On The Effectiveness of Whole Test Suite Generation

Andrea Arcuri[1] and Gordon Fraser[2]

[1] Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway
`arcuri@simula.no`,
[2] University of Sheffield, Dep. of Computer Science, Sheffield, UK
`gordon.fraser@sheffield.ac.uk`

Abstract. A common application of search-based software testing is to generate test cases for all goals defined by a coverage criterion (e.g., statements, branches, mutants). Rather than generating one test case at a time for each of these goals individually, *whole test suite generation* optimizes entire test suites towards satisfying all goals at the same time. There is evidence that the overall coverage achieved with this approach is superior to that of targeting individual coverage goals. Nevertheless, there remains some uncertainty on whether the whole test suite approach might be inferior to a more focused search in the case of particularly difficult coverage goals. In this paper, we perform an in-depth analysis to study if this is the case. An empirical study on 100 Java classes reveals that indeed there are some testing goals that are easier to cover with the traditional approach. However, their number is not only very small in comparison with those which are only covered by the whole test suite approach, but also those coverage goals appear in small classes for which both approaches already obtain high coverage.

Key words: automated test generation, unit testing, search-based testing, EvoSuite

1 Introduction

Search-based software engineering has been applied to numerous different tasks in software development [15], and software testing is one of the most successful of these [1, 19]. One particular task in software testing for which search-based techniques are well suited is the task of automated generation of unit tests. For example, there are search-based tools like AUSTIN for C programs [18] or EvoSuite for Java programs [8].

In search-based software testing, the testing problem is cast as a search problem. For example, a common scenario is to generate a set of test cases such that their code coverage is maximized. A code coverage criterion describes a set of typically structural aspects of the system under test (SUT) which should be exercised by a test suite, for example all statements or branches. Here, the search space would consist of all possible data inputs for the SUT. A search algorithm (e.g., a genetic algorithm) is then used to explore this search space to find the input data that maximize the given objective (e.g., cover as many branches as possible).

Traditionally, to achieve this goal a search is carried out on each individual coverage goal [19] (e.g., a branch). To guide the search, the fitness function exploits information like the approach level [24] and branch distance [17]. It may happen that during the search for a coverage goal there are others goals that can be "accidentally" covered, and by keeping such test data one does not need to perform search for those accidentally covered goals. However, there are several potential issues with such an approach:

– *Search budget distribution:* If a coverage goal is infeasible, then all search effort to try to cover it would be wasted (except for any other coverage goals accidentally covered during the search). Unfortunately, determining whether a goal is feasible or not is an undecidable problem. If a coverage goal is trivial, then it will typically be covered by the first random input. Given a set of coverage goals and an overall available budget of computational resources (e.g., time), how to assign a search budget to the individual goals to maximise the overall coverage?
– *Coverage goal ordering:* Unless some smart strategies are designed, the search for each coverage goal is typically independent, and potentially useful information is not shared between individual searches. For example, to cover a nested branch one first needs to cover its parent branch, and test data for this latter could be use to help the search for the nested branch (instead of starting from scratch). In this regard, the order in which coverage goals are sought can have a large impact on final performance.

To overcome these issues, in previous work we introduced the *whole test suite approach* [11,12]. Instead of searching for a test for each individual coverage goal in sequence, the search problem is changed to a search for a set of tests that covers all coverage goals at the same time; accordingly, the fitness function guides to cover all goals. The advantage of such an approach is that both the questions of how to distribute the available search budget between the individual coverage goals, and in which order to target those goals, disappear. With the whole test suite approach, large improvements have been reported for both branch coverage [11] and mutation testing [12].

Despite this evidence of higher overall coverage, there remains the question of how the use of whole test suite generation influences individual coverage goals. Even if the whole test suite approach covers more goals, those are not necessarily going to be a superset of those that the traditional approach would cover. Is the higher coverage due to more *easy* goals being covered? Is the coverage of *difficult* goals adversely affected? Although higher coverage might lead to better regression test suites, for testing purposes the difficult coverage goals might be more "valuable" than the others. So, from a practical point of view, preferring the whole test suite approach over the traditional one may not necessarily be better for practitioners in industry.

In this paper, we aim to empirically study in detail how the whole test suite approach compares to the traditional one. In particular, we aim at studying whether there are specific coverage goals for which the traditional approach is better and, if that is the case, we want to characterise those scenarios. Based on an empirical study performed on 100 Java classes, our study shows that indeed there are cases in which the traditional approach provides better results. However, those cases are very rare (nearly one hundred times less) compared to the cases in which only the whole test suite approach is able to cover the goals. Furthermore, those cases happen for small classes for which average branch coverage is relatively high.

This paper is organised as follows. Section 2 provides background information, whereas the whole test suite approach is discussed in details in Section 3. The performed empirical study is presented in Section 4. A discussion on the threats to the validity of the study follows in Section 5. Finally, Section 6 concludes the paper.

2 Background

Search-based techniques have been successfully used for test data generation (see [1, 19] for surveys on this topic). The application of search for test data generation can be traced back to the 70s [20], and later the key concepts of *branch distance* [17] and *approach level* [24] were introduced to help search techniques in generating the right test data.

More recently, search-based techniques have also been applied to test object-oriented software (e.g., [13, 21–23]). One specific issue that arises in this context is that test cases are sequences of calls, and their length needs to be controlled by the search. Since the early work of Tonella [22], researchers have tried to deal with this problem, for example by penalizing the length directly in the fitness function. However, longer test sequences can lead to achieve higher code coverage [2], yet properly handling their growth/reduction during the search requires special care [10].

Most approaches described in the literature aim at generating test suites that achieve as high as possible branch coverage. In principle, any other coverage criterion is amenable to automated test generation. For example, mutation testing [16] is often considered a worthwhile test goal, and has been used in a search-based test generation environment [13].

When test cases are sought for individual goals in such coverage based approaches, it is important to keep track of the accidental collateral coverage of the remaining goals. Otherwise, it has been proven that random testing would fare better under some scalability models [5]. Recently, Harman et al. [14] proposed a search-based multi-objective approach in which, although each coverage goal is still targeted individually, there is the secondary objective of maximizing the number of collateral goals that are accidentally covered. However, no particular heuristic is used to help covering these other coverage goals.

All approaches mentioned so far target a single test goal at a time – this is the predominant method. There are some notable exceptions in search-based software testing. The works of Arcuri and Yao [6] and Baresi et al. [7] use a single sequence of function calls to maximize the number of covered branches while minimizing the length of such a test case. A drawback of such an approach is that there can be conflicting testing goals, and it might be impossible to cover all of them with a single test sequence regardless of its length.

To overcome those issues, in previous work we proposed the whole test suite approach [11,12]. In this approach, instead of evolving individual tests, whole test suites are evolved, with a fitness function that considers all the coverage goals at the same time. Promising results were obtained for both branch coverage [11] and mutation testing [12].

3 Whole Test Suite Generation

To make this paper self-contained, in this section we provide a summarised description of the traditional approach used in search-based software testing and the whole test suite approach. For more details on the traditional approach, the reader can for example refer to [19, 24]. For the whole test suite approach, the reader can refer to [11, 12].

Given a SUT, assume X to be the set of coverage goals we want to automatically cover with a set of test cases (i.e., a test suite) T. Coverage goals could be for example branches if we are aiming at branch coverage, or any other element depending on the chosen coverage criterion (e.g., mutants in mutation testing).

3.1 Generating Tests for Individual Coverage Goals

Given $|X| = n$ coverage goals, traditionally there would be one search for each of them. To give more gradient to the search (instead of just counting "yes/no" on whether a goal is covered), usually the approach level $\mathcal{A}(t,x)$ and branch distance $d(t,x)$ are employed for the fitness function [19, 24]. The approach level $\mathcal{A}(t,x)$ for a given test t on a coverage goal $x \in X$ is used to guide the search toward such target branch. It is determined as the minimal number of control dependent edges in the control dependency graph between the target branch and the control flow represented by the test case. The branch distance $d(t,x)$ is used to heuristically quantify how far a predicate in a branch x is from being evaluated as true. In this context, the considered predicate x_c is taken for the closest control dependent branch where the control flow diverges from the target branch. Finally, the resulting fitness function to minimize for a coverage goal x will be:

$$f(t,x) = \mathcal{A}(t,x) + \nu(d(t,x_c)) \ ,$$

where ν is any normalizing function in [0,1] (see [3]). For example, consider this trivial function:

```
public static void foo(int z){
  if(z > 0)
    if(z > 100)
      if(z > 200)
        ; //target
}
```

With a test case t_{50} having the value $z = 50$, the execution would diverge at the second if-condition, and so the resulting fitness function for the target $x_{z>200}$ would be

$$f(t_{50}, x_{z>200}) = 1 + \nu(|50 - 100| + 1) = 1 + \nu(51) \ ,$$

which would be higher (i.e., worse) than a test case having $z = 101$:

$$f(t_{101}, x_{z>200}) = 0 + \nu(|101 - 200| + 1) = 0 + \nu(100) \ .$$

While implementing this traditional approach, we tried to derive a faithful representation of current practice, which means that there are some optimizations proposed in the literature which we did not include:

– New test cases are only generated for branches that have not already been covered through collateral coverage of previously created test cases. However, we do not evaluate the collateral coverage of all individuals during the search, as this would add a significant overhead, and it is not clear what effects this would have given the fixed timeout we used in our experiments.
– When applying the one goal at a time approach, a possible improvement could be to use a *seeding* strategy [24]. During the search, we could store the test data that have good fitness values on coverage goals that are not covered yet. These test data can then be used as starting point (i.e., for seeding the first generation of a genetic algorithm) in the successive searches for those uncovered goals. However, we decided not to implement this, as reference [24] does not provide sufficient details to reimplement the technique, and there is no conclusive data regarding several open questions; for example, potentially a seeding strategy could reduce diversity in the population, and so in some cases it might in fact reduce the overall performance of the search algorithm.
– The order in which coverage goals are selected might also influence the result. As in the literature usually no order is specified (e.g., [14, 22]), we selected the branches in random order. However, in the context of procedural code approaches to prioritize coverage goals have been proposed, e.g., based on dynamic information [24]. However, the goal of this paper is neither to study the impact of different orders, nor to adapt these prioritization techniques to object-oriented code.
– In practice, when applying a single goal strategy, one might also bootstrap an initial random test suite to identify the trivial test goals, and then use a more sophisticated technique to address the difficult goals; here, a difficult, unanswered question is when to stop the random phase and start the search.

3.2 Whole Test Suite Generation

For the whole test suite approach, we used exactly the same implementation as in [11,12]. In the *Whole* approach, the approach level $\mathcal{A}(t,x)$ is not needed in the fitness function, as all branches are considered at the same time. In particular, the resulting fitness function to minimize for a set of test cases T on a set of branches X is:

$$w(T,X) = \sum_{x \in X} d(T,x) \ ,$$

where $d(T,x)$ is defined as:

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

Note that these X coverage goals could be considered as different objectives. Instead of linearly combining them in a single fitness score, a multi-objective algorithm could be used. However, a typical class can have hundreds if not thousands of objectives (e.g., branches), making a multi-objective algorithm not ideal due to scalability problems.

4 Empirical Study

In this paper, we carried out an empirical study to compare the whole test suite approach (*Whole*) with the traditional one branch at a time approach (*One-Branch*). In particular, we aim at answering the following research questions:

RQ1: Are there coverage goals in which *OneBranch* performs better?

RQ2: How many coverage goals found by *Whole* get missed by *OneBranch*?

RQ3: Which factors influence the relative performance between *Whole* and *OneBranch*?

4.1 Experimental Setup

In this paper, for the case study we randomly chose 100 Java classes from the SF100 corpus [9], which is a collection of 100 projects randomly selected from the SourceForge open source software repository. We randomly selected from SF100 to avoid possible bias in the selection procedure, and to have higher confidence to generalize our results to other Java classes as well. In total, the selected 100 classes contain 2,383 branches, which we consider as test goals.

The SF100 currently contains more than 11,000 Java classes. We only used 100 classes instead of the entire SF100 corpus due to the type of experiments we carried out. In particular, on the selected case study, for each class we ran EvoSuite in two modes: one using the traditional one branch at a time approach

(*OneBranch*), and the other using the whole test suite approach (*Whole*). To take randomness into account, each experiments was repeated 1,000 times, for a total of $100 \times 2 \times 1{,}000 = 200{,}000$ runs of EvoSuite.

When choosing how many classes to use in a case study, there is always a tradeoff between the number of classes and the number of repeated experiments. On one hand, a higher number of classes helps to generalize the results. On the other hand, a higher number of repetitions helps to better study in detail the differences on specific classes. For example, given the same budget to run the experiments, we could have used 10,000 classes and 10 repetitions. However, as we want to study the "corner cases" (i.e., when one technique completely fails while the other compared one does produce results), we gave more emphasis on the number of repetitions to reduce the random noise in the final results.

Each experiment was run for up to three minutes (the search on a class was also stopped once 100% coverage was achieved). Therefore, in total the entire case study took up to $600{,}000/(24 \times 60) = 416$ days of computational resources, which required a large cluster to run. When running the *OneBranch* approach, the search budget (i.e., the three minutes) is equally distributed among the coverage goals in the SUT. When the search for a coverage goal finishs earlier (or a goal is accidentally covered by a previous search), the remaining budget is redistributed among the other goals still to cover.

To properly analyse the randomized algorithms used in this paper, we followed the guidelines in [4]. In particular, when branch coverage values were compared, statistical differences were measured with the Wilcoxon-Mann-Whitney U-test, where the effect size was measured with the Vargha-Delany $\hat{A}_{12}$. A $\hat{A}_{12} = 0.5$ means no difference between the two compared algorithms.

When checking how often a goal was covered, because it is a binary variable, we used the Fisher exact test. As effect size, we used the odds ratios, with a $\delta = 1$ correction to handle the zero occurrences. When there is no difference between two algorithms, then the odds ratio is equal to one. Note, in some of the graphs we rather show the natural logarithm of the odds ratios, and this is done only to simplify their representation.

4.2 Results

Table 1 shows the average coverage obtained for each of the 100 Java classes. The results in Table 1 confirm our previous results in [11]: the whole test suite approach leads to higher code coverage. In this case, the average branch coverage increases from 67% to 76%, with a 0.62 effect size. However, there are two classes in which the *Whole* approach leads to significantly worse results: `RecordingEvent` and `BlockThread`. Two cases out of 100 could be due to the randomness of the algorithm, although having 1,000 repetitions does reduce the probability of this. However, in both cases the *Whole* approach does achieve relatively high coverage (i.e, 84% and 90%).

Looking at `RecordingEvent` in detail, we see that there are some branches that are never covered by the *Whole* approach, but sometimes by *OneBranch* (see Figure 1). Specifically, there is a disjunction of two conditions on two static

Table 1. For each class, the table reports the average branch coverage obtained by the *OneBranch* approach and by the *Whole* approach. Effect sizes and p-values of the comparisons are in bold when the p-values are lower than 0.05.

Class	OneB.	Whole	$\hat{A}_{12}$	p-value
MapCell	1.00	1.00	0.50	-
br.com.jnfe.base.CST_COFINS	0.99	1.00	**0.53**	< **0.001**
ch.bfh.egov.nutzenportfolio.service.kategorie.KategorieDaoService	0.00	0.01	**0.97**	< **0.001**
com.browsersoft.aacs.User	0.51	0.87	**1.00**	< **0.001**
com.browsersoft.openhre.hl7.impl.config.HL7SegmentMapImpl	0.04	0.99	**0.99**	< **0.001**
com.gbshape.dbe.sql.Select	0.06	0.08	**0.57**	< **0.001**
com.lts.caloriecount.ui.budget.BudgetWin	0.11	0.12	**0.64**	< **0.001**
com.lts.io.ArchiveScanner	0.07	0.45	**0.99**	< **0.001**
com.lts.pest.tree.ApplicationTree	0.00	0.00	0.50	-
com.lts.swing.table.dragndrop.test.RecordingEvent	0.95	0.84	**0.03**	< **0.001**
com.lts.swing.thread.BlockThread	0.98	0.90	**0.27**	< **0.001**
com.werken.saxpath.XPathLexer	0.51	0.73	**1.00**	< **0.001**
corina.formats.TRML	0.03	0.21	**0.99**	< **0.001**
corina.map.SiteListPanel	0.00	0.00	**0.99**	< **0.001**
de.huxhorn.lilith.data.eventsource.EventIdentifier	0.99	1.00	**0.51**	< **0.001**
de.huxhorn.lilith.debug.LogDateRunnable	0.60	0.60	0.50	-
de.huxhorn.lilith.engine.impl.eventproducer.SerializingMessageBasedEventProducer	0.99	1.00	0.50	0.316
de.outstare.fortbattleplayer.gui.battlefield.BattlefieldCell	0.17	0.21	**0.64**	< **0.001**
de.outstare.fortbattleplayer.statistics.CriticalHit	1.00	1.00	0.50	-
de.paragon.explorer.util.LoggerFactory	1.00	1.00	0.50	-
de.progra.charting.render.InterpolationChartRenderer	0.12	0.55	**0.96**	< **0.001**
edu.uiuc.ndiipp.hubandspoke.workflow.PackageDissemination	0.02	0.09	**0.99**	< **0.001**
falselight	1.00	1.00	0.50	-
fi.vtt.noen.mfw.bundle.common.DataType	1.00	1.00	0.50	-
fi.vtt.noen.mfw.bundle.probe.plugins.measurement.WatchDog	0.03	0.31	**0.86**	< **0.001**
fi.vtt.noen.mfw.bundle.probe.shared.MeasurementReport	0.09	1.00	**0.99**	< **0.001**
fi.vtt.noen.mfw.bundle.server.plugins.webui.sacservice.OperationResult	1.00	1.00	0.50	-
fps370.MouseMoveBehavior	0.19	0.55	**0.98**	< **0.001**
geo.google.mapping.AddressToUsAddressFunctor	0.04	0.52	**0.98**	< **0.001**
httpanalyzer.ScreenInputFilter	0.73	0.83	**0.64**	< **0.001**
jigl.gui.SignalCanvas	0.85	0.95	**0.89**	< **0.001**
jigl.image.io.ImageOutputStreamJAI	0.21	0.54	**0.94**	< **0.001**
jigl.image.utils.LocalDifferentialGeometry	0.04	0.43	**0.99**	< **0.001**
lotus.core.phases.Phase	0.50	0.50	0.50	-
macaw.presentationLayer.CategoryStateEditor	0.00	0.00	0.50	-
messages.round.RoundTimeOverMsg	0.99	1.00	**0.50**	**0.007**
module.ModuleBrowserDialog	0.00	0.00	0.50	-
net.sf.xbus.base.bytearraylist.ByteArrayConverterAS400	0.00	0.00	0.50	-
net.sourceforge.beanbin.command.RemoveEntity	1.00	1.00	0.50	-
net.virtualinfinity.atrobots.robot.RobotScoreKeeper	1.00	1.00	0.50	-
nu.staldal.lagoon.util.Wildcard	0.99	1.00	**0.50**	< **0.001**
oasis.names.tc.ciq.xsdschema.xal._2.PremiseNumberSuffix	1.00	1.00	0.50	-
org.apache.lucene.search.exposed.facet.FacetMapSinglePackedFactory	0.00	0.18	**0.99**	< **0.001**
org.databene.jdbacl.dialect.H2Util	1.00	0.99	**0.49**	< **0.001**
org.databene.jdbacl.identity.mem.AbstractTableMapper	0.20	0.71	**0.99**	< **0.001**
org.dom4j.io.STAXEventReader	0.14	0.28	**0.99**	< **0.001**
org.dom4j.tree.CloneHelper	1.00	1.00	0.50	-
org.dom4j.util.PerThreadSingleton	0.85	0.85	0.49	0.165
org.exolab.jms.config.GarbageCollectionConfigurationLowWaterThresholdType	1.00	1.00	0.50	-
org.exolab.jms.config.SecurityConfigurationDescriptor	0.62	0.62	**0.47**	< **0.001**
org.exolab.jms.selector.And	0.87	0.99	**0.77**	< **0.001**
org.exolab.jms.selector.BetweenExpression	0.33	0.75	**0.94**	< **0.001**
org.fixsuite.message.view.ListView	0.10	0.10	0.50	0.312
org.jcvi.jillion.assembly.consed.phd.PhdFileDataStoreBuilder	0.43	0.83	**0.99**	< **0.001**
org.jcvi.jillion.fasta.FastaRecordDataStoreAdapter	0.00	0.00	0.50	-
org.jsecurity.authc.credential.Md2CredentialsMatcher	1.00	1.00	0.50	-
org.jsecurity.io.IniResource	0.40	0.82	**0.99**	< **0.001**
org.jsecurity.io.ResourceUtils	0.32	0.79	**0.99**	< **0.001**
org.jsecurity.web.DefaultWebSecurityManager	0.07	0.37	**0.99**	< **0.001**
org.quickserver.net.qsadmin.gui.SimpleCommandSet	0.83	0.83	0.50	-
org.quickserver.net.server.AuthStatus	0.33	0.33	0.50	-
org.sourceforge.ifx.framework.complextype.ChkAcceptAddRs_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.ChkInfo_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.ChkOrdInqRs_Type	0.99	1.00	0.50	0.080
org.sourceforge.ifx.framework.complextype.CreditAdviseRs_Type	0.99	1.00	0.50	0.315
org.sourceforge.ifx.framework.complextype.DepAcctStmtInqRq_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.EMVCardAdviseRs_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.ForExDealMsgRec_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.PassbkItemInqRs_Type	0.99	1.00	0.50	0.312
org.sourceforge.ifx.framework.complextype.RecPmtCanRq_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.StdPayeeId_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.SvcAcctStatus_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.complextype.TINInfo_Type	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.AllocateAllowed	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.BillInqRs	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.ChksumModRq	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.ChksumStatusCode	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.CompositeCurAmtId	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.CurAmt	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.CustAddRs	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.CustId	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.CustPayeeRec	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.DepBkOrdAddRs	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.DevCimTransport	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.FSPayee	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.Gender	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.Language	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.StdPayeeRevRs	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.TerminalSPObjAdviseRq	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.element.URL	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.pain001.simpletype.BatchBookingIndicator	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.pain001.simpletype.CashClearingSystem2Code	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.pain004.simpletype.CashClearingSystem2Code	1.00	1.00	0.50	-
org.sourceforge.ifx.framework.simpletype.DevName_Type	1.00	1.00	0.50	-
teder.Teder	1.00	1.00	0.50	-
umd.cs.shop.JSListSubstitution	0.97	0.99	**0.53**	< **0.001**
wheel.components.Block	0.04	0.16	**0.56**	< **0.001**
wheel.json.JSONStringer	0.99	1.00	**0.50**	< **0.001**
Average	0.67	0.76	0.62	

```
class RecordingEvent {
  static protected DataFlavor ourJVMLocalObjectFlavor;
  static protected DataFlavor[] ourFlavors;

  static protected void initializeConstants() {
    if (null != ourJVMLocalObjectFlavor || null !=
        ourFlavors)
      return;

    ourJVMLocalObjectFlavor = ...
    ourFlavors = new DataFlavor[] {
        ourJVMLocalObjectFlavor };
    // ...
  }
  // ...
}
```

Fig. 1. Static behavior in `RecordingEvent`: If the test independence assumption is not satisfied, then results become unpredictable.

variables `ourJVMLocalObjectFlavor` and `ourFlavors`. As EvoSuite works at the level of bytecode, this disjunction results in four branches – two for each of the conditions. The *Whole* approach only succeeds in covering one out of these four branches, i.e., when `outJVMLocalObjectFlavor` is non-null and the return statement is taken. This is because in the default configuration of EvoSuite the static state of a class is not reset, and so once the `initializeConstants` method has been executed, the two static variables are non-null. In the case of *OneBranch*, if the first chosen coverage goal is to make either of the two conditions false, then this will be covered in the first test executed by EvoSuite, and thus the two true-branches will have a covering test. If, however, `initializeConstants` is executed as part of the search for any other branch, then the coverage will be the same as for *Whole*. This is a known effect of static states, and so EvoSuite has an experimental feature to reset static states after test execution. When this feature is enabled, then both *Whole* and *OneBranch* succeed in covering three out of the four branches. (To cover the fourth branch, the assignment to `ourJVMLocalObjectFlavor` would need to throw an exception such that only one of the two variables is initialized). However, even when static state is reset, the overall coverage achieved by *Whole* is significantly lower than for *OneBranch*. The "difficult" branches are cases of a `switch` statement, and branches inside a loop over array elements. These branches are sometimes covered by *Whole*, but less reliably so than by *OneBranch*.

`BlockThread` only has a single conditional branch, all other methods contain just sequences of statements (in EvoSuite, a method without conditional statements is counted as a single branch, based on the control flow graph interpretation). However, the class spawns a new thread, and several of the methods

Table 2. For each branch, we report how often the *Whole* approach is better (higher effect size) than the *OneBranch*, when they are equivalent, and when it is *OneBranch* that is better. We also report the number of comparisons that are statistically significant at 0.05 level, and when only one of the two techniques ever managed to cover a goal out of the 1,000 repeated experiments.

	# of Branches	Statistically at 0.05	Never Covered by the Other
Whole Approach is better:	1631	1402	246
Equivalent:	671	–	–
OneBranch is better:	81	58	3
Total:	2383		

synchronize on this thread (e.g., by calling `wait()` on the thread). EvoSuite uses a timeout of five seconds for each test execution, and any test case or test suite that contains a timeout is assigned the maximum (worst) fitness value, and not considered as a valid solution in the final coverage analysis. In `BlockThread`, many tests lead to such timeouts, and a possible conjecture for the worse performance of the *Whole* approach may be that the chances of having an individual test case without timeout are simply higher than the chances of having an entire test *suite* without timeouts.

To study the difference between *OneBranch* and *Whole* at a finer grained level, Table 2 shows on how many coverage goals (i.e., branches) one technique is better than the other. There are 58 cases in which *OneBranch* led to better results. Three of them, *Whole* never manages to cover.

RQ1*: There are 58 coverage goals in which OneBranch obtained better results. Three of them were never covered by Whole.*

On the other hand, there are 1,402 cases (out of 2,383) in which *Whole* gives better results. For 246 of them, the *OneBranch* approach *never* managed to generate any results in any of the 1,000 runs. In other words, even if there are some (i.e., three) difficult goals that only *OneBranch* can cover, there are many more (246/3 = 82 times) difficult branches that only *Whole* does cover.

RQ2*: Whole test suite generation is able to handle 82 times more difficult branches than OneBranch.*

Once assessed that the *Whole* approach leads to higher coverage, even for the difficult branches, it is important to study what are the conditions in which this improvement is obtained. For each coverage goal (2,383 in total), we calculated the odds ratio between *Whole* and *OneBranch* (i.e., we quantified what are the odds that *Whole* has higher chances to cover the goal compared to *OneBranch*). For each odds ratio, we studied its correlation with three different properties: (1) the $\hat{A}_{12}$ effect size between *Whole* and *OneBranch* on the class the goal belongs to; (2) the raw average branch coverage obtained by *OneBranch* on the class the

Table 3. Correlation analyses between the odds ratios for each coverage goal and three different properties. For each analysis, we report the obtained correlation value, its confidence interval at 0.05 level and the obtained p-value (of the test whether the correlation is different from zero).

Property	Correlation	Confidence Interval	p-value
$\hat{A}_{12}$ Whole vs. OneBranch	0.275	[0.238, 0.312]	< 0.001
OneBranch Coverage	0.016	[-0.024, 0.056]	0.433
# of Branches	0.051	[0.011, 0.091]	0.012

goal belongs to; and, finally, (3) the size of the class, measured as number of branches in it. Table 3 shows the results of these correlation analyses.

There is correlation between the odds ratios and the $\hat{A}_{12}$ effect sizes. This is expected: on a class in which the *Whole* approach obtains higher coverage on average, then it is more likely that on each branch in isolation it will have higher chances to cover those branches. However, this correlation is weak, at only 27%.

On classes with many infeasible branches (or too difficult to cover for both *Whole* and *OneBranch*), one could expect higher results for *Whole* (as it is not negatively affected by infeasible branches [11]). It is not possible to determine if branches are feasible or not. However, we can somehow quantify the difficulty of a class by the obtained branch coverage. Furthermore, one would expect better results of the *Whole* approach on larger, more complex classes. But the results in Table 3 show no significant correlation of the odds ratios with the obtained average branch coverage, and only very small (just 5%) with the class size. In other words, the fact that *Whole* approach has higher chances of covering a particular goal seems irrelevant from the overall coverage obtained on such class and its size.

The analysis presented in Table 3 numerically quantifies the correlations between the odds ratios and the different studied properties. To study them in more details, we present scatter plots: Figure 2 for the $\hat{A}_{12}$ effect sizes, Figure 3 for the *OneBranch* average coverage and, finally Figure 4 for class sizes.

Figure 2 is in line with the 27% correlation value shown in Table 3. There are two main clusters, where low odds ratios lead to low $\hat{A}_{12}$ effect sizes, and the other way round for high values. There is also a further cluster of values around $\hat{A}_{12} = 0.5$ for which higher odds are obtained.

Although there is no clear correlation between the odds ratios and the obtained coverage of *OneBranch* (only 1% in Table 3), Figure 3 shows an interesting trend: the only coverage goals for which *Whole* perform worse (i.e., logarithms of the odds ratios are lower than zero) are in classes for which *OneBranch* obtains high coverage (mostly close to 100%). This is visible in the top-left corner in Figure 3. There are coverage goals for which *Whole* approach has much higher odds (logarithms above 30), and those appear only in classes for which the *OneBranch* approach obtains an overall low branch coverage (see the rightmost values in Figure 3).

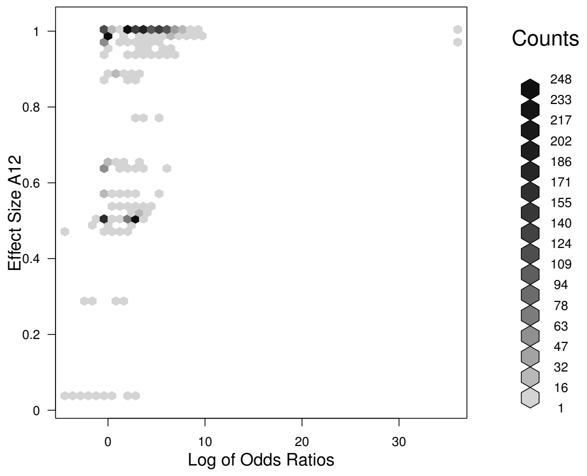

Fig. 2. Scatter plot of the (logarithm of) odds ratios compared to the $\hat{A}_{12}$ effect sizes.

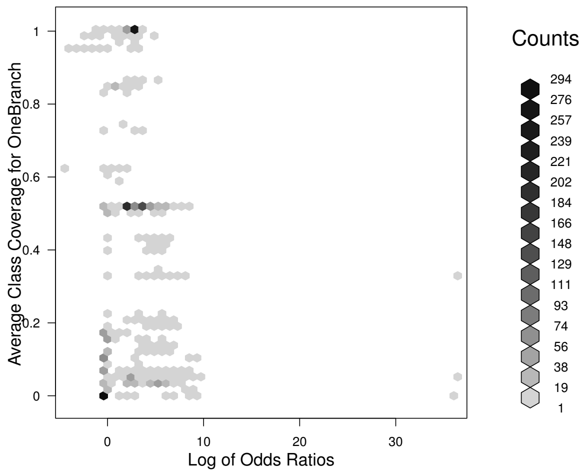

Fig. 3. Scatter plot of the (logarithm of) odds ratios compared to average class coverage obtained by *OneBranch*.

When looking at the effects of size, in Figure 4 we can see that the only cases in which *OneBranch* has better odds ratios are when the SUTs are small. This is visible in the bottom-left corner of Figure 4.

RQ3: *Our data does not point to a factor that strongly influences the relative performance between Whole and OneBranch.*

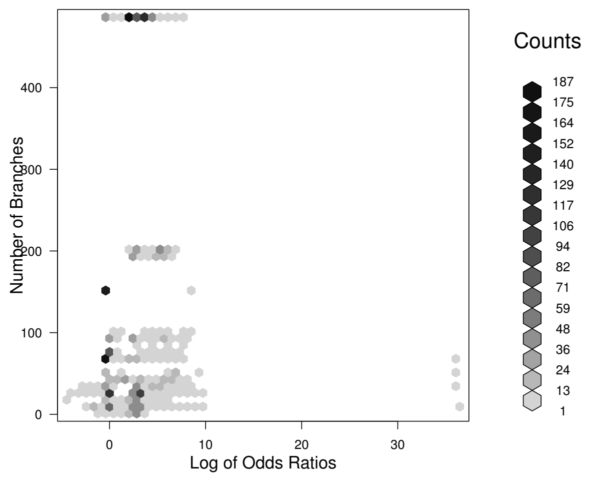

Fig. 4. Scatter plot of the (logarithm of) odds ratios compared to the sizes of the classes.

Interestingly, the few cases in which *OneBranch* obtains better results seem located in small classes in which both approaches obtain relatively high code coverage.

5 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 1,000 times, and we followed rigorous statistical procedures to evaluate their results. For the comparisons between the *Whole* approach and the *OneBranch* approach, both were implemented in the same tool (i.e., EvoSuite) to avoid possible confounding factors when different tools are used.

There is the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Because of the large number of experiments required (in the order of hundreds of days of computational resources), we only used 100 classes for our in depth evaluations. These classes were randomly chosen from the SF100 corpus, which is a random selection of 100 projects from SourceForge. We only experimented for branch coverage and Java software. Whether our results do generalise to other programming languages and testing criteria is a matter of future research.

6 Conclusions

Existing research has shown that the whole test suite approach can lead to higher code coverage [11,12]. However, there was a reasonable doubt on whether it would still perform better on particularly difficult coverage goals when compared to a more focused approach.

To shed light on this potential issue, in this paper we performed an in-depth analysis to study if such cases do indeed occur in practice. Based on a random selection of 100 Java classes in which we aim at automating test generation for branch coverage with the EvoSuite tool, we found out that there are indeed coverage goals for which the whole test suite approach leads to worse results. However, these cases are very few compared to the cases in which better results are obtained (nearly two orders of magnitude in difference), and they are also located in less "interesting" classes: i.e., small classes for which both approaches can already achieve relatively high code coverage.

The results presented in this paper provides more support to the validity and usefulness of the whole test suite approach in the context of test data generation. Whether such an approach could be successfully adapted also to other search-based software engineering problems will be a matter of future research.

To learn more about EvoSuite, visit our website at:

`http://www.evosuite.org/study`

Acknowledgments. This project has been funded by the EPSRC project "EX-OGEN" (EP/K030353/1), a Google Focused Research Award on "Test Amplification", and the Norwegian Research Council.

References

1. S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)*, 36(6):742–762, 2010.
2. A. Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering (TSE)*, 38(3):497–519, 2012.
3. A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability (STVR)*, 23(2):119–147, 2013.
4. A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability (STVR)*, 2012. DOI: 10.1002/stvr.1486.
5. A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):258–277, 2012.
6. A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Inform. Sciences*, 178(15):3075–3095, 2008.

7. L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.
8. G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.
9. G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.
10. G. Fraser and A. Arcuri. Handling test length bloat. *Software Testing, Verification and Reliability (STVR)*, 2013. DOI:10.1002/stvr.1495.
11. G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.
12. G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering (EMSE)*, 2014. To appear.
13. G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.
14. M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *International Workshop on Search-Based Software Testing (SBST)*, 2010.
15. M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.
16. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Technical Report TR-09-06, CREST Centre, King's College London, London, UK, September 2009.
17. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.
18. K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for c programs using cute and austin. *J. Syst. Softw.*, 83(12), Dec. 2010.
19. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
20. W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.
21. J. C. B. Ribeiro. Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1819–1822. ACM, 2008.
22. P. Tonella. Evolutionary testing of classes. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.
23. S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1053–1060. ACM, 2005.
24. J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.