Predicting Effectiveness of Automatic Testing Tools

Brett Daniel[1]
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
Email: bdaniel3@uiuc.edu

Marat Boshernitsan[1]
Coverity, Inc.
San Francisco, California 94107
Email: maratb@acm.org

***Abstract*—Automatic white-box test generation is a challenging problem. Many existing tools rely on complex code analyses and heuristics. As a result, structural features of an input program may impact tool effectiveness in ways that tool users and designers may not expect or understand.**

We develop a technique that uses structural program metrics to predict the test coverage achieved by three automatic test generation tools. We use coverage and structural metrics extracted from 11 software projects to train several decision tree classifiers. Our experiments show that these classifiers can predict high or low coverage with success rates of 82% to 94%.

I. Introduction

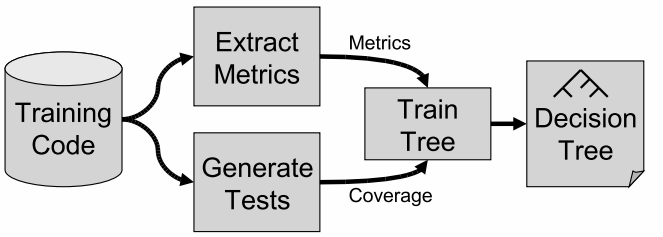
Automatic test generation is a recurring theme in computer science research [1]–[11]. Also called test-input generation, it has many uses in software engineering, ranging from augmenting manual testing efforts to automatically generating test suites.

Determining inputs that cause a program to execute a particular code path is generally undecidable [12]. Nevertheless, researchers have proposed many heuristics and approximations to produce tests for realistic source code. These heuristics and approximations are often based on the intuition (sometimes experimentally supported) about the types of programs that the analysis is likely to encounter. The unfortunate side effect is that the accuracy or the performance of the analysis algorithms may be adversely affected by the structure of the input program.

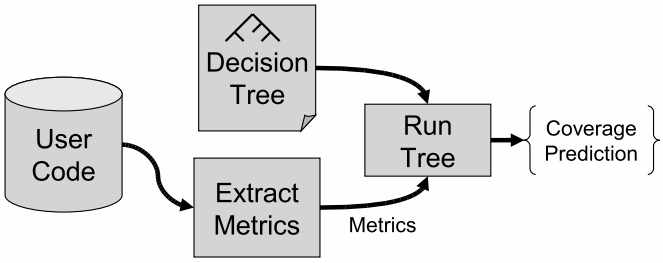
We are unaware of any techniques that automatically correlates program structure to tool effectiveness. Such knowledge would help both users and tool designers. Users would gain knowledge of why a tool did not perform in the way they expected, allowing them to adapt to the tool's limitations. Tool designers would learn the tool's limitations, allowing them to improve the tool's effectiveness.

Our technique provides a path to achieving this goal. It entails training decision trees to predict test coverage using metrics derived from program structure. First, we extract structural program metrics from a large corpus of Java source code with coverage produced by a given test generation tool. We then use off-the-shelf machine learning techniques to train a decision tree to predict whether the testing tool will achieve high or low coverage on any method for which the required metrics can be computed.

[1]The work presented in this paper was done while both authors were employed by Agitar Software, Inc., which has since ceased operations.



(a) Process for tool designers to train a decision tree



(b) Process for tool users to obtain prediction for their code

Fig. 1. Overview of the process to predict a testing tool's effectiveness

Our key contributions are the following:

- We present a novel technique by which one can use machine learning algorithms to predict automatic testing tool effectiveness from structural metrics.
- We evaluate the technique using three automatic testing tools: Agitator [6] and Mockitator from Agitar Software, and Randoop [5] from the Massachusetts Institute of Technology. We find that decision trees trained on 11 open-source Java projects can predict high or low coverage with 82% to 94% success rate.

II. Overview

In this section we give a broad overview of the process we propose to predict effectiveness of an automatic test generation tool. The process has two parts: training a decision tree using code with known coverage characteristics and using the tree to predict coverage on new code.

Figure 1(a) illustrates the training process. We begin with a large corpus of source code, which we refer to as the training code. We extract two data sets from the training code. First, we extract many metrics that characterize method structure. Second, we run the automatic testing tool to produce a suite

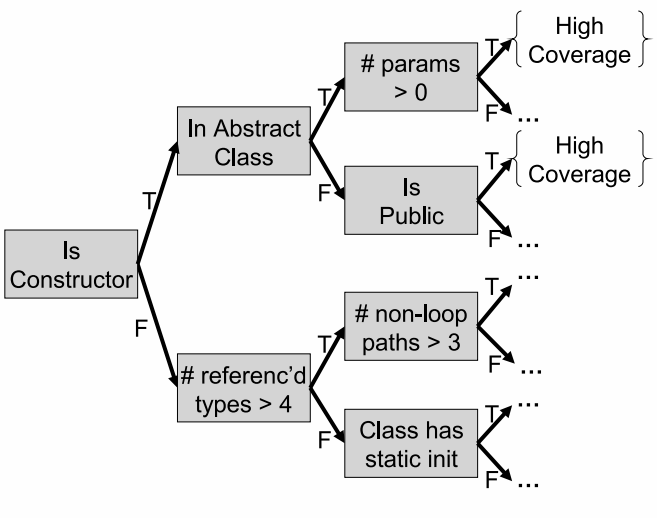


Fig. 2. The first three levels of a decision tree trained using our 11 open-source projects

of tests. These tests exercise the training code and produce coverage numbers for the methods. The coverage numbers and metrics flow into a machine learning algorithm that trains a decision tree.

Figure 1(b) presents the process a tool user would follow to predict coverage based on the structure of his or her code. A user first extracts the same metrics from his or her body of code as were extracted from the training code. These metrics flow into the decision tree that renders a coverage prediction in terms of the metrics. In our case, the prediction is a high or low coverage label, along with the set of metrics that the decision tree used.

III. Example

To illustrate the process described in Section II, we examine the `java.util.HashMap` class's `putAll` method:

```
public void putAll(Map m)
```

We train a decision tree using the coverage that Agitator (one of our test generation tools) achieved on a large body of source code. Figure 2 illustrates the first three levels of such a decision tree. An internal node contains a test against a structural metric, an edge represents the result of the test, and a leaf node represents a high or low coverage prediction. The ellipses represent deeper portions of the tree that we omit for space. A trace from the root to a leaf is the conjunction of the metric tests that led to a particular prediction.

The following partial trace for `putAll` predicts that Agitator will get low coverage. The prediction is correct; in reality, Agitator achieves just 14% coverage.

$\neg$ Is Constructor
$\wedge$ Number of Referenced Types > 4
$\wedge$ Number of Non-Loop Paths > 3
...
$\wedge$ Number of Visible Types > 0
$\wedge$ Number of Branches > 4
$\implies$ Low Coverage

Each of the metrics in the trace provides a clue to the program features that likely affected Agitator's test generation. In reality, Agitator is unable to construct a `Map` object for the `m` argument with size greater than 0. This is related to the "Number of Referenced Types" and "Number of Visible Types" metrics.

IV. Implementation and Evaluation

We are confident that the approach described in Section II is applicable to many kinds of coverage criteria, software, metrics, machine learning algorithms, and automatic testing tools. For our evaluation, we instantiated the proposed technique with combined statement and branch coverage, a training set of 11 open-source projects, 49 simple structural metrics, an off-the-shelf decision tree algorithm, and three testing tools with differing approaches to test generation. See [13] for discussion of the motivation behind these choices.

A. Training Code

We extracted metrics and coverage from 11 open-source Java projects: ANTLR, Apache Commons Collections, Apache Commons Primitives, ASM, java.util, javax.xml, Java Simple Argument Parser, Jython, TestOrrery, the TimeAndMoney library, and our own metrics extraction code. These projects ranged in size from 200 to 8,000 methods, giving us a corpus of about 21,000 methods with which to train decision trees. We have posted the full data set at http://mir.cs.uiuc.edu/predictcoverage/.

B. Structural Metrics

The key to predicting coverage lies in the choice of program features to extract from source code. We call these features *structural metrics*, and they range from familiar object-oriented metrics to metrics derived from control flow. All structural metrics are numeric or nominal values derived by traversing the program's parse tree or control flow graph.

We explored 49 metrics derived from a method's signature, body, or containing class. We found that method-targeted metrics can predict coverage much more accurately than metrics derived from larger program elements.

Each metric falls into one of the following categories:

- **Dependency metrics** derived from the types that the method references. **Example:** number of types visible in the method's signature
- **Field access metrics** derived from reads or writes to fields. **Example:** number of static field accesses
- **Method structure metrics** derived from the method's declaration. **Example:** whether the method is public or private
- **Containing class metrics** derived from the class in which the method is declared. **Example:** whether the class has a public constructor
- **Control flow metrics** derived from the method's control-flow graph. **Example:** cyclomatic complexity [14]

The full list of metrics can be found at http://mir.cs.uiuc.edu/predictcoverage/.

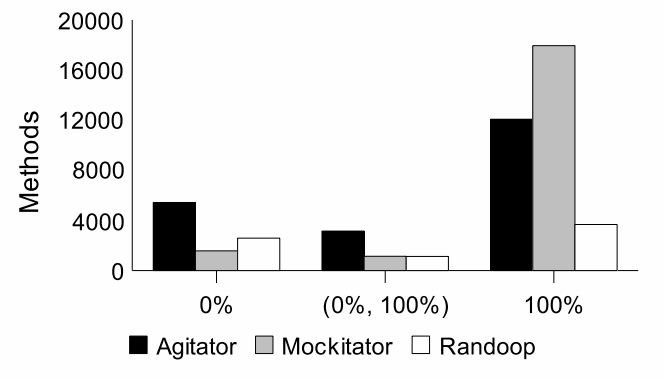


Fig. 3. Coverage distribution across all projects for each of the test generation tools

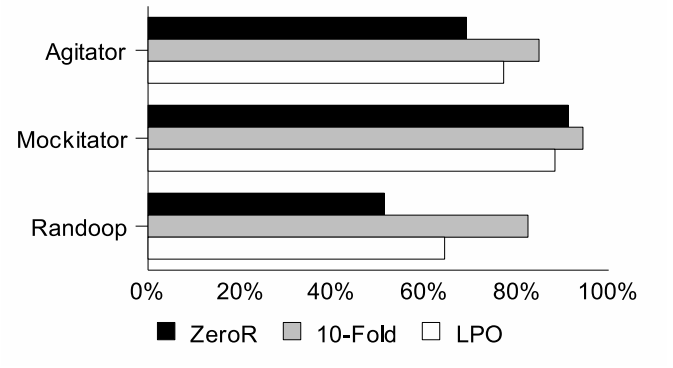


Fig. 4. Success rate using the majority class (ZeroR), 10-fold cross-validation, and leave-project-out (LPO) cross-validation

C. Decision Trees

Predictive models fall into one of two general categories: *regression models* that predict continuous values and *classification models* that predict discrete values.

Our first experiments focused on regression models, but we found that they performed poorly since coverage is bounded by $[0\%, 100\%]$. Many closely-related nominal metrics or an unusually large or small numeric metric could send the prediction out of this range, causing a nonsensical prediction less than 0% or greater than 100%.

We then experimented with several classification models and found that decision trees best served our purposes for the following reasons:

- One can easily understand a decision tree's prediction. As Figure 2 shows, a tree's internal nodes test metrics, and a prediction can be represented as a trace down the tree.
- A decision tree is convertible to and from other classification models such as decision tables, rules, and certain probabilistic models, yet is a useful representation for execution.
- A decision tree learning algorithm will ignore metrics that do not reveal new information. Thus, we can add many metrics and the tree will only contain those that the algorithm determines are useful.
- There are many mature algorithms for creating decision trees. We used an off-the-shelf implementation of the C4.5 algorithm [15] implemented in the Weka [16] toolkit.

The type of trees we chose required discrete rather than continuous coverage values. We experimented with several ways of dividing coverage into nominal values and found that a simple two-way division into high and low coverage yielded the best results. This was a natural choice since as Figure 3 shows, there are many more methods with 0% and 100% coverage than methods with coverage in the non-inclusive range $(0\%, 100\%)$. Furthermore, we found that the "split point" had little effect on the success rate, though higher splits were marginally better.

D. Testing Tools

Our experiments dealt with three automatic test generation tools: two packaged with Agitar Software's AgitarOne product, and one developed at the Massachusetts Institute of Technology.

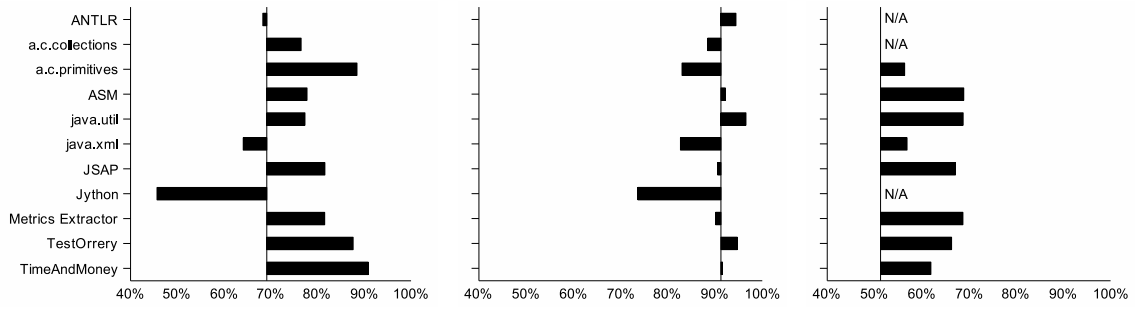
1) Test Generation with AgitarOne: AgitarOne is a comprehensive unit testing product for Java developed at Agitar Software. Test generation is supported in AgitarOne by two distinct test generation engines. Agitar's *Agitator* is a dynamic test generation engine that uses runtime feedback and limited on-the-fly static analysis to generate input data for software agitation and for regression test generation. Agitar's *Mockitator* is a static test generation engine that creates regression tests using constraint solving and mock objects [17]. In the AgitarOne product, the two engines are used in tandem, with Mockitator filling in for the coverage paths undiscovered by Agitator. For the purposes of our evaluation, we treated these two engines as independent and created separate decision trees for each.

2) Test Generation with Randoop: Randoop is a tool created by Pacheco et al. [5] that implements feedback-directed random test generation for object-oriented programs. Randoop incrementally constructs sequences of method calls that create and mutate objects. It constructs each sequence by randomly selecting an existing sequence and "extending" it with a call to a random public method. Following construction, it reflectively executes the sequence to produce results that feed back into later sequences.

E. Evaluation

We evaluated decision trees using their *success rate*, measured as the ratio of correctly classified methods to the total number of methods [16]. We found that decision trees could predict high or low coverage with success rates of 82% to 94%.

When evaluating decision trees (or, indeed, any machine learning classifier), one must partition the data set into two disjoint subsets: one for training and the other for testing. *Cross-validation* repeats the train-test process for several divisions and reports the average success rate.



(a) Agitator (b) Mockitator (c) Randoop

Fig. 5. Success rate on projects when left out of the training set, relative to each tool's overall ZeroR success rate

We used two methods of cross-validation. *10-fold cross-validation*, a widely-accepted method for verifying machine learning classifiers [16], divides the data into 10 random subsets and performs cross-validation using 9 subsets for training and the remaining subset for testing. *Leave-project-out (LPO) cross-validation*, a method specific to our experiments, trains a decision tree on all projects except one which is left out for testing.

We compared these two success rates against the success rate of the trivial *ZeroR* classifier that always predicts the most common classification. For example, if most of the training data got low coverage, ZeroR will predict low coverage on all inputs. ZeroR provides a useful indication of a predictor's worst performance, since a nonrandom prediction scheme should do better.

Figure 4 compares the ZeroR classifier's success rate against the success rates of the best decision trees measured by the two cross-validation schemes. In our experiments, 10-fold cross-validation yielded success rates between 82% and 94%, an average of 17% above ZeroR. LPO cross-validation indicated that the decision trees performed 7% and 13% better than ZeroR for Agitator and Randoop tests, but 2% worse for Mockitator tests. We speculate that the Mockitator tree's poor LPO success rate was due to Mockitator's already high coverage (and correspondingly high ZeroR success rate). A larger number of projects would likely improve LPO cross-validation, since loss of one project would have a smaller impact on the success rate.

LPO cross-validation also indicated how well a decision tree would predict coverage on individual projects. Figure 5 shows the success rate on each project when left out of the training set, relative to each tool's ZeroR success rate. "N/A" indicates that Randoop crashed while generating tests.

We found that the success rates for the Agitator (Figure 5(a)) and Randoop (Figure 5(c)) trees were higher than ZeroR's success rate on most projects. This result indicates that a decision tree trained on one corpus of code can accurately predict coverage on a previously unseen corpus of code. The success rate of the Mockitator classifier (Figure 5(b)) was less convincing, due, once again, to Mockitator's high ZeroR success rate.

V. Conclusion and Future Work

Program structure impacts the effectiveness of automatic testing tools in unexpected ways. Our proposed technique can predict tool effectiveness, measured in terms of coverage, using metrics derived from program structure. This prediction provides the first step toward an understanding of testing tool behavior in terms of program structure. In future work, we hope to explore how users and tool designers can apply the knowledge that our technique provides to explain tool behavior [13], build better testing tools, and improve user code.

References

[1] D. Beyer, A. J. Chlipala, and R. Majumdar, "Generating tests from counterexamples," in *ICSE*, 2004.
[2] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *ISSTA*, 2004.
[3] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *TACAS*, 2005.
[4] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in *ASE*, 2000.
[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007.
[6] M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing," in *ISSTA*, 2006.
[7] C. Cadar and D. R. Engler, "Execution generated test cases," in *SPIN*, 2005.
[8] P. Godefroid, "Compositional dynamic test generation," in *POPL*, 2007.
[9] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005.
[10] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for C," in *ESEC/FSE*, 2005.
[11] K. Inkumsah and T. Xie, "Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs," in *ASE*, 2007.
[12] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC*, 1971.
[13] B. Daniel and M. Boshernitsan, "Predicting and explaining automatic testing tool effectiveness," University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2008-2956, April 2008.
[14] T. McCabe, "A complexity measure," *IEEE Trans. Soft. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.
[15] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
[16] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. Morgan Kaufmann, 2005.
[17] K. Beck, *Test-Driven Development by Example*. Addison Wesley, 2003.