristics are harder to infer than others or require a huge training sample to infer. Non-linear behaviour (especially pronounced in BMICalculator and Fisher) means that tiny variances in a given input variable can cause disproportionately large variances in the output, and only for small intervals in the input space.

For each of the aforementioned characteristics, machine learning algorithms have been proposed to cater for these situations. Certain algorithms are particularly good with a sparse sample of examples. Others can deal with special data structures. Others excel at dealing with noisy data and so on. Invariably, different algorithms excel in one aspect but trade off performance in another. This is the root of Wolpert's *no free lunch* theorem [54], mentioned in RQ4.

The BESTEST$_{CV}$ approach has been validated on a selection of the most popular algorithms, using their default parameter configurations. However, in principle, any classifier can be applied, enabling the tester to exploit knowledge about the type of system being tested. Choosing a suitable algorithm requires expertise and knowledge about the circumstances and data against which it will be applied. This may not necessarily be possible for all software systems, given the current state of the art in machine learning. However, BESTEST$_{CV}$ provides a framework within which to incorporate future algorithms as they appear.

## 6. CONCLUSIONS AND CONSEQUENCES

What test cases are necessary to ensure that a test set will expose all of the faults in a program? How can they be collected? How can the final test set be assessed? These questions lie at the heart of Djikstra's famous assertion that tests can only show the presence of faults but not their absence, and have formed the basis for the bulk of software testing research since.

So far, the vast majority of approaches have focussed on the source code of a system. Test cases should execute all statements, branches, data–flow relations or mutants. Such syntax-centric views of testing are rarely sufficient. In practice, they are at best used as minimum requirements for a test set. A test set that satisfies these criteria rarely finds all of the mutants, for example.

One exciting, alternative approach to the syntactic approach was first proposed by Weyuker in 1983 [2]. She pointed out that if we can correctly infer the behaviour of a system from its test set, it can be concluded that we have tested the behaviour adequately. Although exciting in theory, putting this into practice raises several significant challenges. It is necessary to infer a suitable model that represents the test set, to reliably quantify the accuracy of the model and, above all, to develop a suitable procedure by which to generate test sets that will optimize the model accuracy. As a result, this idea has remained predominantly in the theoretical domain.

This paper shows how the idea can be put into practice. It shows how model inference can be used to provide a much more reliable assessment of test sets, which can in turn form a much more powerful basis for the generation of rigorous test sets. Our experiments have shown that if configured properly, optimizing test generation with respect to behavioural adequacy can significantly outperform current baseline techniques in terms of fault detection, especially for larger functions with a complex branching structure.

The work in this paper was primarily concerned with the development of a proof of concept. As such, it has only been applied to Java units, where inputs and outputs can easily be reduced to simple primitive types. To make the approach more broadly applicable, our future work will focus on the following five dimensions:

1. **Adapt to systems with arbitrarily complex inputs and outputs:** The current approach is restricted to systems with a reasonably low number of input variables, and a single output. We will investigate the use of different classes of model inference algorithms (i.e. support vector machine algorithms [60]) that can, in principle, be applied to arbitrarily systems with large numbers of inputs and outputs.



DOI: 10.1002/stvr

2. **Integrating sequential state:** The current approach is restricted to systems that take an input and return an output in a single step. Our future work will explore the use of state machine inference algorithms [13, 61] to enable the application of the technique to sequential-state systems, such as GUIs, network protocols, or APIs.
3. **Alternative syntactic coverage criteria:** In this paper, we used branch coverage as the underlying syntactic coverage criterion for our experiments. This choice seemed reasonable given the nature of our prototype. However, as future work, we will consider the effect of combining other syntactic criteria (e.g. MCDC [24], dataflow or mutation testing) with behavioural adequacy.
4. **Behavioural coverage for specification-based testing:** Although our experiments focus on behavioural adequacy as an additional dimension over syntactic code coverage, this does not mean behavioural coverage needs to be purely a white-box technique. Structural coverage criteria are also commonly used when testing using formal specifications or test models, and we will consider defining behavioural coverage in terms of such black-box coverage criteria.
5. **Applications of the inferred model:** In this paper, we focussed on the evaluation and optimization of test sets with respect to behavioural coverage. However, a behaviourally adequate test set is not necessarily a means to an end: Given a behaviourally adequate test set, there are potential applications of the model that can be inferred from this test set; for example, the model can predict expected outputs even for inputs for which there is not an explicit test.

## ACKNOWLEDGEMENTS

This work is funded by a Google Focused Research Award on 'Test Amplification', the DSTL-funded BATS project DSTLX1000062430 and the DSTL-funded HASTE project. This project has been funded by the EPSRC project 'EXOGEN' (EP/K030353/1).

## REFERENCES

1. Goodenough JB, Gerhart SL. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* 1975; **1**(2):156–173.
2. Weyuker E. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems* 1983; **5**(4):641–655.
3. Valiant L. A theory of the learnable. *Communications of the ACM* 1984; **27**(11):1134–1142.
4. Fraser G, Walkinshaw N. Behaviourally adequate software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Antoniol G, Bertolino A, Labiche Y (eds). IEEE: Montreal, QC, Canada, April 17–21, 2012; 300–309. 2012. ISBN 978-1-4577-1906-6.
5. Ernst MD. Static analysis, dynamic: Synergy and duality. *WODA 2003: ICSE Workshop on Dynamic Analysis,* Portland OR, USA, 2003; 24–27.
6. Zhu H, Hall P, May J. Inductive inference and software testing. *Software Testing, Verification, and Reliability* 1992; **2**(2):69–81.
7. Zhu H. A formal interpretation of software testing as inductive inference. *Software Testing, Verification and Reliability* 1996; **6**(1):3–31.
8. Cherniavsky J, Smith C. A recursion theoretic approach to program testing. *IEEE Transactions on Software Engineering* 1987; **13**.
9. Bergadano F, Gunetti D. Testing by means of inductive program learning. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(2):119–145.
10. Romanik K. Approximate testing and its relationship to learning. *Theoretical Computer Science* 1997; **188**(1–2): 175–194.
11. Briand L, Labiche Y, Bawar Z, Spido N. Using machine learning to refine category-partition test specifications and test suites. *Information and Software Technology* 2009; **51**:1551–1564.
12. Shahbaz M, Groz R. Inferring mealy machines. In *FM, Volume 5850 of Lecture Notes in Computer Science*, Cavalcanti A, Dams D (eds). Springer: Eindhoven, HNetherlands, 2009; 207–222. ISBN 978-3-642-05088-6.
13. Walkinshaw N, Derrick J, Guo Q. Iterative refinement of reverse-engineered models by model-based testing. *Formal Methods (FM),* LNCS, Springer: Eindhoven, Netherlands, 2009; 305–320.
14. Walkinshaw N, Bogdanov K, Derrick J, Paris J. Increasing functional coverage by inductive testing: A case study. *International Conference on Testing Software and Systems (ICTSS),* LNCS: Natal, Brazil, 2010; 126–141.
15. Mitchell T. *Machine Learning*. McGraw-Hill: Boston MA, USA, 1997.
16. Kohavi R. A study of cross-validation and bootstrap for accuracy estimation and model selection, Mellish CS (ed.) Morgan Kaufmann: San Mateo, August 20; 1137–1145. ISBN 1-55860-363-8.
17. Van Rijsbergen CJ. *Information Retrieval*. Butterworths, London, UK, 1979.



DOI: 10.1002/stvr

18. Cohen J. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological Bulletin* 1968; **70**(4):213.
19. Walkinshaw N. Assessing test adequacy for black-box systems without specifications. *Proceedings of the International Conference on Testing Systems and Software (ICTSS'11),* Paris, France, 2011; 209–224.
20. Fallah F, Devadas S, Keutzer K. Occom-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2001; **20**(8):1003–1015.
21. Whalen M, Gay G, You D, Heimdahl MPE, Staats M. Observable modified condition/decision coverage. *Proceedings of the 2013 International Conference on Software Engineering,* IEEE Press: San Francisco, CA, USA, 2013; 102–111.
22. Schuler D, Zeller A. Assessing oracle quality with checked coverage. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST),* IEEE: Berlin, Germany, 2011; 90–99.
23. Frankl PG, Weyuker EJ. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 1988; **14**(10):1483–1498.
24. Chilenski JJ, Miller SP. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 1994; **9**(5):193–200.
25. McMinn P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156.
26. Fraser G, Arcuri A. Evolutionary generation of whole test suites. *International Conference On Quality Software (QSIC),* IEEE Computer Society: Los Alamitos, CA, USA, 2011; 31–40.
27. Arcuri A. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability (STVR)* 2013:119–147. DOI: 10.1002/stvr.1495.
28. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The weka data mining software: An update. *SIGKDD Explorations Newsletter* 2009; **11**:10–18.
29. Lakhotia K, Harman M, McMinn P. A multi-objective approach to search-based test data generation. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation,* ACM: London, UK, 2007; 1098–1105.
30. Fraser G, Arcuri A. The seed is strong: Seeding strategies in search-based software testing. *IEEE International Conference on Software Testing, Verification and Validation (ICST),* Montreal, Canada, 2012; 121–130.
31. Fraser G, Arcuri A. Handling test length bloat. *Software Testing, Verification and Reliability (STVR)* 2013.
32. Fraser G, Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering* 2013; **39**(2):276–291. ISSN 0098-5589. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.14.
33. Schneckenburger C, Mayer J. Towards the determination of typical failure patterns. *4th International Workshop on Software Quality Assurance, co-located with ESEC/FSE'07 (SOQUA'07),* ACM: Dubrovnik, Croatia, 2007; 90–93.
34. The Apache Commons Mathematics Library. Available from: http://commons.apache.org/math [last accessed 2012].
35. Ghani K, Clark JA. Strengthening inferred specifications using search based testing. *Proceedings of the International Workshop on Search-based Software Testing,* IEEE Computer Society: Lillehammer, Norway, 2008; 187–194. ISBN 978-0-7695-3388-9.
36. Apache JMeter. Available from: http://jmeter.apache.org [last accessed 2012].
37. Dorrer E. F-distribution. *Communications of the ACM* 1968; **11**(2):116–117.
38. Java Luhn algorithm for credit card number validation. Available from: http://megasnippets.com/source-codes/java/luhn_algorithm_credit_card_number_validation [last accessed 2012].
39. Sthamer H. The automatic generation of software test data using genetic algorithms., *PhD thesis*, University of Glamorgan, Pontyprid, Wales, UK, April 1996.
40. Poulding S, Clark JA. Efficient software verification: Statistical testing using automated search. *IEEE Transactions on Software Engineering* 2010; **36**(6):763–777. ISSN 0098-5589. DOI: 10.1109/TSE.2010.24.
41. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**:405–435. ISSN 1382-3256.
42. Staats M, Păsăreanu C. Parallel symbolic execution for structural test generation. *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10),* ACM: New York, NY, USA, 2010; 183–194. ISBN 978-1-60558-823-0. DOI: http://doi.acm.org/10.1145/1831708.1831732.
43. Quinlan JR. *C4. 5: Programs for Machine Learning*. Morgan Kaufmann: San Mateo, CA, 1993.
44. Freund Y, Schapire R. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*. Springer: Jerusalem, Israel, 1995; 23–37.
45. Arcuri A, Fraser G. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering (EMSE)* 2013; **18**:1–30. DOI: 10.1007/s10664-013-9249-9.
46. Fraser G, Arcuri A. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* 2014. DOI: 10.1007/s10664-013-9299-z.
47. Vivanti M, Mis A, Gorla A, Fraser G. Search-based data-flow test generation. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE),* IEEE: Pasadena, CA, USA, 2013; 370–379.
48. Cliff N. *Ordinal Methods for Behavioral Data Analysis*. Routledge, Press: New York, USA, 1996.
49. Peng C-YJ, Chen L-T. Beyond Cohen's *d*: Alternative effect size measures for between-subject designs. *The Journal of Experimental Education* 2014; **82**(1):22–50.
50. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)* 2006; **32**(8):608–624.

51. Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '09,* ACM: New York, NY, USA, 2009; 297–298. ISBN 978-1-60558-001-2. DOI: 10.1145/1595696.1595750.
52. Romano J, Kromrey JD, Coraggio J, Skowronek J. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's *d* for evaluating group differences on the nsse and other surveys. *Annual Meeting of the Florida Association of Institutional Research,* Tampa FL, USA, 2006; 1–3.
53. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? [software testing]. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005,* IEEE: St. Louis, MO, USA, 2005; 402–411.
54. Wolpert DH. The lack of a priori distinctions between learning algorithms. *Neural Computation* 1996; **8**(7): 1341–1390.
55. Vos TEJ, Baars AI, Lindlar FF, Kruse PM, Windisch A, Wegener J. Industrial scaled automated structural testing with the evolutionary testing tool. *2010 Third International Conference on Software Testing, Verification and Validation (ICST),* IEEE: Paris, France, 2010; 175–184.
56. Vos TEJ, Lindlar FF, Wilmes B, Windisch A, Baars AI, Kruse PM, Gross H, Wegener J. Evolutionary functional black-box testing in an industrial setting. *Software Quality Journal* 2013; **21**(2):259–288.
57. Gross F, Fraser G, Zeller A. Search-based system testing: High coverage, no false alarms. *Proceedings of the 2012 International Symposium on Software Testing and Analysis,* ACM: Minneapolis, MN, 2012; 67–77.
58. Arcuri A, Iqbal MZ, Briand L. Black-box system testing of real-time embedded systems using random and search-based testing. In *Testing Software and Systems*. Springer: Natal, Brazil, 2010; 95–110.
59. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 2001; **27**(2):99–123.
60. Vapnik V. *The Nature of Statistical Learning Theory*. Springer Science and Business Media, 1999.
61. Walkinshaw N, Taylor R, Derrick J. Inferring extended finite state machine models from software executions. *International Working Conference on Reverse Engineering (WCRE'13),* Koblenz, Germany, 2013.