UNIT-TEST SOURCE CODE IS A GOOD APPROXIMATION FOR ABSTRACT STATES

Mario Gonzalez Macedo

Ortask
Next-Generation Test Management
mgm@ortask.com

ABSTRACT

This paper introduces a mathematical model for a form of test redundancy called *duplication*, and studies the relationship between unit-test source code and *abstract states*, a concept utilized by the model for performing test comparison. In particular, it presents evidence that unit-test source code can be used verbatim to effectively compare tests without having to modify (i.e. instrument) neither the system under test nor to the tests themselves. Thus, the model is utilized to study the feasibility of quickly comparing unit-tests (and *only* the tests) to detect test duplication. It then presents results of running an implementation of a comparison engine based on the model against existing suites containing redundant unit-tests. Finally, the results are discussed to provide objective considerations regarding the practicality and usefulness of such an approach to unit-test comparison.

1. INTRODUCTION

The goal of software testing is to provide convincing evidence for the correct functionality of tested systems. Since software development and testing are engineering practices, one of their main goals is to achieve the highest levels of return with the lowest risk and cost. However, redundant tests of any kind, including unit-tests, represent extra work and wasted time that increase the cost of testing and development. As such, redundant tests should be minimized or removed to achieve the engineering optimum.

As more tests are written to accommodate a growing software system, the chances of introducing redundant unit-tests grows as well. The likelihood for redundancy and the associated cost grows all the more when multiple people write tests for the same functionality, as in open-source projects.

Yet test comparison has thus far been solely an intuition-based manual process that is applied on a case-by-case basis. That is, the most widely used process to detect redundant tests is by people specifically studying pairs of tests which are then deemed, either by using domain knowledge or a hunch, as redundant. Nevertheless, this process is impractical and highly unreliable due to its subjectivity, which stems from disagreement between software testing practitioners about what constitutes a redundant test. This, in turn, originates from a lack of a solid, universal mathematical foundation for such a concept.

Other proposed approaches to test comparison such as [1, 2, 3, 4, 5] are simply not practical because they run counter to the optimization goal of engineering. For instance, [1] requires multiple phases of time-consuming processing in the form of runtime profiling, static code analysis, and execution of machine learning algorithms. The execution of tests alone required for the runtime analysis is already time-consuming for [1, 2, 3, 4, 5], yet they all have the unfortunate extra requirement of having to execute the tests against modified (i.e. instrumented) versions of the SUT[1] for the plethora of analyses to be remotely effective. Moreover, some approaches such as [2] also require instrumenting the test code prior to analysis.

What is desired, therefore, is an accurate, yet quick, approach to test comparison that does not necessitate instrumenting neither the SUT nor the tests themselves. Further, the desired approach should be *offline*, which means that it does not require running the tests at all. In other words, test comparison should be a process that is aligned, rather than orthogonal to, the development and testing process already in place. Otherwise, we would increase the cost of testing and development, which is contrary to the engineering goal.

To solve these problems, this paper introduces a mathematical model for test redundancy. The model captures the intuitive understanding of what test duplicates are in a way that is consistent and applicable across many different types of tests, from unit to manual, from system-level (i.e. end-to-end) to security, and even to exploratory tests. However, in this paper we focus our attention only on unit-tests[2].

There are two types of redundant tests. The first type is usually called test *duplication* by software engineer practitioners. Loosely speaking, *duplicate* tests are tests that "do" exactly what previous tests already do, albeit in possibly different ways. Such duplicate tests are clearly redundant and can thus be removed from their test suites since they provide no new coverage and, hence, no value. This form of redundancy is called *test equivalence* in the model.

Another form of test redundancy occurs when new tests are not exactly duplicates, yet what they "do" overlaps with other tests. That is, tests that cover part of the functionality that other tests attempt to verify also represent redundant testing. As such, these redundant tests do not necessarily lead to software systems with higher quality (i.e. better tested) yet they require work to write and time to run. Such form of redundancy is called *test overlap* or *test coverage* in the model[3].

This paper focuses on the first form of test redundancy called test equivalence[4] and provides evidence for the claim that source code of unit-tests can be used as-is to perform accurate comparison of tests[5]. The motivation is two-fold: 1) to be able to detect test equivalence by quickly comparing unit-tests verbatim without ne-

[1] System Under Test.

[2] Subsequent papers will study the rest of the test types.

[3] Not to be confused with *code* coverage.

[4] The second form of test redundancy, that of test coverage, will be discussed in subsequent papers as it is a very deep and interesting subject in its own right.

[5] That is, lines of source code from unit-tests are a good approximation for modeling abstract states, a new concept that will be explained below.

cessitating modification to either the tests or the SUT, thus making test comparison an exercise that optimizes our development and testing practices; and 2) to be able to extend the area of test management to real-world problems that are not addressed by typical test management tools[6].

2. MATHEMATICAL MODEL

We begin by establishing the foundation upon which the rest of the model is based: the concept of an *abstract state*.

2.1. Definition: Abstract State

An abstract state is a set of properties about the SUT that are expected to be true at a specific point in a test. Further, an abstract state is only composed of properties that are *relevant* at that particular point of the test.

All abstract states are members of the class $ABSTRACT\_STATE_{SUT}$, which is the entire collection of sets of properties that the given SUT is expected to have.

Finally, an abstract state represents a collection and an abstraction of one or more underlying (i.e. real) states of the SUT.

Note that by the definition above, no irrelevant property forms part of an abstract state. For instance, time is always a part of real system states. However, unless time is relevant to a particular point in a test, the associated abstract state will not contain any property related to time. Due to this "casting away" of irrelevant details, abstract states can represent one or more real states from the SUT.

Recall that the argument put forth by this paper is that the code from unit-tests is a close approximation to abstract states. In other words, the nature of the high-level statements utilized in unit-tests, which are expressed in a rigorous and consistent language for its corresponding virtual machine, automatically ignores many irrelevant details. As such, the claim in this paper is that unit-tests can be utilized as-is to perform unit-test comparison without the need to instrument/modify the SUT or the tests themselves.

Next, we formalize the concept of a *test*[7]. Although not central to the point of this paper (i.e. test duplication), it provides the necessary context for the rest of the framework.

2.2. Definition: Test Case

A test case is a 3-tuple (α, σ, ω) where

- α is a non-empty abstract state called the *alpha-set* and contains the *pre-conditions* of the test.
- σ is a finite, non-empty sequence called the *sigma-sequence* and contains the *steps* of the test, in order.
- ω is a non-empty abstract state called the *omega-set* and contains the *post-conditions* of the test.

When convenient, we will utilize the function notation to express the relationship between a specific test and its components. For example, $\omega(t)$ denotes ω of test *t*.

[6]By *typical* it is meant the common-place test management tools composed of a database plus a reporting engine, with no support for management of redundant tests.

[7]Interchangeably called *test case* in the literature and in this paper.

Every test has a *model*. Each model is a sequence of abstract states. In turn, each abstract state is related to a step in σ of its corresponding test. The following definition formalizes this concept.

2.3. Definition: Model of a test

Let $t \in TESTS$ be a test case in a suite of tests. We call *M(t)* the *model* of the SUT for test case *t* (or just "the model of test case *t*) and define it as the finite sequence of abstract states induced by $\sigma(t)$ as follows:

$$M(t) = < \alpha, q_2, q_3, \ldots, q_{|\sigma|}, \omega >$$

Before our next definition, we need a useful concept. In particular, let $t \in TESTS$ be a test and let $M(t)^{\{\}}$ be the set that results by collecting all elements of its model, $M(t)$. Now we're ready to define the *core* of a test.

2.4. Definition: Core of a test

Let $t \in TESTS$ be a test in a suite of tests. Let $M(t)^{\{\}}/\sim$ denote the quotient set of $M(t)^{\{\}}$ and define it as the set of all possible equivalence classes of $M(t)^{\{\}}$ induced by the equivalence relation $\sim$.

Then define the *core* of test *t* as:

$$core(t) = M(t)^{\{\}}/\sim = \{[e]_\sim \mid e \in M(t)^{\{\}}\}$$

Let $i : M \rightarrow \aleph^+$ be a function that maps an abstract state from a model *M* to a positive integer. The image of *i* represents the *position* of the given abstract state in *M*.

2.5. Definition: the natural abstract state ordering relation

Let *M* be a model associated to some arbitrary test. Then, the relation $\preceq$ applies a total order such that

$$\forall q, p \in M : q \preceq p \leftrightarrow i(q) \leq i(p)$$

The *intent* of a test can be described by the positions of abstract states in its model (as arranged by $\preceq$) and its core. We now define the basic form of test redundancy called *coverage* (*overlap*).

2.6. Definition: Coverage (overlap) of tests

For any two tests $v, t \in TESTS$, we say that *t covers v*, written $t \lhd v$, if and only if there is a homomorphism $h : core(v) \rightarrow core(t)$ such that:

$$\forall c, c' \in core(v) : c \preceq c' \rightarrow h(c) \preceq h(c')$$

For this paper, we will use a more generalized notion of coverage called *threshold-coverage*. This concept uses something called a *reduced* core which allows the overlap to be arbitrarily strict or lenient.

2.7. Definition: Reduced core

Let $v, t \in TESTS$ be two tests in a suite and let *core(t)* and *core(v)* be their cores. Also, let $tcmp \in \Re$ be a real in the range [0,1]. Then the *reduced* core of *t* (with respect to *v*), written $red(t)_v$, is a subset of *core(t)* and is defined as:

$$red(t)_v = \{e \mid e \in core(t) \wedge \exists e' \in core(v) : e \sim e'\}$$

such that

$$\frac{|red(t)_v|}{|core(t)|} \geq tcmp$$

where $\sim$ is an equivalence relation.

Notice that if the reduced core, $red(t)_v$, of a test *t* does not exist for a particular threshold, $tcmp$, then the homomorphism does not exist. This, in turn, implies that no overlap exists between tests *t* and *v*.

We now define *threshold-coverage*.

2.8. Definition: Threshold-Coverage of tests

For any two tests $v, t \in TESTS$, we say that *t covers v*, written $t \lhd v$, if and only if there is a homomorphism $h : red(v)_t \rightarrow core(t)$ such that:

$$\forall c, c' \in red(v)_t : c \preceq c' \rightarrow h(c) \preceq h(c')$$

Note that we have preserved the symbols for threshold-coverage the same as for coverage above. This is because definition **2.6** can be seen as satisfying $tcmp = 1$.

We can now define the second form of test redundancy called *equivalence* (*duplication*), which is the focus of this paper.

2.9. Definition: Equivalence (duplication) of tests

For any two tests $v, t \in TESTS$, we say that *t* is *equivalent* to *v*, written $t \equiv v$, if and only if $t \lhd v$ and $v \lhd t$

3. EXPERIMENTAL RESULTS

3.1. Preliminaries

For the equivalence relation in **2.7**, a pseudo-metric is used, which also ranges from 0 to 1. We denote by t the value used for the pseudo-metric.

The values for $tcmp$ and t are varied independently during the testing of the comparison engine on all experiment sets. Varying these values allows experimenting with different sensitivities and precisions for the engine. In turn, this allows for more thorough experimentation and, thus, understanding of how the engine (and, ultimately, the claim of this paper) performs on different unit-test suites with respect to size and language.

To compare the results between the experiment sets, several standard scores are used. *Precision* is the probability that a test that is selected by the comparison engine is indeed a duplicate. Higher values for precision imply lower false positives. It is defined as

$$Precision = \frac{true\ positives}{true\ positives\ +\ false\ positives} \quad (1)$$

Recall is the average probability that a duplicate test is found. In other words, higher values for recall imply lower false negatives. It is defined as

$$Recall = \frac{true\ positives}{true\ positives\ +\ false\ negatives} \quad (2)$$

The *F1-measure* is a value from 0 to 1 that is utilized to assess a balanced accuracy of a classifier. A value of 1 represents the ideal score. It is defined as

$$F1{-}measure = 2\ \times\ \frac{precision\ \times\ recall}{precision\ +\ recall} \quad (3)$$

A slightly better quality measure is the *Matthews Correlation Coefficient* (*MCC*), and has a value in the range [-1,1]. A value of 1 is ideal; a value of 0 implies random selection; while a value of -1 implies results that are opposite to the true classification. *MCC* overcomes some limitations of the *F1-measure* and is not sensitive to the size of the classes. As such, *MCC* is a better measurement for when the amount of duplicate tests is much smaller than the total number of tests in a suite. *MCC* is defined as

$$MCC = \frac{tp\ \times\ tn -\ fp\ \times\ fn}{\sqrt{(tp\ +\ fp)(tp\ +\ fn)(tn\ +\ fp)(tn\ +\ fn)}} \quad (4)$$

where *tp* stands for *true positives*, et cetera.

Finally, to assess how the comparison engine is affected by the size of the suite as well as the size of the tests, the average test size (lines of code) is computed and presented per suite. Let the function $size(t)$ denote the quantity of lines of code for test *t*. The average test size measure is calculated as

$$Average\ test\ size = \frac{\sum_{t \in TESTS}\ size(t)}{|TESTS|} \quad (5)$$

3.2. Assumptions

For this paper, two important assumptions are made about test suites: 1) the amount of duplicate tests in the typical suite is much smaller than the total size of the suite; 2) duplicate tests are non-uniformly distributed throughout the suite.

Due to assumption 1, it is expected that *MCC* give slightly higher scores than *F1-measure*. Further, due to assumption 2, it is expected that the performance of the comparison engine does not always exhibit the typical (uniform) precision-recall trade-off seen in many classifier systems.

3.3. Experiment Set 1

First, the comparison engine is run on two test suites that were originally implemented by the author to verify its functionality. These suites are utilized because they were developed with the sole purpose of testing the implementation of the components that make up the engine, and so represent actual, rather than artificial, test suites.

The suites are implemented using the Ruby programming language. The names of the suites in this experiment set are "TestAbstractState" and "TestTrace". The first suite, "TestAbstractState", contains 27 tests, yielding 351 pairs to verify and an average test size (lines of code) of 6.18. The second suite, "TestTrace", contains 14 tests, yielding 91 pairs to verify and an average test size of

10.85. In terms of processing time, the engine takes only several minutes on both suites.

Figure 1 shows the best performance scores of the engine on test suite "TestAbstractState". The figure also shows the settings for *t* and *tcmp* that yield the associated scores. Likewise, Figure 2 shows the best performance scores of the engine on suite "TestTrace", along with the associated engine settings.

Figure 3 shows the overall performance of the engine on suite "TestAbstractState". Figure 4 shows the performance curves of the engine on "TestTrace". Notice how both sets of curves exhibit a "dip" at the very end, on the right-hand side of the figures. This is because the equivalence relation becomes equality as *t* increases. Since there are no identical duplicate tests in the suites and the engine is not able to find such tests, the true positive mark goes down to 0, thus explaining the apparent dip in performance. As such, the dip is expected when suites contain no identical copies of tests and the sensitivity of the engine is very high.

Figure 5 shows two tests that are correctly identified as duplicates by the engine for suite "TestAbstractState." Further, figure 6 shows two duplicate tests that the engine correctly identifies for suite "TestTrace."

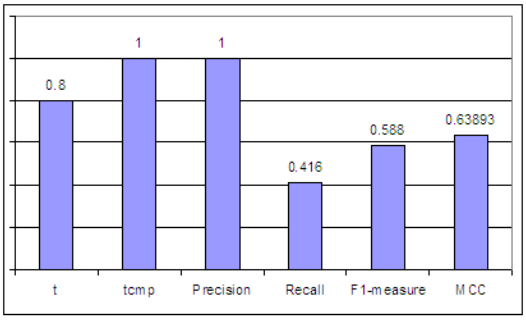

Figure 1: *Best scores for the engine on TestAbstractState*

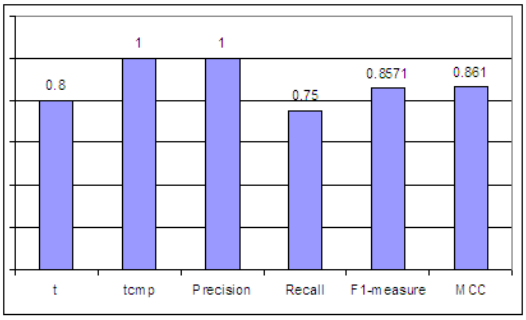

Figure 2: *Best scores for the engine on TestTrace*

3.4. Experiment Set 2

For this experiment set, the five Java-based sample tests presented in [5] are used verbatim. They are presented in this paper again for convenience.

```
Test 1 (T1):
   IntStack s1 = new IntStack();
   s1.isEmpty();
   s1.push(3);
   s1.push(2);
   s1.pop();
   s1.push(5);

Test 2 (T2):
   IntStack s2 = new IntStack();
   s1.push(3);
   s1.push(5);

Test 3 (T3):
   IntStack s3 = new IntStack();
   s1.push(3);
   s1.push(2);
   s1.pop();

Test 4 (T4):
   IntStack t = new IntStack();
   IntStack s4 = t;
   for (int i = 0; i <= 1; i++)
      s4.push(i);

Test 5 (T5):
   IntStack s5 = new IntStack();
   int i = 0;
   s5.push(i);
   s5.push(i + 1);
```

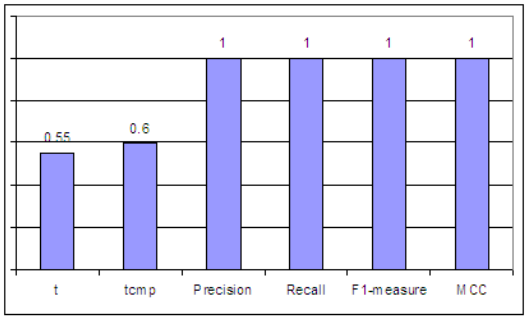

Figure 7: *Best scores for the engine on experiment set 2*

The average test size for this suite is 4.2. Moreover, the engine requires only several seconds to process the suite.

The comparison engine correctly identifies T5 and T4 as duplicates. It also correctly identifies T2 as a duplicate of both T4 and T5. The reason for this is that all three tests merely push integers on their IntStacks, so any two of them can be removed to provide the same testing value.

Note that even though [5] considers T3 a duplicate of T1 when using some of their more sophisticated techniques, our comparison engine finds them to be distinct. This is because T1 covers T3 but T3 does *not* cover T1. In particular, T1 queries the state of its IntStack, while T3 does not. Since T1 contains steps that force the SUT to go through different states (i.e. execute code paths that T3 does not induce), the intents of the two tests are clearly distinct. Further, since the overlap is only one-way, the engine correctly finds the two tests to be distinct.

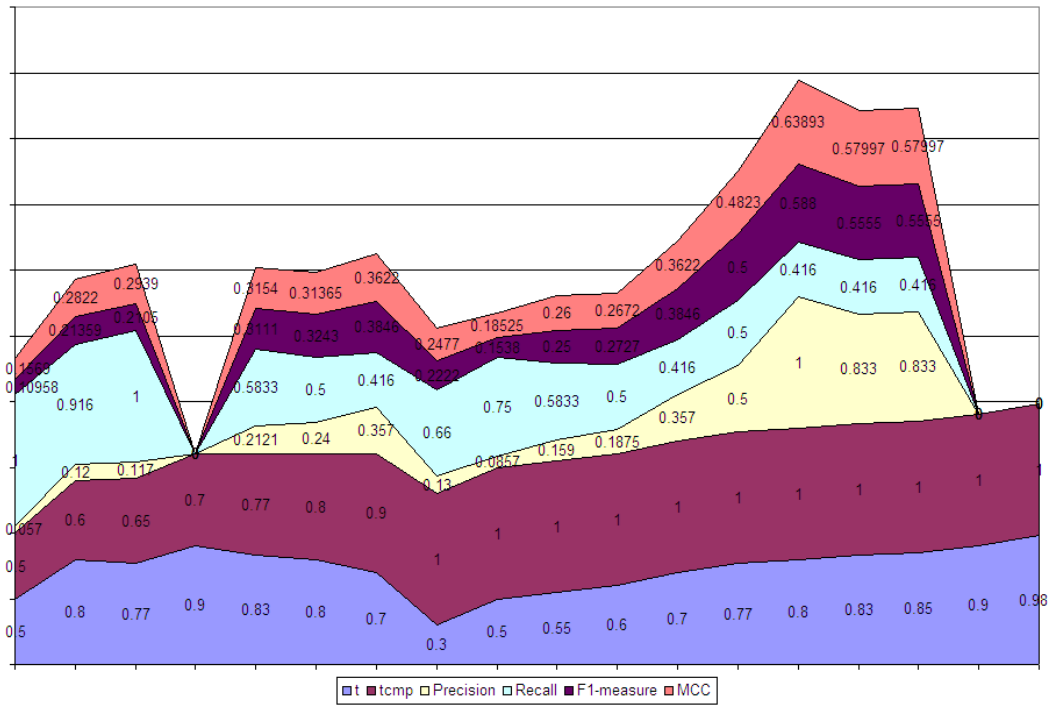

Figure 3: *Performance curves for the engine on TestAbstractState*

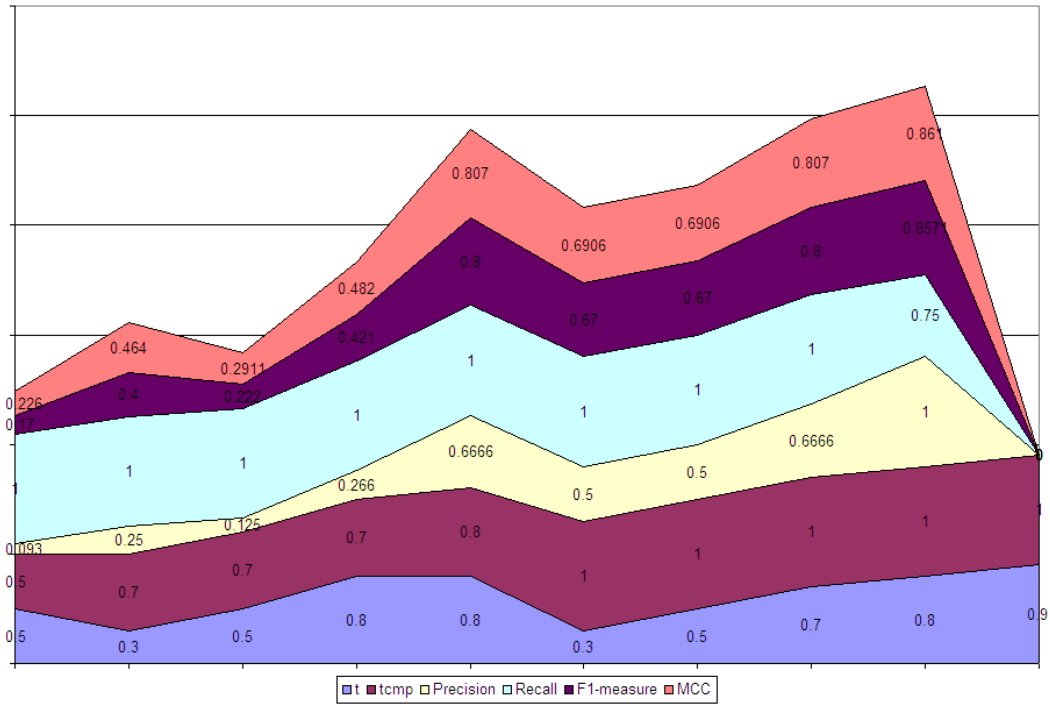

Figure 4: *Performance curves for the engine on TestTrace*

```
def testSetIntersectionEmptyResult2
    set1 = Set.new
    set2 = Set.new
    set1 << AbstractState.new(["hola", "como", "estas", "?"])
    set2 << AbstractState.new(["hola", "como", "estas"])
    actual = (set2 & set1).to_a
    expected = []
    assert( expected == actual, "Expecting '#{expected}' but got '#{actual}'" )
    actual = (set1 & set2).to_a
    expected = []
    assert( expected == actual, "Expecting '#{expected}' but got '#{actual}'" )
end

def testSetIntersectionEmptyResult3
    set1 = Set.new
    set2 = Set.new
    set1 << AbstractState.new(["1", "2", "3", "4"])
    set2 << AbstractState.new(["1", "2", "3"])
    actual = (set2 & set1).to_a
    expected = []
    assert( expected == actual, "Expecting '#{expected}' but got '#{actual}'" )
    actual = (set1 & set2).to_a
    expected = []
    assert( expected == actual, "Expecting '#{expected}' but got '#{actual}'" )
end
```

Figure 5: *Two correctly identified duplicate tests in suite TestAbstractState*

```
def testWithNonEmptyInferredModel_TwoElementsDifferent()
    inferredModel = Model.new()
    inferredModel << AbstractState.getEmpty()
    inferredModel << AbstractState.new(["hola!"])
    actual = Trace.inferTrace( inferredModel, debug=false )
    expected = Trace.new(debug=false)
    expected << EquivalenceClass.getEmpty()
    expected << EquivalenceClass.new(AbstractState.new(["hola!"]))
    assert( expected == actual, "Expecting '#{expected}' but got '#{actual}'" )
end

def testWithNonEmptyInferredModel_SomeElementsDifferent()
    inferredModel = Model.new()
    inferredModel << AbstractState.getEmpty()
    inferredModel << AbstractState.new(["hola!"])
    inferredModel << AbstractState.new(["hola!"])
    inferredModel << AbstractState.new(["alo!"])
    actual = Trace.inferTrace( inferredModel, debug=false )
    expected = Trace.new(debug=false)
    expected << EquivalenceClass.getEmpty()
    expected << EquivalenceClass.new(AbstractState.new(["hola!"]))
    expected << EquivalenceClass.new(AbstractState.new(["hola!"]))
    expected << EquivalenceClass.new(AbstractState.new(["alo!"]))
    assert( expected == actual, "Expecting '#{expected}' but got '#{actual}'" )
end
```

Figure 6: *Two correctly identified duplicate tests in suite TestTrace*

Similarly, the comparison engine determines that T2 and T1 are not equivalent. However, as in the case between T1 and T3, the comparison engine correctly finds that T1 covers T2.

Figure 7 shows the settings for the engine that provide the best results, along with the corresponding best scores. Figure 8 shows the overall performance. Note that the comparison engine performs best on this experiment set when both its sensitivity and its threshold for coverage are lowered, yet it surprisingly gives neither false positives nor false negatives.

3.5. Experiment Set 3

For this experiment set, the suite "MoneyTest.java"[8] is utilized. "MoneyTest.java" is a sample test suite packaged with the source of the popular unit-test framework JUnit [8]. It contains 22 tests, yielding 231 pairs to verify. The average test size is 2.59.

Manual inspection uncovered 0 (zero) duplicate tests, as did the comparison engine. The engine takes only several seconds to process the suite.

3.6. Experiment Set 4

The test suite *GenotypeTest.java*[9] from the Allelogram project [9] is utilized for this experiment set. The suite contains 9 tests, yielding 36 pairs. The average test size is 6.22.

The comparison engine finds no duplicates in this suite. Manual inspection confirms the results. The engine takes only a few seconds to process the tests.

Both [3] and [4] utilize this suite for their experiments. However, at least two tests in [4] are incorrectly mark as duplicates. Two such tests are presented in [4] and shown in figure 9 of this paper for convenience.

It is important to mention that both [3] and [4] use the code coverage tool CodeCover [10] for their analyses. However, [4] uses an earlier version of the tool, which includes some limitations that lead to the incorrect results. This shows that relying on code coverage tools is not always reliable and might actually increase the false positive rate unnecessarily. Fortunately, the comparison engine here overcomes the limitations of the engines in [3] and [4] by directly analyzing the test suite without requiring assistance of external tools.

4. CONCLUSION

This paper has presented evidence that direct, offline analysis of test suites is powerful enough to enable accurate comparison of unit-tests. Using a comparison engine based on the mathematical model provided in this paper, results of running the engine on several unit-test suites are shown to be as good as, or better than, results of engines that utilize code coverage tools and instrumentation of the SUT and the tests themselves.

Additionally, the results validate the primary claim of this paper. Namely, that verbatim unit-test source code is a good approximation of abstract states. Hence, unit-test source code can be utilized verbatim to efficiently achieve high accuracy in identifying duplicate unit-tests.

Further, the results show that the accuracy of the engine is not specific to a single programming language. The engine is, therefore, effective across a variety of programming languages chosen for unit-test suites.

5. FUTURE WORK

It is interesting to consider the extent of the impact of different language paradigms on the engine. That is, do unit-tests implemented using, say, a functional language such as ML affect the performance of the engine? In particular, does the engine perform the same, worse, or better than with unit-tests implemented in imperative languages? Thus, the effect on the engine of the programming language paradigm continuum is an interesting subject for future investigation.

Another area of further investigation is the impact on the engine of the distribution of tests in a suite. For instance, an interesting questions is: How will the performance of the engine be affected if tests are uniformly distributed throughout a suite?

A related area of further investigation is that of unit-test style. An example question is: How does the engine perform on unit-tests written in entirely different styles within the same suite?

A final thread to follow is to investigate ways of extracting more accurate abstract states from the information already contained in unit-tests. Doing so will naturally increase the accuracy of the engine. Moreover, it will also allow the engine to become universal in the sense of being able to compare unit-tests from many different suites, possibly written in distinct programming languages.

6. ACKNOWLEDGMENTS

The author would like to thank Dr. Gordon Fraser for his valuable feedback o a draft of this paper.

7. REFERENCES

[1] V. Vangala, J. Czerwonka, and P. Talluri, "Test Case Comparison and Clustering using Program Profiles and Static Execution," Microsoft.

[2] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring Test Case Similarity to Support Test Suite Understanding," 2012.

[3] N. Koochakzadeh, V. Garousi, and F. Maurer, "A Tester-Assisted Methodology for Test Redundancy Detection," 2009.

[4] N. Koochakzadeh, V. Garousi, and F. Maurer, "Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned," 2009.

[5] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," 2004.

[6] M. Gordon, and M. Kochen, "Recall-Precision Trade-off: a Derivation," 1989.

[7] S. A. Alvarez, "An exact analytical relation among recall, precision and classification accuracy in information retrieval," 2002.

[8] http://junit.org/

[9] http://code.google.com/p/allelogram/

[8] https://raw.github.com/junit-team/junit/master/src/test/java/junit/samples/money/MoneyTest.java

[9] http://allelogram.googlecode.com/svn/trunk/Allelogram/org/carlmanaster/allelogram/model/tests/GenotypeTest.java

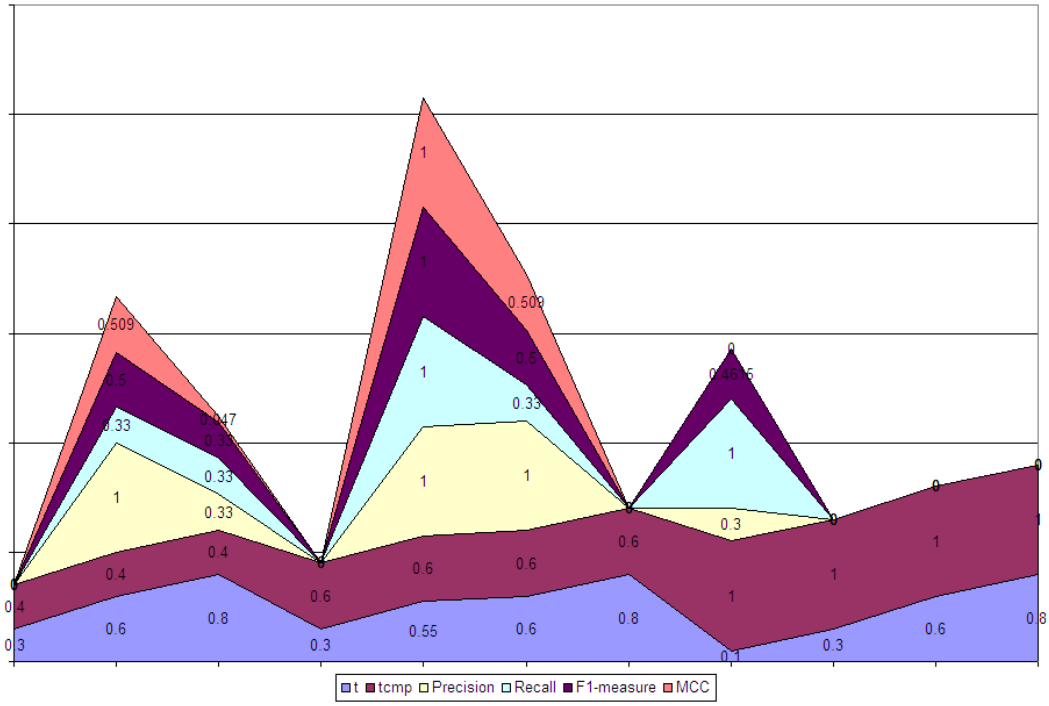

Figure 8: *Performance curves for the comparison engine on example set 2*

```
public void testAlleleOrderDoesntMatter() throws Exception {
  Genotype g1 = new Genotype(new double[]{0,1});
  Genotype g2 = new Genotype(new double[]{1,0});
  assertTrue(g1.getAdjustedAlleleValues(2).equals(g2.getAdjustedAlleleValues(2)));
}


public void testOffset() throws Exception {
  Genotype g = new Genotype(new double[]{0,1});
  g.offsetBy(0.5);
  List<Double> adjusted = g.getAdjustedAlleleValues(2);
  assertEquals(2, adjusted.size());
  assertEquals(0.5, adjusted.get(0));
  assertEquals(1.5, adjusted.get(1));
  g.clearOffset();
  adjusted = g.getAdjustedAlleleValues(2);
  assertEquals(0.0, adjusted.get(0));
  assertEquals(1.0, adjusted.get(1));
}
```

Figure 9: *Two tests that the engine in [4] incorrectly marks as duplicates*

[10] http://codecover.org/

