

Implications Of Ceiling Effects In Defect Predictors

Tim Menzies^a, Burak Turhan^b, Ayse Bener^b, Gregory Gay^a, Bojan Cukic^a, Yue Jiang^a *

^a Dept of CS& EE, West Virginia University, Morgantown, WV, USA

^b Dept. of Computer Engineering, Bogazici University, Turkey

tim@menzies.us; turhanb@boun.edu.tr; bener@boun.edu.tr

greg@4colorrebellion.com; yue, cukic@csee.wvu.edu

ABSTRACT

Context: There are many methods that input static code features and output a predictor for faulty code modules. These data mining methods have hit a “performance ceiling”; i.e., some inherent upper bound on the amount of information offered by, say, static code features when identifying modules which contain faults.

Objective: We seek an explanation for this ceiling effect. Perhaps static code features have “limited information content”; i.e. their information can be quickly and completely discovered by even simple learners.

Method: An initial literature review documents the ceiling effect in other work. Next, using three sub-sampling techniques (under-, over-, and micro-sampling), we look for the lower useful bound on the number of training instances.

Results: Using micro-sampling, we find that as few as 50 instances yield as much information as larger training sets.

Conclusions: We have found much evidence for the limited information hypothesis. Further progress in learning defect predictors may not come from better algorithms. Rather, we need to be improving the information content of the training data, perhaps with case-based reasoning methods.

Categories and Subject Descriptors

i.5 [learning]: machine learning; d.2.8 [software engineering]: product metrics

General Terms

algorithms, experimentation, measurement

Keywords

Naive Bayes, under-sampling, over-sampling, defect prediction

*The research described in this paper was supported by Bogazici University research fund under grant number BAP-06HA104 and at West Virginia University under grants with NASAs Software Assurance Research Program. Reference herein to any specific commercial product, process, or service by trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PROMISE'08, May 12–13, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-036-4/08/05 ...\$5.00.

1. INTRODUCTION

The current state of the art in learning defect predictors is curiously static. As shown below, better results have not been forthcoming, despite the application of supposedly better data miners.

If better algorithms are not useful, perhaps it is time to better understand the training data. Accordingly, instead of:

- Changing the *data miners*...
- this paper changes the *training data*.

Specifically, we study how *throwing training data away* effects the performance of our defect predictors. The results are quite surprising. In experiments with under/over-sampling and incremental cross-validation, we found that *much of the training data can be discarded without losing effectiveness in defect prediction*. This leads to the following notion:

Hypothesis: Static code features have *limited information content*.

This, in turn, leads to three predictions:

Prediction1: The information from static code features can be quickly and completely discovered by even simple learners.

Prediction2: More complex learners will not find new information.

Prediction3: Further progress in learning defect predictors will not come from better algorithms, but from improving the information content of the training data.

We provide empirical evidence for and discuss the validity of **Prediction1** and **Prediction2**. It turns out that a simple learner like Naive Bayes can extract the embedded information in static code features better than a number of sophisticated learners, including more complex Bayesian learners. **Prediction3** can be used to greatly simplify the collection of more insightful training data. We show that simple techniques like log filtering and feature weighting can improve the prediction performance. The final discussion section of this paper discusses methods for increasing the information in the training data in detail.

2. BACKGROUND

For some time now, we have applied data miners to build defect predictors from static code measures [4, 8, 20, 27–31, 33–37]. To learn such predictors, tables of historical examples (like those in Figure 1) are formed where one column has a boolean value for “faults detected” and the other columns describe software features such as (i) lines of code, (ii) number of unique symbols [18], or (iii) max. number of possible execution pathways [26].

data	language	(# modules) examples	features	%defective
pc5	C++	17,186	38	3.0
mc1	C++	9,466	38	0.71
pc2	C++	5,589	36	0.41
kc1	C++	2,109	21	15.45
pc3	C++	1,563	37	10.23
pc4	C	1,458	37	12.2
pc1	C++	1,109	21	6.94
kc2	C++	522	21	20.49
cm1	C++	498	21	9.83
kc3	JAVA	458	39	9.38
mw1	C++	403	37	7.69
mc2	C++	61	39	32.29
		40,422		

Figure 1: Twelve tables of data, sorted in order of number of examples. All these tables are in the PROMISE repository.

Each row in the table holds data from one “module”; i.e. the unit of functionality. Depending on the language, modules may be called “functions”, “methods”, “procedures” or “files”.

The data mining task is to find combinations of features that predict for the value in the defects column. Once such combinations are found, managers can use them to determine where to best focus their QA effort. Better yet, if they have already focused their QA effort on the most critical portions of the system, the detectors can “nudge” them towards areas that need the most attention. Note that these data miners do not predict the total number of defects, just the number of modules containing more than zero defects.

In theory, such defect predictors are not useful. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [13].

In practice, they are quite effective, at least for the NASA code we have studied [20, 28, 30, 31, 33–37, 41–43]. In those data sets, our fault prediction models find defect predictors [35] with a probability of detection (pd) and probability of false alarm (pf) of $mean(pd, pf) = (71\%, 25\%)$. These values are much higher than known industrial averages for manual defect detection [11, 40].

In January 2007, we published a study [35] that defined a repeatable experiment in learning defect predictors. The intent of that work was to offer a benchmark in defect prediction that other researchers could repeat/ improve/ refute. That experiment used:

- Public domain data sets (from the the PROMISE repository);
- Open source data mining tools (the WEKA toolkit [44]);
- Randomly sorting training data rows (stops order effects);
- 10-way cross-validation (to test on data *not* used in training);
- Learning via multiple types of machine learning algorithms (rule learners, decision tree learners, Bayes classifiers);
- Assessment via multiple criteria such as probability of detection (pd), probability of false alarm (pf), and *balance* that combines $\{pd, pf\}$ ($balance$ is defined in Figure 2);
- Statistical hypothesis tests over the assessment criteria;
- Novel visualization methods for the results;
- Feature subset selection to find the most important subset of the static code features;

Surprisingly, very simple Bayes classifiers (with a simple pre-processor for the numerics) out-performed the other studied methods. Since that study, we have tried to find better data mining algorithms for defect prediction. To date, we have failed. Our recent (as yet, unpublished) experiments have found no additional statistically significant improvement from the application of the following

If $\{A, B, C, D\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

$$pd = \text{recall} = \frac{D}{B + D} \quad (1)$$

$$pf = \frac{C}{A + C} \quad (2)$$

$$bal = \text{balance} = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \quad (3)$$

All these values range zero to one. Better and *larger* balances fall *closer* to the desired zone of no false alarms and 100% detection. Other measures such as *accuracy* and *precision* were not used since, as shown in Figure 1, the percent of defective examples in our tables was usually very small (median value around 8%). Accuracy and precision are poor indicators of performance for data were the target class is so rare (for more on this issue, see [33, 35]).

Figure 2: Performance measures.

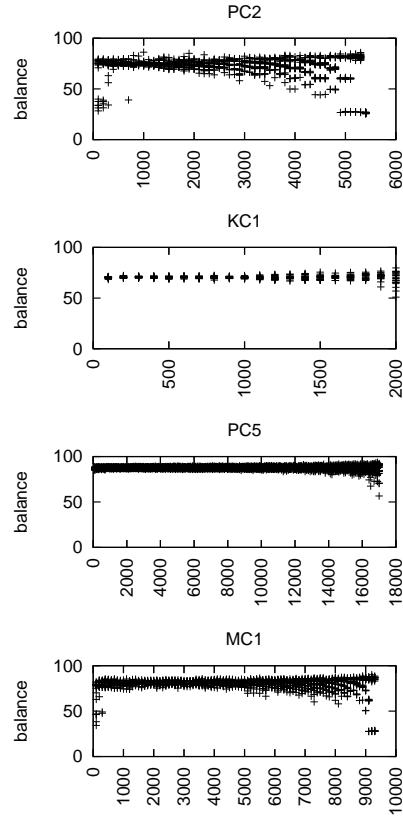


Figure 3: Experiments with training set size vs *balance*.

data mining methods: logistic regression; average one-dependence estimators [15]; under- or over-sampling [10] random forests [2], RIPPER [6], J48 [39], OneR [19] and bagging [3]. Only boosting [14] on discretized data offers a statistically better result than a Bayes classifier. However, we cannot recommend boosting: the median improvement is quite negligible and boosting is orders of magnitudes slower than a simple Bayes classifier.

Other researchers have also failed to improve our results. Due to our connection with PROMISE, we are aware of studies by other researchers (currently under review) that tried other data mining methods. Those studies investigated the statistical difference of the

results from two dozen learners on the same datasets. The simple Bayesian method discussed above ties in first place along with 15 other methods.

How can we explain all these failed attempts to improve fault prediction models in repeatable experiments using same (PROMISE) data sets? One lesson from the above is that exploring better algorithms may not be productive. Hence, we explored the data that the algorithms were processing.

Elsewhere [32] we checked how *little* information was required to learn a defect predictor. Defect predictors were learned from $N = 100, N = 200, N = 300$. instances then *Tested* on another 100 instances. For each N , 10 experiments were performed where training was conducted on $|Train| = 90\% * N$ instances and testing on $|Test| = 100$ (and $Train \cap Test = \emptyset$). For all experiments, the $N, Train, Test$ instances were selected at random.

This study was conducted on the twelve data sets of Figure 1. Space does not permit showing all the results but a representative sample are shown in Figure 3 [32]. In that figure, the X-axis is the size of training set and the Y-axis is the *balance* measure defined in Figure 2. Note that the performance does not change much regardless of whether the model is inferred from 100 instances or from up to several thousand instances. In fact, learning from too many training examples was actually detrimental (witness the widening variance as the training set increases). A Mann Whitney U test [25] (95% confidence) confirms the visual pattern apparent in Figure 3: static code features used as the basis for predicting module’s fault content reveal all that they can reveal after as little as 100 instances.

The above results motivated new experiments, reported below.

3. EXPERIMENTS

The goal of these experiments was two-fold:

1. Can we confirm the effects of Figure 3; i.e. that ignoring large amounts of the training data is not harmful?
2. Can we exploit that effect to build better defect predictors?

Goal #1 was achieved and we have a promising lead on Goal #2.

3.1 Experiment #1: Over- & Under- Sampling

The Figure 3 experiment randomly discarded training data. Perhaps a *more controlled sub-sampling method* would not damage the information content of the data. If so then, contrary to Figure 3, increasing the sample size will improve performance.

Over- and under-sampling [5, 10] are examples of “more controlled sub-sampling methods”. Both methods might be useful in data sets with highly imbalances class frequencies. For example, in Figure 1, the frequency of the target class is very low (median value lies between 6.94% and 7.69%).

To sub-sample, a target class is selected; in this study, we selected defective modules as the target class. The sampling program runs in two passes through the data. In pass 1, a table of frequency counts is built. If the desired goal is reached in fewer instances than the other classes, a second pass is taken through the data. This is where the two sampling techniques differ:

- In the case of *under-sampling*, instances are selected until the combined number of instances with other classes is equal to the number of instances with the desired goal. This results in a much smaller dataset, but the desired goal is no longer buried inside a larger set of other classes.
- *Over-sampling* follows a similar procedure but, instead of limiting the number of instances with other classes, the sampling program adds randomly selected examples of the target class back into the data set. Where as under-sampling produces smaller data sets, over-sampling grows the size of the

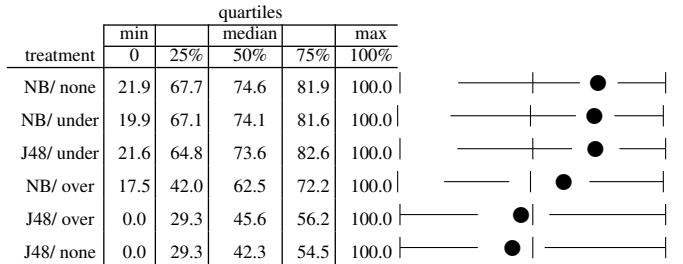


Figure 4: Over- & under- & no sampling results. Sorted descending by median *balance* results. Visually, there is clear loser (J48/none) and a three-way tie for the winning method (NB/none, NB/under, J48/under). These visual impression are confirmed by the statistical tests of Figure 5.

data set until the minority class has the same frequency as the majority class.

Regardless of the which of sub-sampling method is used, the result is a data set with an equal number of target and non-target classes.

Over- and under-sampling experiments were conducted on the Figure 1 data using 10^6 10-way cross-validation¹. This study used a simple Bayes classifier (since it was useful in our prior experiment [35]) plus a C4.5-like decision tree learner, J4.8 [44] (since that was used in prior under- and over- sampling experiments [10]).

The *balance* results (defined in Figure 2 for each *treatment* (data miner, plus sampling policy) were sorted and displayed as *quartile charts* of Figure 4. To generate these charts, the *balance* results for some treatment are sorted and labeled as follows:

$$\begin{array}{c} q1 \\ \overbrace{4, 7, 15, 20, 31,} \quad \overbrace{40,} \quad \overbrace{52, 64, 70, 81, 90} \\ min \qquad median \qquad max \end{array}$$

In a quartile chart, the upper and lower quartiles are marked with black lines; the median is marked with a black dot; and a vertical bar is added to mark the 50% value. The above numbers would therefore be drawn as follows:

$$0\% | \text{---} \bullet | \text{---} | 100\%$$

We prefer quartile charts of performance deltas to other summarization methods for M*N studies. They offer a very succinct summary of a large number of experiments.

The Figure 4 results are consistent with certain prior results:

- The simple Naive Bayes we recommended previously [35] performed as well as anything else. In a result consistent with the limited information hypothesis discussed in the introduction, seemingly cleverer learning schemes did not outperform simple Bayesian classifiers.
- Just like the Figure 3 results, throwing away data (i.e. under-sampling) does not degrade the performance of the learner. In fact, in the case of J48, throwing away data improved the median *balance* performance from around 40% to over 70%.
- Under-sampling beat over-sampling for both J48 and Naive Bayes. This result is consistent with Drummond & Holte’s

¹Ten times, randomize the order of the instances in the data. Each time, divide data into i bins. For each bin $j \in 1..i$. Let *test* be bin j and let *train* be the remaining bins. Learn on *train* then evaluate on *test*.

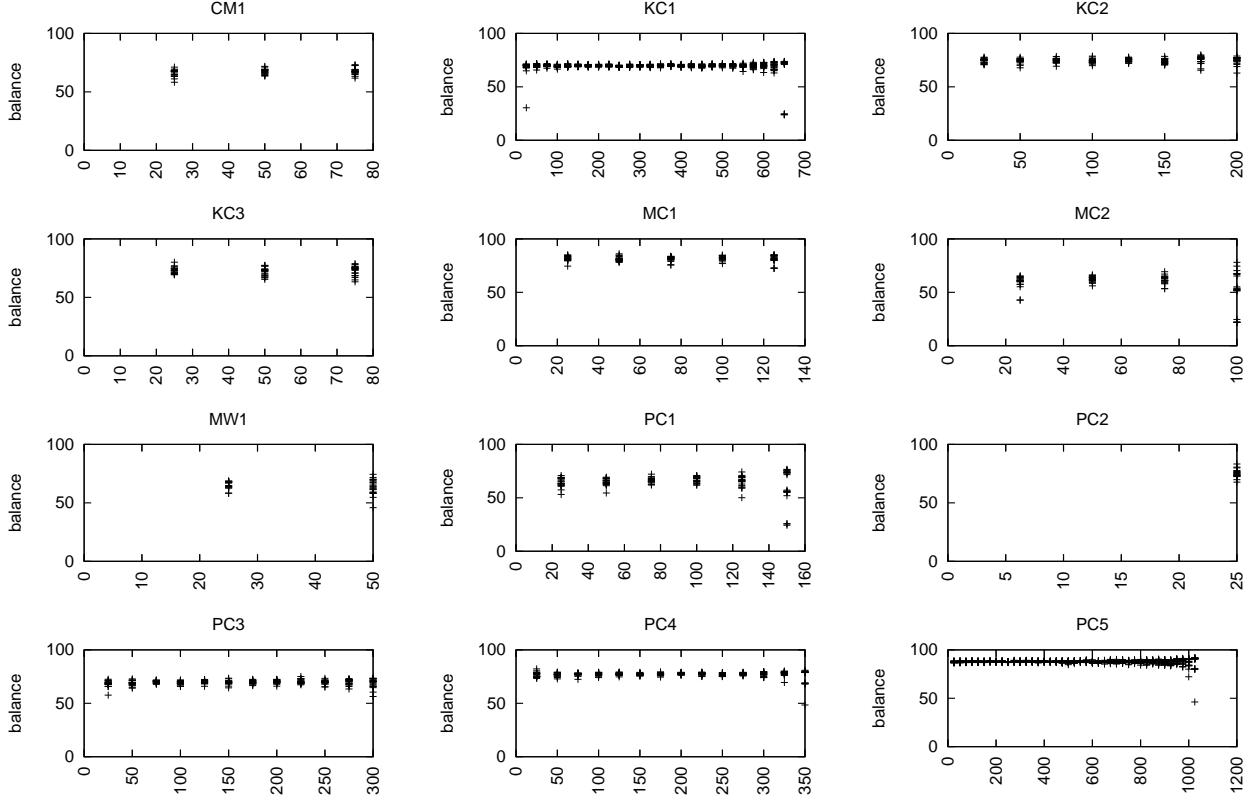


Figure 6: Micro-sampling results

learner / sampler	win - loss	win	loss	ties
NB / none	3	3	0	2
NB / under	3	3	0	2
J48 / under	3	3	0	2
NB / over	-1	2	3	0
J48 / over	-3	1	4	0
J48 / none	-5	0	5	0

Figure 5: Statistical tests on the Figure 4 results. Mann-Whitney [25] compared the median ranks of all the *balance* values seen in Figure 4. Two treatments tied if their median ranks were statistically insignificantly different (95% confidence). Otherwise, medians were compared numerically to assign “win” or “loss”. Table sorted descending on total number of *win - loss* for each treatment against the other five.

sub-sampling experiments [10] and the sub-sampling classification tree experiments of Kamei et al. [21]².

However, Figure 4 has some new results which, as far as we are aware, have not been reported elsewhere:

- Observe how *NB/none* is one of the topped-ranked methods. That is, sub-sampling decision tree learning does not out-perform Naive Bayes.
- *NB/none* ties with *NB/under*. That is, while sub-sampling offers no improvement over un-sampled Bayesian learning, under-sampling does not harm classifier performance,

²Due to differences in experimental methods, we find we cannot compare our results to the regression tree and LDA analysis of [21].

This last point is the most significant. It means effective detectors can be learned from a very small sample of the available data—an issue we explore below.

3.2 Experiment #2: Micro-sampling

In order to determine the lower-limit on the number of cases that require manual inspection, we performed another experiment. Another under-sampling policy was employed, which we call *micro-sampling*. Given N defective modules in a data set,

$$M \in \{25, 50, 75, \dots\} \leq N$$

defective modules were selected at random. Another M non-defective modules were selected, at random. The combined $2M$ data set was then passed to a 10^*10 -way cross validation.

Formally, under-sampling is a micro-sampling where $M = N$. Micro-sampling explores training sets of size up to N , standard under-sampling just explores once data set of size $2N$.

For this study we used Naive Bayes since, in all the above work, it seems to be doing as well as anything else. Figure 6 shows the results of an under-sampling study where $M \in \{25, 50, 75, \dots\}$ defective modules were selected at random, along with an equal M number of defect-free modules. Note the same visual pattern as before: increasing data does not necessarily improve *balance*.

Mann-Whitney tests were applied to test this visual pattern. Detectors learned from small M instances do as well as detectors learned from any other number of instances.

- For eight data sets, {CM1,KC2, KC3,MC1, MC2, MW1, PC1, PC2}, micro-sampling at $M = 25$ did just as well as anything larger sample size.

- For one data sets, {PC3}, best results were seen at $M = 75$. However , in many cases $\frac{8}{11}$, $M = 25$ did just as well as anything else.
- For three data sets {PC4,KC1,PC5} best results were seen at $M = \{200, 575, 1025\}$, respectively. However, for these three data sets, in all by one case $M = 25$ did as well as any larger value.

In summary, the number of cases that must be reviewed in order to arrive at the performance ceiling of defect predictor is very small: as low as 50 randomly selected modules (25 defective and 25 non-defective).

4. DISCUSSION

There exist numerous incremental case-based reasoning tools [7, 23, 24] that ask humans to audit a stochastic sample of real-world cases. Insights gained from those sessions are automatically generalized and applied to another random sample. Experts then review the classifications made on the new sample, and offer further refinements. As far back as 1999, software metrics experts Fenton and Neil [12] postulated that such human-machines-based system might out-perform systems based on static code measures (since other features/metrics could be accounted for that cannot currently be addressed using static code metrics).

When is case-based reasoning preferable to automatic data mining? While there are many answers to this question, this paper can make specific comments about one particular issue. Case-based reasoning methods require humans to examine and comment on specific cases. This is impractical if learning adequate theories requires examining a very large number of cases.

The results of experiment #1 suggest that, for static code measures, it is not necessary to manually inspect thousands of cases. In fact, just a few hundred may suffice. Consider the 9,466 modules of MC1. This data set has a defective module rate of 0.71% ; i.e. $9,466 * 0.71 / 100 = 67$ modules:

- When under-sampling, a data set of 134 modules is created (all the defective, plus 67 others, selected at random).
- When micro-sampling, the results of Experiment #2 suggest that 50 modules would suffice (any 25 defective modules, but any other 25- selected at random).

These results raise the possibility that a human-in-the-loop case-based reasoning environment might *perform as well as* automatic methods, despite the automatic methods exploring more examples.

Based on other recent research, we go further and hypothesize that such an environment might *perform better* than automatic methods. Elsewhere, we have explored combining static code measures with other measures that, serendipitously, a particular domain may contain. For example, at ISSRE 2007 [20], we reported experiments where static code measures were combined with results from an ultra-lightweight text miner. Figure 7 shows how a remarkable improvement in learner performance was achieved by applying *combinations* of requirements and code features. Other results are analogous to Figure 7:

- In the JMLR special issue on feature selection, Guyon and Elisseeff provide simple examples showing that the information content of data can be significantly increased when features are used together rather than individually [17]. In the context of this paper, we would re-express that result as: *using features from different sources, e.g. requirements and source code, can significantly increase the information content of SE data.*

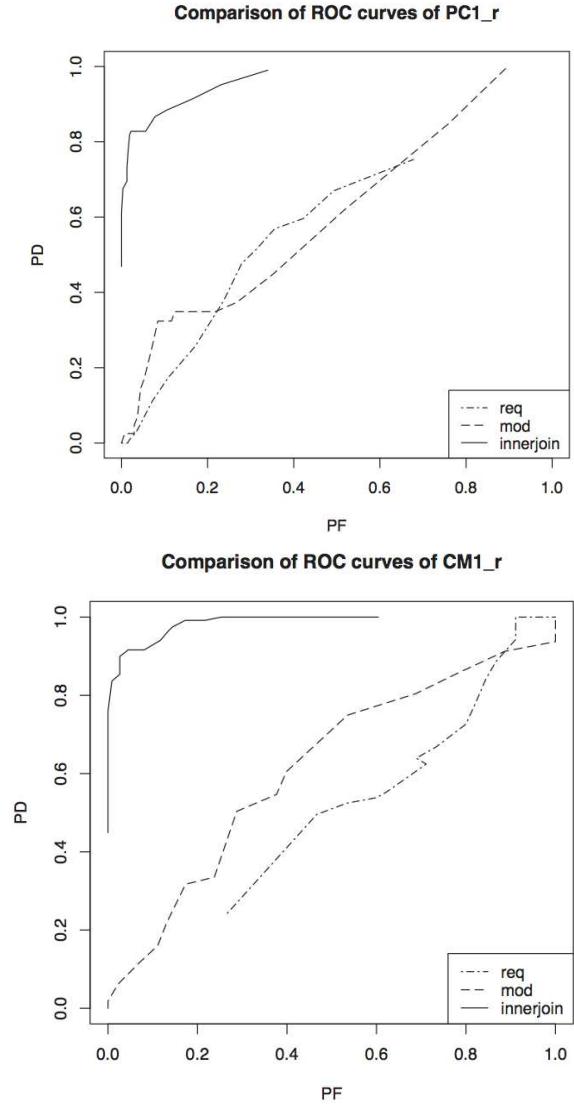


Figure 7: $\{pd, pf\}$ curves seen when using code and/or requirements features. Results from CM1 and PC1 after a 10-way cross validation. The ideal spot on these ROC curves is top left; i.e. no false alarms and perfect detection ($\{pd, pf\} = \{1, 0\}$). The dashed lines on those plots show $\{pd, pf\}$ results when fault prediction models used features mined from requirements text, or features mined from static code measures (in isolation). The solid lines show the results of models which used these two kinds of features in combination. From [20].

- Face recognition differentiates from other image processing research by using facial properties such as the location of the nose, lips, eyes etc. Combination of facial characteristics from such different sources improves the performance of face detectors that solely use i.e. pixel values of the image [16]. We argue that SE domain should make use of domain specific knowledge (when possible) to differentiate from general data mining tasks.

Note that the lesson of Figure 7 is not “always augment static code features with requirements features”. This is impractical ad-

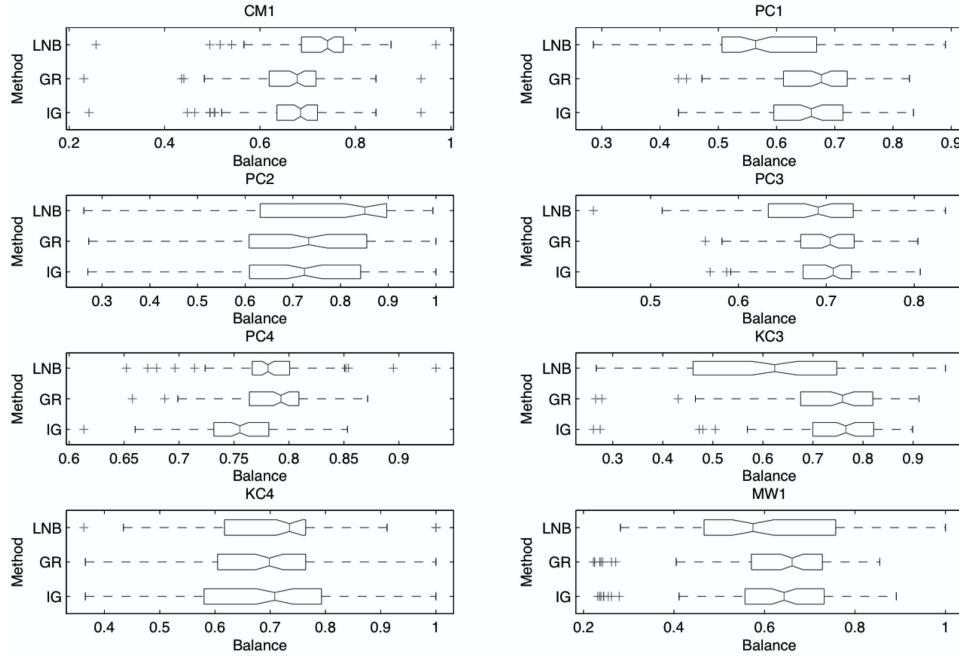


Figure 8: Comparison of three Naive Bayes schemes. LNB is the log-filtered Naive Bayes used in our IEEE TSE 2007 paper [35]. IG and GR are two variants on Naive Bayes where some oracle has weighted the attributes: IG uses an InfoGain oracle and GR uses a Gain Ratio oracle. Box plots for three learners after 10x10 cross validation. Balance values can be improved by weighting static code features. Feature subset selection corresponds to 0–1 hard weight assignment. Weights of non-informative features are automatically assigned 0. Other features are assigned a soft weight value in [0..1] interval according to their defect related information content. Note that, in $\frac{7}{8}$ of these results, the weighted schemes have higher medians or higher maximum values, or both.

vice since not all domains allow data mining specialists access to source code and the requirements that produced them; e.g. very few open source projects have detailed textual requirements documents.

Rather, the lesson of Figure 7 is that it can be very useful to let experts access and combine features from whatever sources are locally available. Such an “explore whatever” environment is not an automatic black box data miner. Rather, it is a human-in-the-loop case-based reasoning (CBR) environment where humans reflect on the specifics of particular cases, connect to different data sources, and (sometimes) run automatic data miners on combinations or subsets of a variety of types of features.

Two prior PROMISE papers [1, 9] suggest how such CBR environments might operate:

- At PROMISE 2006, Amor et.al. [1] describe an incremental classification cycle where, in round i , an expert classifies a random selection of the available data. A classifier learned from this examples is then applied to other, randomly selected examples. Then, in round $i + 1$, the expert reviews the results of that those classifications to fix any incorrect classifications from round i (formally, this is boosting where humans are used to identify examples that were harder to classify in the proceeding rounds).
- At PROMISE 2007, Dekhtyar et.al. [9] showed improvement in the performance of a traceability environment after several rounds of an expert reviewing results automatically inferred in a prior round.

Both Amor et.al. and Dekhtyar et.al. report that it took hundreds of examples and several rounds to generate useful detectors. Nei-

ther team experimented with (a) decreasing the number of examples seen in each round; or (b) micro-sampling the examples from each round. Based on this paper, we would speculate that policies (a) and (b) would significantly decrease the time required by an expert to achieve performance plateaus.

Another role for human experts in a CBR environment is to instruct the learner how *combinations* of attributes can work together to provide solutions. For example a standard Naive Bayes classifier gives equal weights to all attributes then uses frequency counts to learn the relative importance of each attribute. Elsewhere [42, 43] we have allowed an oracle to offer unequal attribute weights. In this scheme, instead of feature subset selection [22], we have used all attributes with weights assigned to them. We have converted the information gain and gain ratio of features into weight values. The results are compelling such that the performance can improve over standard Naive Bayes, and there is also no need for dealing with feature subset selection. Though the improvements in Figure 8 are not ground-breaking, they provide a hint regarding the value of unequal treatment of information sources. In the case of Figure 8, weights are assigned by the model. As Fenton and Neil warn us, the weights provided by models may not be meaningful to humans in the process [12]. However, we argue that weights assigned to different information sources by human experts *with business knowledge* can increase the quality of solutions.

5. CONCLUSIONS

The development of fault prediction models has been a very active research area. The reason for such a significant attention to automated quality predictors lays in their practical importance. Cur-

rent models are useful, as they allow software project managers to better guide the allocation usually meager quality assurance resources to artifacts which need them the most.

Recent results now indicate that this current research paradigm, which relied on relatively straightforward application of machine learning tools, has reached its limits. Building software quality predictors via data mining is essentially an inductive generalization over past experience. According to Mitchell's classic model of data mining [38], any inductive generalization explores a version space of possible theories. All data miners hit a performance ceiling effect when they cannot find additional information that better relates software measure with fault occurrence.

To build better quality predictors that break through ceiling effects, we must introduce more topology into the search space. Standard machine learning algorithms lack the business knowledge which characterizes software projects. To add that business knowledge, we propose human-in-the-loop CBR tools. We expect that this approach will allow software managers to safely focus on the application of quality assurance techniques of choice, confident that automated quality predictors will raise alerts about the artifacts where quality issues actually exist.

Three results make us advocate this kind of CBR tool:

- The environment would impractical if the human operator must examine a very large number of cases. The micro-sampling results suggest that this might not be the case. Indeed, as few as 50 randomly selected instances might suffice.
- The environment would run fast. If 50 instances are enough, then training should be virtually instantaneous. A human user could explore a very large number of features or combinations of features, all the while getting very rapid feedback.
- The environment, or some other new direction, is required. As argued above, our current generation of AI algorithms have hit a ceiling effect. We doubt that further progress will be made using better algorithms. Instead, we would advocate methods (like the CBR tool briefly described above), that increase the information content of the training data.

6. FUTURE WORK

In summary, we think it is time to change the subject of the *which* question. Rather than *which learner* we should focus on *which data*.

In this context, our future direction is clear- build a human-in-the-loop CBR environment for learning defect predictors, then benchmark that environment against automatic methods.

Other candidates for future work are

- To better understand the attribute weighting schemes explored in Figure 8.
- To improve on the randomized sampling strategies used in this paper. It may be possible to increase the performance of defect predictors with wiser sampling strategies (e.g. some initially clustering to find a small number of most representative, or most unusual, instances).

7. REFERENCES

- [1] J. Amor, G. Robles, J. Gonzalez-Barahona, and A. Navarro. Discriminating development activities in versioning systems: A case study. In *Proceedings, PROMISE 2006*, 2006. Available from http://promisedata.org/pdf/phil2006AmorRoblesGonzales_BarahonaNavarro.pdf.
- [2] L. Breiman. Random forests. *Machine Learning*, pages 5–32, October 2001.
- [3] L. Brieman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.
- [4] Zhihoa Chen, Tim Menzies, and Dan Port. Feature subset selection can improve software cost estimation. In *Proceedings, PROMISE workshop, ICSE 2005*, 2005. Available from <http://menzies.us/pdf/05/fsscocomo.pdf>.
- [5] Alexander Yun chung Liu. The effect of oversampling and undersampling on classifying imbalanced text datasets. Master's thesis, 2004. Available from http://www.lans.ece.utexas.edu/~aliu/papers/aliu_masters_thesis.pdf.
- [6] W.W. Cohen. Fast effective rule induction. In *ICML'95*, pages 115–123, 1995. Available on-line from <http://www.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps>.
- [7] P. Compton, L. Peters, G. Edwards, and T.G. Lavers. Experience with ripple-down rules. *Knowledge-Based Systems*, 19(5):356–362, September 2006. Available from http://www.cse.unsw.edu.au/~compton/#Starter_Papers.
- [8] A. Dekhtyar, J. Huffman Hayes, and T. Menzies. Text is software too. In *International Workshop on Mining Software Repositories (submitted)*, 2004. Available from <http://menzies.us/pdf/04msrtext.pdf>.
- [9] A. Dekhtyar, J.H. Hayes, and J. Larsen. Make the most of your time: How should the analyst work with automated traceability tools? In *3rd International Workshop on Predictive Modeling in Software Engineering (PROMISE'2007)*, 2007.
- [10] C. Drummond and R. C. Holte. C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on Learning from Imbalanced Datasets II*, 2003.
- [11] M. Fagan. Advances in software inspections. *IEEE Trans. on Software Engineering*, pages 744–751, July 1986.
- [12] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. Available from <http://citesear.nj.nec.com/fenton99critique.html>.
- [13] N. E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.
- [14] Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *JCSS: Journal of Computer and System Sciences*, 55, 1997.
- [15] G.I.Webb, J. Boughton, and Z. Wang. Not so naive bayes: Aggregating one-dependence estimators. *Machine Learning*, 58(1):5–24, 2005. Available from <http://www.csse.monash.edu.au/~webb/Files/WebbBoughtonWang05.pdf>.
- [16] B. Gokberk, M. O. Irfanoglu, L. Akarun, and E. Alpaydin. Learning the best subset of local features for face recognition. *Pattern Recogn.*, 40(5):1520–1532, 2007.
- [17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection'. *Journal of Machine Learning Research*, pages 1150–1182, March 2003.
- [18] M.H. Halstead. *Elements of Software Science*. Elsevier, 1977.
- [19] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

- [20] Y. Jiang, B. Cukic, and T. Menzies. Fault prediction using early lifecycle data. In *ISSRE'07*, 2007. Available from <http://menzies.us/pdf/07issre.pdf>.
- [21] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken-ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 196–204, 20-21 Sept. 2007.
- [22] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.
- [23] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann, 1993.
- [24] J.L. Kolodner. Improving Human Decision Making Through Case-Based Decision Aiding. *AI Magazine*, page 68, Summer 1991.
- [25] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 1947. Available on-line at <http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&hand%1=euclid.aoms/1177730491>.
- [26] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
- [27] T. Menzies. Practical machine learning for software engineering and knowledge engineering. In *Handbook of Software Engineering and Knowledge Engineering*. World-Scientific, December 2001. Available from <http://menzies.us/pdf/00ml.pdf>.
- [28] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman. Assessing predictors of software defects. In *Proceedings, workshop on Predictive Software Models, Chicago*, 2004. Available from <http://menzies.us/pdf/04psm.pdf>.
- [29] T. Menzies, J. Di Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and Davis J. When can we test less? In *IEEE Metrics'03*, 2003. Available from <http://menzies.us/pdf/03metrics.pdf>.
- [30] T. Menzies, J.S. Di Stefano, and M. Chapman. Learning early lifecycle IV&V quality indicators. In *IEEE Metrics '03*, 2003. Available from <http://menzies.us/pdf/03early.pdf>.
- [31] T. Menzies, Justin S. Di Stefano, Chris Cunanan, and Robert (Mike) Chapman. Mining repositories to assist in project planning and resource allocation. In *International Workshop on Mining Software Repositories*, 2004. Available from <http://menzies.us/pdf/04msrdefects.pdf>.
- [32] T. Menzies, B. Turhan, A. Bener, and J. Distefano. Cross- vs within-company defect prediction studies. Technical report, Computer Science, West Virginia University, 2007. Available from <http://menzies.us/pdf/07ccwc.pdf>.
- [33] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with precision. *IEEE Transactions on Software Engineering*, September 2007. <http://menzies.us/pdf/07precision.pdf>.
- [34] Tim Menzies, Justin S. DiStefano, Mike Chapman, and Kenneth McGill. Metrics that matter. In *27th NASA SEL workshop on Software Engineering*, 2002. Available from <http://menzies.us/pdf/02metrics.pdf>.
- [35] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from <http://menzies.us/pdf/06learnPredict.pdf>.
- [36] Tim Menzies, Robyn Lutz, and Carmen Mikulski. Better analysis of defect data at NASA. In *SEKE03*, 2003. Available from <http://menzies.us/pdf/03superodc.pdf>.
- [37] Tim Menzies and Justin S. Di Stefano. How good is your blind spot sampling policy? In *2004 IEEE Conference on High Assurance Software Engineering*, 2003. Available from <http://menzies.us/pdf/03blind.pdf>.
- [38] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [39] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.
- [40] F. Shull, V.R. Basili ad B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M.V. Zelkowitz. What we have learned about fighting defects. In *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, pages 249–258, 2002. Available from http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps.
- [41] J.S. Di Stefano and T. Menzies. Machine learning for software engineering: Case studies in software reuse. In *Proceedings, IEEE Tools with AI*, 2002, 2002. Available from <http://menzies.us/pdf/02reusetai.pdf>.
- [42] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. *qsic*, 0:231–237, 2007.
- [43] B. Turhan and A. Bener. Software defect prediction: Heuristics for weighted naive bayes. In *Second International Conference in Software and Data Technologies (ICSOFT 2007)*, pages 244–249, 2007.
- [44] Ian H. Witten and Eibe Frank. *Data mining. 2nd edition*. Morgan Kaufmann, Los Altos, US, 2005.

How to Build Repeatable Experiments

Gregory Gay, Tim Menzies, Bojan Cukic
CS& EE, WVU, Morgantown, WVU
gregoryg@csee.wvu.edu,
tim@menzies.us, cukic@csee.wvu.edu

Burak Turhan
NRC Institute for Information Technology
Ottawa, Canada
Burak.Turhan@nrc-cnrc.gc.ca

ABSTRACT

The mantra of the PROMISE series is “repeatable, improvable, maybe refutable” software engineering experiments. This community has successfully created a library of reusable software engineering data sets. The next challenge in the PROMISE community will be to not only share data, but to share experiments. Our experience with existing data mining environments is that these tools are not suitable for publishing or sharing repeatable experiments.

OURMINE is an environment for the development of data mining experiments. OURMINE offers a succinct notation for describing experiments. Adding new tools to OURMINE, in a variety of languages, is a rapid and simple process. This makes it a useful research tool. Complicated graphical interfaces have been eschewed for simple command-line prompts. This simplifies the learning curve for data mining novices. The simplicity also encourages large scale modification and experimentation with the code.

In this paper, we show the OURMINE code required to reproduce a recent experiment checking how defect predictors learned from one site apply to another. This is an important result for the PROMISE community since it shows that our shared repository is not just a useful academic resource. Rather, it is a valuable resource industry: companies that lack the local data required to build those predictors can use PROMISE data to build defect predictors.

Categories and Subject Descriptors

i.5 [learning]: machine learning; d.2.8 [software engineering]: product metrics

Keywords

algorithms, experimentation, measurement

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© ACM 2009 ISBN: 978-1-60558-634-2...\$10.00

1. INTRODUCTION

With the recent rise of powerful open-source development platforms also came the development of data mining environments that offered seamless connection between a wide range of tools. The most famous of these tools is the WEKA¹ [11] from the Waikato University’s Computer Science Machine Learning Group (see Figure 1). WEKA contains hundreds of data miners, has been widely used for teaching and experimentation, and won the the 2005 SIGKDD Data Mining and Knowledge Discovery Service Award. Other noteworthy tools are “R”², ORANGE³ (see Figure 2) and MATLAB⁴ (since our interest is in the ready availability of tools, the rest of this paper will ignore proprietary tools such as MATLAB).

Our complaint with these tools is the same issue raised by Ritthoff et al. [9]. Real-world data mining experiments (or applications) are more complex than running one algorithm. Rather, such experiments or applications require *intricate combinations* of a large number of tools that include data miners, data pre-processors and report regenerators. Ritthoff et al. argue (and we agree) that the standard interface of (say) WEKA does not support the rapid generation of these intricate combinations.

The need to “wire up” data miners within a multitude of other tools has been addressed in many ways. In WEKA’s visual *knowledge flow* programming environment, for example, nodes represent different pre-processors/ data miners/ etc, and arcs represent data flows between them. A similar visual environment is offered by many other tools including ORANGE (see Figure 2). The YALE tool (now RAPID-I) built by Ritthoff et al. formalize these visual environments by generating them from XML schema describing *operator trees* like Figure 3.

In our experience in CS, students find these visual environments either discouraging or distracting:

- Some are discouraged by the tool complexity. These students shy away from extensive modification and experimentation.
- Others are so enamored with the impressive software engineering inside these tools that they waste an entire semester building environments to support data mining, but never get around to the data mining itself.

A similar experience is reported by our colleagues in the

¹<http://www.cs.waikato.ac.nz/ml/weka/>

²<http://www.r-project.org/>

³<http://magix.fri.uni-lj.si/orange/>

⁴<http://www.mathworks.com>

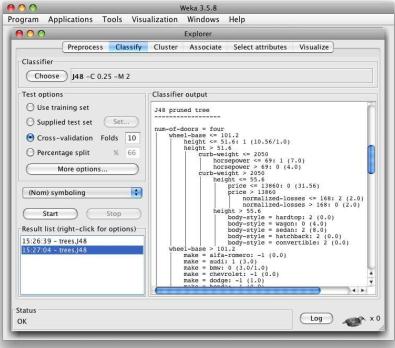


Figure 1: WEKA.

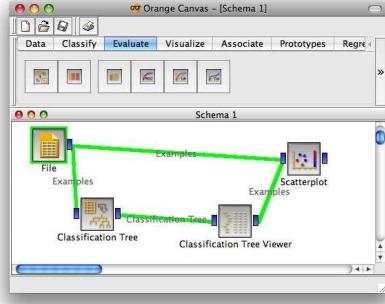


Figure 2: Orange.

WVU statistics department. In order to support the learning process, they bury these data mining tools inside high-level scripting environments. Their scripting environments shields the user from the complexities of “R” by defining some high-level LISP code that, internally, calls “R.”

Our experience with existing data mining toolkits is that these tools are not suitable for publishing or sharing repeatable experiments. This paper discusses why we believe this to be so and offers OURMINE as an alternate format for sharing executable experiments.

OURMINE is a data mining environment used at West Virginia University to teach data mining as well as to support the generation of our research papers. OURMINE does not replace WEKA, “R”, etc. Rather, it takes a UNIX shell scripting approach that allows for the “duct taping” together of various tools. For example, OURMINE currently calls WEKA learners as sub-routines, but it is not limited to that tool. This is no requirement to code in a single language (e.g. the JAVA used in WEKA) so data miners written in “C” can be run along side those written in JAVA or any other language. Any program with a command-line API can be combined with other programs within OURMINE. In this way, we can lever useful components from WEKA, R, etc, while quickly adding in scripts to handle any missing functionality.

Based on three years experience with OURMINE, we assert that the tool has certain pedagogical advantages. The simplicity of the tool encourages experimentation and mod-

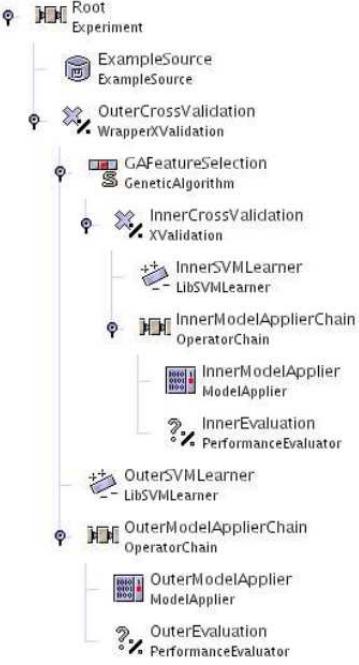


Figure 3: Data mining operator trees. From [7].

ification, even by data mining novices. We also argue that OURMINE is a productive data mining research environment. For example $\frac{9}{10}$ of the first and second authors’ recent papers used OURMINE.

More importantly, OURMINE offers a succinct executable representation of an experiment that can be included in a research paper. To demonstrate that, we present:

- The OURMINE script reproducing a recent SE experiment [10].
- The results obtained from that script.

The experiment selected is of particular importance to the PROMISE community since it shows that PROMISE data sets collected from one source can be applied elsewhere to create good defect predictors. This in turn means that the PROMISE repository is not just a useful academic resource, but is also a valuable source of data for industry. If a company wants to build defect predictors, but lacks the local data required to build those predictors, then by using the right *relevancy filters* (described below) it is possible to use data imported from the PROMISE repository to generate defect predictors nearly as good as those that might be built from local data.

2. OURMINE

While a powerful language, we find that LISP can be arcane to many audiences. Hence, OURMINE’s script are a combination of BASH [8] and GAWK [1]. Our choice of BASH/GAWK over, say, LISP is partially a matter of taste but we defend that selection as follows. Once a student learns RAPID-I’s XML configuration tricks, then those learned skills are highly specific to that particular tool. On the other hand, once a student learns BASH/GAWK meth-

```

#naive bayes classifier in gawk
#usage: gawk -F, -f nbc.awk Pass=1 train.csv Pass=2 test.csv

Pass==1 {train()}
Pass==2 {print $NF "|" classify()}

function train( i,h) {
    Total++;
    h=$NF;      # the hypothesis is in the last column
    H[h]++;     # remember how often we have seen "h"
    for(i=1;i<NF;i++) {
        if ($i=="?")
            continue;      # skip unknown values
        Freq[h,i,$i]++;
        if (++Seen[i,$i]==1)
            Attributes[i]++;
        # remember unique values
    }
}

function classify( i,temp,what,like,h) {
    like = -100000;      # smaller than any log
    for(h in H) {        # for every hypothesis, do...
        temp=log(H[h]/Total); # logs stop numeric errors
        for(i=1;i<NF;i++) {
            if ($i=="?")
                continue;      # skip unkwnon values
            temp += log((Freq[h,i,$i]+1)/(H[h]+Attributes[NF])) )
        if ( temp >= like ) { # we've found a better hypothesis
            like = temp
            what=h
        }
    }
    return what;
}

```

Figure 4: A Naive Bayes classifier for data sets in csv format where the last column is the class.

ods for data pre-processing and reporting, they can apply those scripting tricks to any number of future applications.

Another reason to prefer scripting over the complexity of RAPID-I, WEKA, “R”, etc, is that it reveals the inherent simplicity of many of our data mining methods. For example, Figure 4 shows a GAWK implementation of a Naive Bayes classifier for discrete data where the last column stores the class symbol. This tiny script is no mere toy- it successfully executes on very large datasets such as those seen in the 2001 KDD cup. WEKA, on the other hand, cannot process these large data sets as it always loads its data into RAM. Figure 4, on the other hand, only requires enough memory to store one instance as well as the frequency counts in the hash table “*F*”.

More importantly, in terms of teaching, Figure 4 is easily customizable. For example, Figure 5 shows four warm-up exercises for novice data miners that (a) introduce them to basic data mining concepts and (b) show them how easy it is to script their own data miner: Each of these tasks requires changes to fewer than 10 lines in Figure 4. The ease of these customizations fosters a spirit of “this is easy” for novice data miners. This, in turn, empowers them to design their own extensive and elaborate experiments.

2.1 Built-in Data/Functions

The appendix of this paper describes the download and install instructions for OURMINE. The standard install comes with the following public domain data sets (found in the \$OURMINE/lib/arffs directory):

- PROMISE: cm1, kc1, kc2, kc3, mc1, mc2, mw1, pc1, pc2, pc3, pc4, pc5, ar3, ar4, ar5

1. Modify Figure 4 so that there is no train/test data. Instead, make it an incremental learner. Hint: 1) call the functions *train*, then *classify* on every line of input. 2) The order is important: always *train* before *classifying* so the the results are always on unseen data.
2. Convert Figure 4 into HYPERPIPES [2]. Hint: 1) add globals *Max[h, i]* and *Min[h, i]* to keep the max/min values seen in every column “*i*” and every hypothesis class “*h*”. 2) Test instance belongs to the class that most overlaps the attributes in the test instance. So, for all attributes in the test set, sum the returned values from *contains1*:

```

function contains1(h,i,val,numerip) {
    return numerip ?
        Max[h,i] >= val && Min[h,i] <= val :
        (h,i,value) in Seen }

```

3. Use Figure 4 for anomaly detector. Hint: 1) make all training examples classify as the same class; 2) an anomalous test instance has a likelihood $\frac{1}{\alpha}$ of the mean likelihood seen during training (*alpha* needs tuning but *alpha* = 50 is often useful).
4. Using your solution to #1, create an incremental version of HYPERPIPES and an anomaly detector.

Figure 5: Four Introductory OURMINE programming exercises.

- UCI (discrete): anneal, colic, hepatitis, kr-vs-kp, mushroom, sick, waveform-5000, audiology, credit-a, glass, hypothyroid, labor, pcoic, sonar, vehicle, weather, auto, credit-g, heart-c, ionosphere, letter, primary-tumor, soybean, vote, weather.nominal, breast-cancer, diabetes, heart-h, iris, lymph, segment, splice, vowel
- UCI (numeric): auto93, basketball, cholesterol, detroit, fruitfly, longley, pbc, quake, sleep, autoHorse, bodyfat, cleveland, echoMonths, gascons, lowbwt, pharynx, schlvote, strike, autoMpg, bolts, cloud, elusage, housing, mbagrade, pollution, sensory, veteran, autoPrice, breastTumor, cpu, fishcatch, hungarian, meta, pwLinear, servo, vineyard

OURMINE also comes with a library of common functions used in machine learning experiments. All learners are the WEKA implementations. The functions that interface OURMINE with the WEKA set certain command-line parameters. To see those flag settings, type *show <learner name>* at the command prompt. That library includes:

- The data manipulation functions of Figure 6;
- The statistical functions of Figure 7;
- Some reporting functions such as code to auto-generate the latex files required to report quartile charts. For examples of those charts, see Figure 12 and Figure 13 (later in this article).
- Learners: oner, jRip, jRip10, part, aode, aode10, nb (naive bayes), nb10, nbk, lwl, j48, j4810, j4810c, lsr, m5p, 1Bkx, 1Bk, apriori

2.2 Extending the Functionality

OURMINE’s library contains the functions used by a team of 12 graduate students and one faculty student researching data mining over a three year period. This library is hardly complete and one of the goals of this paper is to create a community of OURMINE programmers so that the library

```

buildIncrementalSet builds a set containing a set number of
instances;
classes find all values for a class attribute;
combineFilesRandom combines two data sets, randomizing
the lines;
combineFiles combines two data sets with common attributes
gains runs InfoGain on the data
intersectAttributes finds the intersection of attributes over
several data sets;
logNumbers logs all numeric values in a data set ;
randomizeFiles randomizes lines in a data set;
rankViaInfoGain ranks attributes by InfoGain values;
removeAttributes performs column pruning;
sample creates an over or under-sampled dataset;
shared find shared attributes;
someArff splits a data set into train.arff and test.arff where
the test fail contains the  $B$ -th division of  $\frac{1}{B}$ -the data and
train file contains the rest.
some generates an ARFF file containing certain attributes;

```

Figure 6: OURMINE data manipulation functions.

of functions can be extended.

Adding new functionality to OURMINE is not difficult. OURMINE’s library of scripts are contained in a file called *minerc* which is just a collection of functions written in the BASH scripting language. Hence, to add functionality:

- Create a separate BASH file (i.e. *functions.sh*).
- Write all new functions in this file.
- At the end of *minerc*, add the line “. *functions.sh*”.

Very little knowledge of BASH is required to add functions to OURMINE. If the new function is actually in a JAVA file, the BASH function could just instantiate the Java file. In Figure 8, for example, the BASH function *nb* uses a series of commands to instantiate a JAVA classifier. The *blab* is a built-in function that prints text to the screen. The second line is the command to launch an external Java application (-Xmx1024M is a flag that gives the Java application a certain

abcd: finds a,b,c,d values and reports pd, pf, accuracy, precision, and balance. If $\{A, B, C, D\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

$$\begin{aligned}
 pd &= \text{recall} = \frac{D}{(B + D)} \\
 pf &= \frac{C}{(A + C)} \\
 \text{balance} &= 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}}
 \end{aligned}$$

columnStats: find column min, max, mean, and std.deviations.
medians: finds medians;
quartiles: generates box plots showing quartiles;
uniques: finds the unique entries in a column;
winLossTie: runs the Mann-Whitney test

Figure 7: OURMINE statistical functions.

```

nb() {
    blab "n"
    java -Xmx1024M -cp $Tmp/weka.jar \
        weka.classifiers.bayes.NaiveBayes \
        -p 0 -t $1 -T $2
}

```

Figure 8: OURMINE calling the WEKA

```

1 someData() {
2     cat $1 | gawk '
3     BEGIN           { IGNORECASE =1
4                     srand('$RANDOM')
5                     N='$2' }
6     gsub(/%.*/, "")'
7     /^[ \t]*$/      { next }
8     In              { Data[rand()]=$0 }
9     @data/          { In=1 }
10    /@/
11    END             { for(D in Data) {
12                     print Data[D]
13                     if ((--N) <= 0) exit }} '
14 }

```

Figure 9: SomeData: An OURMINE function

amount of memory). The process to call external programs is similar for any language.

The majority of the functions in OURMINE are BASH scripts that glue together GAWK code. GAWK is simple to learn and encourages short, easily modifiable, programs. R. Loui, an associate professor in Computer Science at Washington University in St. Louis, strongly supports the use of GAWK in his artificial intelligence classes. He writes:

There is no issue of user-interface. This forces the programmer to return to the question of what the program does, not how it looks. There is no time spent programming a binsort when the data can be shipped to /bin/sort in no time. [5]

For example, the GAWK script of Figure 9 extracts N randomly selected data lines from an arff file. It uses two parameters as input: $$1=input\ file$; $$2=N$ (how many lines of data we require).

- Lines 4 and 5 initializes random numbers and N .
- Lines 6 and 7 are the standard idiom for “skip lines that are all blanks or comments”. If, after deleting all characters after comment, the line contains only white space then we skip that line.
- Line 8 stores a data line at a random location in the *Data* array, but only if Line 9 ran before and recognized the start of the data section in this file (marked with “@data”).
- Line 10 skips all the meta lines in the file.
- Lines 10 through 13 print out N random data items.

2.3 Writing Experiments

The OURMINE user writes experiments in a BASH notation into a file loaded by *minerc*. Running an experiment is as simple as typing the name of its function at the OURMINE command prompt (i.e. if your function is called *exp1*, it will run when you type *exp1* at the command prompt).

```

1 demo15() {
2   cd $Tmp
3   (echo "#data,bin, a,b,c,d,acc,pd,pf,prec,bal"
4   seed=$RANDOM;
5   for((bin=1;bin<=10;bin++)); do
6     blab "$bin"
7     auto93discreteClass |
8       someArff --seed $seed --bins $Bins --bin $bin
9     nb train.arff test.arff |
10    gotwant |
11    abcd --goal "_20" --prefix "auto93,$bin" --decimals 1
12  done |
13  sort -t, -n -k 11,11
14  ) |
15  malign > demo15.csv
16  blabln "
17  echo ""; cat demo15.csv
18  cp demo15.csv $Safe/demo15.csv
19  cd $Here
20 }

```

Figure 10: An OURMINE experiment.

Figure 10 shows a basic experiment from OURMINE. When reading BASH scripts like Figure 10, one thing to note is that the pipe character “|” links the output of one process into the input of another. Also, a “|” at the end of line means that the next line inputs the piped output of the last line.

Line 5 of Figure 10 shows that this a 10-way cross-validation experiment, executed over one file (see line 7: auto93 from the UCI collection). At line 8, the *someArff* function splits the data into ”train.arff” (containing 90% of the data) and ”test.arff” (containing 10% of the data).

At line 9, a Naive Bayes learner of Figure 8 is called. A more general construct, seen in the experiment described below is to loop through, a set of learners using

for learner in \$learners; do

in which case, line 9 would become

\$learner train.arff test.arff

The code in lines 10 and 11 collect the performance statistics and print out *a, b, c, d, acc, pred, pf, prec, bal*. Some of these statistics are class-dependent. Line 11 shows the idiom *abcd -goal "_20"* which instructs OURMINE to only report statistics for the class ”_20”. If statistics for all classes are required, then the OURMINE idiom is to cache the result of the learner, then loop through all classes calling *abcd* on each class in turn. When using the OURMINE tools, that idiom can be see in lines 20 through 29 of Figure 11.

The sort function of line 13 of Figure 8 sorts these results on accuracy (column 11) and the results are saved to a safety box *\$Safe*. This final saving step is required because, by default, OURMINE cleans up after itself as it exits. Unless result files are saved to some safe place, they will be wiped.

This is a simple experiment, and it should be fairly easy to see the flow between statements. If the settings used for the parameters of a function are confusing, students learning OURMINE can run that function separately at the BASH prompt or look at its source code using the *show <function name>* command within OURMINE.

One nuance, not readily apparent in Figure 10, is that these scripts are naturally parallelizable. The idiom

ssh me@someMachine.edu runThis

logs into a machine and runs the script *runThis*. The script executes in some default, rather restricted, environment so *runThis* must now how to create the paths and temp directories needed for execution. OURMINE is designed to automatically create its execution environment so it is easy to write *runThis* scripts based on OURMINE. In a standard university environment, where students have the same password on multiple machines, then very large experiments can be farmed out over those CPUs by running:

```

ssh me@someMachine1.edu runThis1
ssh me@someMachine2.edu runThis2
ssh me@someMachine3.edu runThis3
etc

```

This lets researchers complete lengthy data mining experiments in minutes to hours, rather than days to weeks.

3. OURMINE & RESEARCH

To illustrate the use of OURMINE, the rest of this paper uses the tool to reproduce a recent *Empirical Software Engineering* paper written by the second and fourth authors [10]. Note that the following code was written by the first author with minimal assistance from the others: for the most part, the first author worked straight from the text of [10].

In their paper, Turhan et al. [10] focus on binary defect prediction (defective or non-defective) and perform three experiments to reach a conclusion in favor of either:

- Cross-company (CC) data imported from other sites;
- or within-company (WC) data collected locally.

That paper made several conclusions:

- **Turhan#1:** Using local WC data produces significantly better defect predictors than using imported CC data.
- **Turhan#2:** However, when *relevancy filtering* (see below) is applied to the CC data, then the imported CC data leads to defect predictors nearly as effective as using local WC data.
- **Turhan#3:** Hence, while local data is the preferred option, it is feasible to use imported data provided that it is selected by a relevancy filter.

This experiment is of great importance to the PROMISE community. It shows that PROMISE data sets collected from one source can be applied elsewhere to learn good defect predictors; i.e. the PROMISE repository is not just a useful academic resource, but is also a valuable source of data for industry. Companies wishing to build defect predictors can do so, even if they lack the local data required to build those predictors.

Such an important result needs confirmation. Tools like OURMINE are useful for such replication studies.

3.1 Experiment

Many research papers in the field of data mining present results but bury the exact details of the implementation. After years of experience, it is clear that the majority of time is spent in the tedium of pre and post-processing. The main contribution of Ourmine is to provide functions to handle these tasks. The script shown in Figure 11 is a complete structured experiment that is run in the Ourmine environment. What is not shown are the highly customized tools

that are frequently built and thrown away to deal with tasks like splitting a data set into training and test subsets. An Ourmine function, *someArff* does this for the researcher. Many of the lines in this figure are actually references to internal Ourmine functions.

As a preliminary to this experiment, we took seven PROMISE defect data sets (PC1,CM1,KC1,KC2,KC3,MW1,MC2) and built seven combined data sets, each containing $\frac{6}{7}$ -th of the data. For example, the file:

```
$OURMINE/lib/arffs/mdp/combined_PC1.arff
```

contains all seven data sets *except* PC1. This is the training

```

1 an2() {
2     local me="promise_an2"
3     local bins=10
4     local repeats=10
5     local learners="nb lwl"
6     local datas="PC1 CM1 KC1 KC2 KC3 MW1 MC2"
7     cd $Tmp
8     (echo "#data,repeat,bin,treatment,learner,goal,a,b,c,d \
9      ,acc,pd,pf,prec,bal"
10    for((r=1;r<=$repeats;r++)); do
11        for data in $datas; do
12            arff=$OURMINE/lib/arffs/mdp/shared_$data.arff
13            combined=$OURMINE/lib/arffs/mdp/combined_$data.arff
14            blab "data=$data repeat=$r "
15            seed=$RANDOM;
16
17            for((bin=1;bin<=$bins;bin++)); do
18                blab "$bin"
19                cat $arff |
20                someArff --seed $seed --bins $bins --bin $bin
21                goals='cat $arff | classes --brief'
22                for learner in $learners; do
23                    $learner train.arff test.arff |
24                    gotwant > results.dat
25                    for goal in $goals; do
26                        cat results.dat |
27                        abcd --goal "$goal" \
28                        --prefix "$data,$r,$bin,WC,$learner,$goal" \
29                        --decimals 1
30
31                done
32                goals='cat $arff | classes --brief'
33                blab "CC"
34                local N=$((RANDOM % 10 + 1)
35                blab "($N)"
36                cat $arff |
37                someArff --seed $seed --bins $bins --bin $N
38                for learner in $learners; do
39                    makeTrainCombined $combined > trainCom.arff
40                    $learner trainCom.arff test.arff |
41                    gotwant > results.dat
42                    for goal in $goals; do
43                        cat results.dat |
44                        abcd --goal "$goal" \
45                        --prefix "$data,$r,CC,CC,$learner,$goal" \
46                        --decimals 1
47
48                done
49
50            done
51            blabln
52        done
53    done
54    ) | sort -t, -n -k 12,12 | malign > $me.csv #each bin
55    blabln "
56    echo ""; cat $me.csv
57    cp $me.csv $Safe/$me.csv
58    cd $Here
59 }
```

Figure 11: WC or CC defect prediction. From [10].

set used for the CC (cross-company) data.

Next, we conducted a 10*10-way cross validation experiment; i.e. 10 times we asked *someArff* to randomly sort each data set, then generate 10 train and test sets containing 90% and 10% of each data set. The 90% sets are the training set of the WC (within-company) data. Lines 17 to 31 of Figure 11 shows that standard cross-validation study.

The CC study is shown on lines 32 to 46 of Figure 11. This is the same as the WC study but this time the training set is one the of the *combined_X.arff* files described above.

The learners in this study were two Naive Bayes classifiers. *Nb* (from Figure 8) and Frank et al.'s locally weighted Bayes classifier called *lwl* [3]. Naive Bayes classifiers were used since:

- Previously [6], we have offered evidence that they do better than other commonly used learners;
- Lessmann et al. report that many other learners do no better than Naive Bayes on the PROMISE defect data sets [4].

Lwl applies *relevancy filtering* to the training set. The idea behind relevancy filtering is to collect similar instances together in order to construct a learning set that is homogeneous with the testing set. Specifically, the aim of this filter is to try to introduce a bias in the model by using training data that shares similar characteristics with the testing data. Turhan et al. implemented this with a combination of k-Nearest Neighbor (k-NN) method and Naive Bayes. For each test instance in the test set, the k=10 nearest neighbors in the training set are collected. Duplicates are removed, and the remaining examples are used to train a Naive Bayes Classifier.

For reasons of repeatability, this study did not use Turhan et al.'s relevancy filter. Rather, we used the *lwl* from the WEKA. For each test instance, *lwl* constructs a new naive Bayes model using a weighted subset of training instances in the locale of the test instance. In this approach, the size of the training subset is based on the distance of the kth nearest-neighbor to the instance being classified. The training instances in this neighborhood are weighted, with more weight assigned to the closest instances. An instance identical to the test instance is given a weight of one, with the weight decreasing linearly to zero as the distance increases. Higher values for the number of neighbors (*k*) will result in models that are less sensitive to data anomalies, while smaller models will conform more closely to the data. The authors caution against using too small of a value for *k*, as it will be sensitive to the noise in the data. At the suggestion of Frank et al. we ran Figure 11 using $k \in \{10, 25, 50, 100\}$.

3.2 Results

Our results, shown in Figure 12 and Figure 13, are divided into reports of probability of detection (*pd*) and probability of false alarms (*pf*). When a method uses *k* nearest neighbors, the *k* value is shown (in brackets). For example, the first line of Figure 12 reports WC data when *lwl* used 100 nearest neighbors.

The results are sorted, top to bottom, from best to worse (higher *pd* is better; lower *pf* is better). Quartile plots are shown on the right-hand side of each row. The black dot in those plots shows the median value and the two “arms” on either side of the median show the second and third quartile respectively. The three vertical bars on each quartile chart

mark the (0%,50%,100%) points.

Column one of each row shows the results of a Mann-Whitney test (95% confidence): learners are ranked by how many times they lose compared to every other learner (so the top-ranked learner loses the least). In Figure 12 and Figure 13, row i has a different rank to row $i+1$ if Mann-Whitney reports a statistically significant difference between them.

Several aspects of these results are noteworthy:

- Measured in terms of pd and pf , the learners are ranked the same. That is, methods that result in higher pd values also generate lower pf values. This result means that, in this experiment, we can make clear recommendations about the value of different learners.
- When performing relevancy filtering on CC data, extreme locality is not recommended. Observe how $k \in \{10, 25\}$ are ranked second and third worst in terms of both pd and pf .
- When using imported CC data, some degree of relevancy filtering is essential. The last line of our results shows the worst pd, pf results and on that line we see that Naive Bayes, using all the imported data, performs worst.
- The locally-weighted scheme used by *lwl* improved WC results as well as CC results. This implies that a certain level of noise still exists within local data sets and it is useful to remove extraneous factors from the local data.

These results confirm the Turhan et al. results:

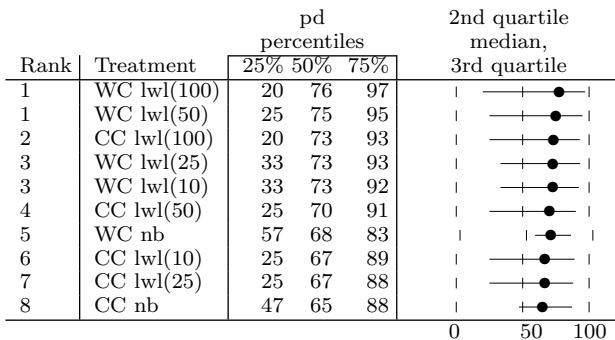


Figure 12: Probability of Detection (PD) results, sorted by median values.

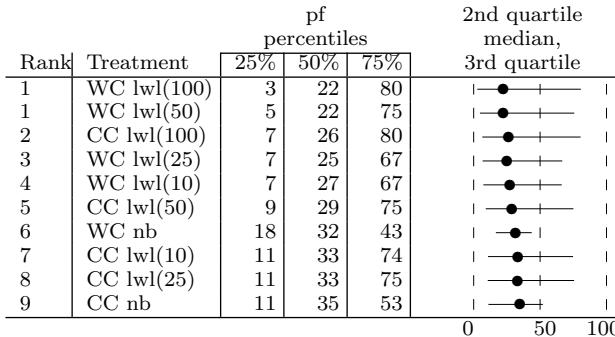


Figure 13: Probability of False Alarm (PF) result, sorted by median values.

- In a result consistent with **Turhan#1**, best results were seen using WC data (see the top two rows of each table of results).
- In a result consistent with **Turhan#2**, after relevancy filtering with $k = 100$, the CC results are nearly as good as the WC result: a loss of only 3% in the median pd and a loss of 4% in the median pf .

Our conclusions from this study are the same as **Turhan#3**: while local data is the preferred option, it is feasible to use imported data provided it is selected by a relevancy filter. If a company has a large collection of local development data, they should use that to develop defect predictors. If no such local repository exists, then a cross-company data collection filtered by a locally-weighted classifier will yield usable results. Repositories like PROMISE can be used to obtain that CC data.

4. DISCUSSION

As Figure 14 illustrates, the replication for this experiment was completed within a matter of hours. Original research would undoubtedly take more time, but the tools in OURMINE clearly cut the necessary time to set up an experiment, or to reproduce an experiment, by a large amount.

Note that the slowest task in reproducing the experiment was the 200 minutes for *experiment alteration*; i.e. finding and fixing conceptual bugs in version i of the functions, then improve them with $i+1$. In the past, we considered this work wasted time since it seemed that we were just making dumb mistakes and correcting them. Now, we have a different view. It is this process of misunderstanding, then correcting the details of an experiment, that is the real point of the experimentation:

- If repeating an experiment takes too long, we may abandon the exercise and learn nothing from the process.
- If repeating an experiment is too easy (e.g. just editing a RAPID-I operator tree) then repeating the experiment does little to change our views on software engineering and data mining.
- But if repeating the experiment takes the *right* amount of time (not too long, not too short), and it challenges just enough of our understanding of some topic, then we find that we “engage” with the problem; i.e, we think hard about:
 - The base assumptions of the experiment;
 - The interesting properties of the data;
 - and the alternative methods that could achieve the same or better results.

That is, we advocate OURMINE and the ugly syntax of Figure 11 over beautiful visual programming environments like WEKA’s knowledge flow or the RAPID-I operator tree

Step	Time spent (minutes)
Prep work	38
Data manipulation	152
Coding	163
Experiment alteration (bugs, rework)	210
Total	563 \approx 10 hours

Figure 14: Time spent on each step

because the visual environments are too easy to use (so we are not forced into learning new ideas).

On the other hand, we still want a productive development environment for developing our experiments. It should be possible to update a data mining environment and get new insights from the tools, fast enough to keep us motivated and excited to work on the task. In our experience, the high-level scripting of BASH and GAWK lets us find that middle ground between too awkward and too easy.

5. CONCLUSIONS

The mantra of the PROMISE series is “repeatable, improvable, maybe refutable” software engineering experiments. This community has successfully created a library of reusable software engineering data sets that is growing in size and which have been used in multiple PROMISE papers:

- 2006: 23 data sets
- 2007: 40 data sets
- 2008: 67 data sets
- 2009: 85 data sets (at the time of this writing)

The next challenge in the PROMISE community will be to not only share data, but to share experiments; i.e. the PROMISE repository should grow to include not just data, but the code required to run experiments over that data. We look forward to the day when it is routine for PROMISE submissions to come not just with supporting data but also with a full executable version of the experimental rig used in the paper. This would simplify the goal of letting other researchers repeat, and maybe even improve, the experimental results of others.

In the paper, we have used OURMINE to reproduce an important result:

- It is best to use local data to build local defect predictors;
- However, if imported data is selected using a *relevancy filtering*, then....
- ... imported data can build defect predictors that function nearly as well as those built from local data.

This means that repositories like PROMISE are actually an important source of data for industrial applications.

We prefer OURMINE to other tools. For example, four features of Figure 8 and Figure 11 are worthy of mention:

1. OURMINE is very succinct, a few lines can describe even complex experiments.
2. OURMINE’s experimental descriptions are complete. There is nothing hidden in Figure 11; it is not the pseudocode of the our experiments, it is the experiment.
3. OURMINE code is executable and can be executed by other researchers directly.
4. Lastly, the execution environment of OURMINE is readily available. Many machines already have the support tools required for OURMINE. For example, we have run OURMINE on Linux, Mac, and Windows machines (with Cygwin installed).

Like Ritthol et al., we doubt that the standard interfaces of tools like WEKA, etc, are adequate for representing the space of possible experiments. Impressive visual programming environments are not the answer: their sophistication can either distract or discourage novice data miners

from extensive modification and experimentation. Also, we find that the functionality of the visual environment can be achieved with little BASH and GAWK scripts, with a fraction of the development effort and a greatly increased chance that novices will modify the environment.

OURMINE is hence a candidate format for sharing descriptions of experiments. The PROMISE community might find this format unacceptable but discussions about the drawbacks (or strengths) of OURMINE would help evolve not just OURMINE, but also the discussion on how to represent data mining experiments for software engineering.

6. REFERENCES

- [1] Brian W. Kernighan Alfred V. Aho and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
- [2] Jacob Eisenstein and Randall Davis. Visual and linguistic information in gesture classification. In *ICMI*, pages 113–120, 2004. Available from <http://icicle.googlecode.com/svn/trunk/share/pdf/eisenstein04.pdf>.
- [3] Eibe Frank, Mark Hall, and Bernhard Pfahringer. Locally weighted naive bayes. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 249–256. Morgan Kaufmann, 2003.
- [4] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, May 2008.
- [5] R. Loui. Gawk for ai. *Class Lecture*. Available from <http://menzies.us/cs591o/?lecture=gawk>.
- [6] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from <http://menzies.us/pdf/06learnPredict.pdf>.
- [7] I. Mierswa, M. Wurst, and R. Klinkenberg. Yale: Rapid prototyping for complex data mining tasks. In *KDD’06*, 1996.
- [8] Chet Ramey. Bash, the bourne-again shell. 1994. Available from <http://tiswww.case.edu/php/chet/bash/rose94.pdf>.
- [9] O. Ritthoff, R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske. Yale: Yet another learning environment. In *LLWA 01 - Tagungsband der GI-Workshop-Woche, Dortmund, Germany*, pages 84–92, October 2001. Available from <http://ls2-www.cs.uni-dortmund.de/~fischer/publications/YaleLLWA01.pdf>.
- [10] Burak Turhan, Tim Menzies, Ayse B. Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 2009. Available from <http://menzies.us/pdf/08ccwc.pdf>.
- [11] Ian H. Witten and Eibe Frank. *Data mining. 2nd edition*. Morgan Kaufmann, Los Altos, US, 2005.

APPENDIX

Installing OURMINE

OURMINE is an open source tool licensed under GPL 3.0. It can be downloaded from <http://unbox.org/wisp/trunk/>

`our/INSTALL`.

As OURMINE is a command-line tool, the system requirements are insignificant. However, there are a few things that are necessary before installing OURMINE.

- A Unix-based platform. OURMINE is designed to work in the BASH shell, and will not operate on a Windows system. If no other platform is available, a BASH emulator like Cygwin will need to be installed before using OURMINE. Users running any Linux distribution, BSD, or Mac OS X can run OURMINE natively.
- The Java Runtime Environment. Most computers will already have this installed. The ability to run Java programs is required for the WEKA learners.
- The GAWK Programming Language. Many of the scripts within OURMINE are written using GAWK. Any up-to-date Linux system will already have GAWK installed. Cygwin or Mac OS X users will need to install it themselves.

To install OURMINE, follow these instructions:

- Go to a temporary directory
- wget -q -O INSTALL
<http://unbox.org/wisp/trunk/our/INSTALL>
- bash INSTALL

OURMINE is installed to the `$HOME/opt` directory by default. To run OURMINE, simply move into that directory and type `bash our minerc`. This will launch the OURMINE shell. OURMINE has several demos included to familiarize you with the built-in functionality. These demos may be run by typing `demoX` into the command prompt, where `X` is a number between 3 and 19. To look at the source code for that demo, type `show demoX`. It is highly recommended that new users run each demo and take a close look at its source code. This `show` command works for any function in OURMINE, not just the demos.

On the use of Relevance Feedback in IR-based Concept Location

Gregory Gay¹, Sonia Haiduc², Andrian Marcus², Tim Menzies¹

¹Lane Department of Computer Science,
West Virginia University
Morgantown, WV, USA

²Department of Computer Science
Wayne State University
Detroit, MI, USA

gregoryg@csee.wvu.edu; sonja@wayne.edu; amarcus@wayne.edu; tim@menzies.us

Abstract

Concept location is a critical activity during software evolution as it produces the location where a change is to start in response to a modification request, such as, a bug report or a new feature request. Lexical based concept location techniques rely on matching the text embedded in the source code to queries formulated by the developers. The efficiency of such techniques is strongly dependent on the ability of the developer to write good queries. We propose an approach to augment information retrieval (IR) based concept location via an explicit relevance feedback (RF) mechanism. RF is a two-part process in which the developer judges existing results returned by a search and the IR system uses this information to perform a new search, returning more relevant information to the user. A set of case studies performed on open source software systems reveals the impact of RF on the IR based concept location.

1. Introduction

Biggerstaff et al. [3] defined the *concept assignment problem* as "... discovering human oriented concepts and assigning them to their implementation instances within a program ...". The problem has been rephrased in the research community in the past decades as *concept location* in software. The redefinition relates it to the problem of *feature location* in software [28], as *features* are regarded as (a set of) concepts associated with a software system's functional requirements, reflected in user visible functionality. Biggerstaff et al.'s early definition of the problem needs to be instantiated for practical use. It needs a well defined context, which in turn helps define the operational goal and scope of concept location. The context is defined by establishing the software engineering task supported by concept location, such as, change management, fault localization, traceability link recovery, etc. This in turn helps define the input and output parameters for concept location: how are the human oriented concepts

described and what are the "implementation instances within a program" referred to by Biggerstaff? For example, Wilde et al.'s approach "is not expected to cover all possible functionalities nor always to discover all code segments associated with a functionality. It is only intended to provide starting points for more detailed exploration of the code" [28].

In our research, we define concept location in the context of software change. The software change process [24] starts with a modification request and ends with a set of changes to the existing code and addition of new code. The software maintainer undertakes a set of activities to determine the parts of the software that need to be changed: concept location, impact analysis, change propagation, and refactoring. *Concept location starts with the change request and ends when the developer finds the location in the source code where the first change must be made (e.g., a class or a method).* The other activities start with the result of concept location and establish the extent of the change.

In our operating context (i.e., software change), the developers must decide where in the code they will start the change, hence their involvement is critical. The challenge here is to make sure the tools and methodologies work equally well for users with a wide range of expertise and abilities.

Marcus et al. [20] proposed an Information Retrieval (IR) based approach to concept location. The idea of the approach is to treat source code as a test corpus and use IR methods to index the corpus and build a search engine, which allows developers to search the source code much like they search other source of digital information (e.g., the internet). The methodology was further refined in [22] and also combined with other feature location techniques [10, 14, 16, 21, 29]. This family of approaches relies on the user to formulate a query and if she does not identify fast enough the location of the change, then she rewrites the query and restarts the search.

Two issues remained constant in all these methods:
(1) each approach is highly sensitive to the ability of

the user to write good queries; and (2) the knowledge gained by the user during the location process is not captured explicitly. Specifically, developers start the process from a change request and they either use it as is as a query or they extract a set of words from the change request and use them as a query. Extracting a good query from a change request depends on the experience of the developer and on her knowledge of the system. Previous work showed that developers tend to write queries with significantly different performance starting from the same change request [14]. As the developer investigates the results of the first search, she learns more about the system and eventually decides to improve the query by adding or removing words. There is a gap between the source code representation as classes and methods (or other decomposition units) and the words in a query and some developers can fill this gap easier than others.

In this paper we propose and evaluate an approach aimed at addressing these two issues, based on explicit relevance feedback (RF) provided by the user. We call the approach IRRF (Information Retrieval with Relevance Feedback) based concept location. The approach is motivated by similar work on traceability [8, 11] in software.

We present a case study where we explore under what circumstances IRRF improves the classic IR based concept location. We reenact changes associated with several bug fixes in three open source projects.

2. Concept location and relevance feedback

Concept location is sometimes regarded as an instance of an information seeking activity, where developers search and browse the source code to find the location to start the change. Different tools target the searching or the navigation aspects of the process. Regardless of the tool support, most concept location methodologies are interactive and iterative, as they require the user to decide whether a certain element of the source code, recommended by a tool, is relevant or not to the change. These decisions affect subsequent steps in the process (i.e., navigation or new search).

Our new concept location methodology combines the IR based concept location [20] with explicit RF from the user, which is used to formulate new queries. This section describes each technology and the proposed combination.

2.1. IR based concept location

Lexical based static concept location considers source code a text corpus and leverages the information encoded in identifiers and comments from the source code to guide the search. As such, it can be seen as a classic IR problem: given a document

collection and a query, determine those documents from the collection that are relevant to the query. Given that relevance is defined with respect to a textual query, user involvement is necessary to convert this relevance measure into a change related decision.

IR based concept location, proposed in [20], is based on the above ideas. It implies the use of an IR method to index the corpus extracted from the software and uses the index to compute a similarity measure between the document and a query. It is composed of five major steps, which may be instantiated differently based on the type of IR method used and how the corpus is created:

1. *Corpus creation.* The source code is parsed using a developer-defined granularity level (i.e., methods or classes) and documents are extracted from the source code. Each method (or class) will have a corresponding document in the corpus. Natural language processing (NLP) techniques and other filtering techniques can be applied to the corpus.
2. *Indexing.* The corpus is indexed using the IR method and a mathematical representation of the corpus is created. Each document (hence each method or class) has a corresponding index.
3. *Query formulation.* A developer selects a set of words that describe the concept to be located. This set of words constitutes the query. The tool checks whether the words from the query are present in the vocabulary of the source code. If a word is not present, then the tool eliminates the word from the initial query. If filtering or NLP was used in the corpus creation, the query will get the same treatment.
4. *Ranking documents.* Similarities between the query and every document from the source code are computed. The similarity measure depends on the IR method used. Based on these measures the documents in the corpus are ranked with respect to the query.
5. *Results examination.* The developer examines the ranked list of source code documents, starting with documents with highest similarities. For every source code document examined, a decision is required whether the document will be changed or not. If it will be changed, then the search succeeded and concept location ends. Else, if new knowledge obtained from the investigated documents helps formulate a better query (e.g., narrow down the search criteria), then step 3 should be reapplied, else the next document in the list should be examined.

2.2. Relevance feedback in IR

Relevance feedback analysis is a technique to utilize user input to improve the performance of retrieval algorithms. Relevance feedback has been one of the successes of information retrieval research for the past 30 years [17]. For example, the Text Retrieval

Conference¹ (co-sponsored by the National Institute of Standards and Technology - NIST and the U.S. Department of Defense) has a relevance feedback track. While the applications of relevance feedback and type of user input to relevance feedback have changed over the years, the actual algorithms have not changed much. Most algorithms are either pure statistical word based, or are domain dependent. There is no general agreement of what the best RF approach is, or what the relative benefits and costs of the various approaches are. In part, that is because RF is hard to study, evaluate, and compare. It is difficult to separate out the effects of an initial retrieval run, the decision procedure to determine what documents will be looked at, the user dependent relevance judgment procedure (including interface), and the actual RF reformulation algorithm. Our case study aims at evaluating only a subset of these aspects of RF.

There are three types of feedback: explicit, implicit, and blind ("pseudo") feedback. In our approach, we chose to implement an explicit RF mechanism. Explicit feedback is obtained from users by having them indicate the relevance of a document retrieved for a query. Users may indicate relevance explicitly using a binary or graded relevance system. Binary relevance feedback indicates that a document is either relevant or irrelevant for a given query. Graded relevance feedback indicates the relevance of a document to a query on a scale using numbers, letters, or descriptions (such as "not relevant", "somewhat relevant", "relevant", or "very relevant").

Classic text retrieval applications of RF make several assumptions [17], which are not always true in the case of source code text and make the problem more challenging for us:

- The user has sufficient knowledge to formulate the initial query. This is not always the case when it comes to software, as developers might be unfamiliar with a software system or they might not have enough knowledge about a particular problem domain.
- There are patterns of term distribution in the relevant vs. non-relevant documents: (i) term distribution in relevant documents will be similar; (ii) term distribution in non-relevant documents will be different from that in relevant documents (i.e., similarities between relevant and non-relevant documents are small). There is no evidence so far that this is true for source code.

RF has some known limitations, some of which we are also faced with:

- It is often harder to understand why a particular document was retrieved after applying relevance

feedback. We found this to be quite true in the case of source code based corpora.

- It is easy to decrease effectiveness (i.e., one irrelevant word can undo the good caused by lots of good words). This is hard to judge in our case, but quite likely.
- Long queries are inefficient for a typical IR engine and we found that source code based corpora tends to increase query length significantly and rapidly.

2.3. IRRF based concept location

The IRRF based concept location combines the IR based concept location described in Section 2.1 with an explicit RF mechanism. The new concept location methodology is defined as follows:

1. *Corpus creation* – same as IR.
2. *Indexing* – same as IR.
3. *Query formulation* – same as IR.
4. *Ranking documents* – same as IR.
5. *Results examination*. The developer examines the top N documents in the ranked list of results. For every source code document examined, a decision is required whether the document will be changed or not. If it will be changed, then the search succeeded and concept location ends. Else, the user marks the document as relevant or irrelevant. After the N documents are marked a new query is automatically formulated and the methodology resumes at step 4. If several rounds of feedback do not result in reaching the result, then the query may be reformulated manually by the user and resume at step 4.

In order to provide tool support for the IRRF methodology, several options are available. There are several options on how to generate the corpus from the source code, such as: document granularity (e.g., method, class, etc.), identifier splitting (i.e., keep original identifier or not), stop word removal, keyword removal, stemming, and comments inclusion. Some of these options are programming language specific. Section 3 provides details on the options we used in the evaluation. Indexing can be done with a variety of IR methods. The initial query formulation can be done by the developer or automatically extracted from the change request (i.e., use the entire change request as the query).

2.3.1. Vector space model

The original IR based concept location technique [20] was built around Latent Semantic Indexing (LSI) [13], an advanced IR method. Researchers investigated the use of other IR techniques, such as Vector Space Models (VSM) [27], Latent Dirichlet Allocation (LDA), or Bayes classifiers for concept location and other related activities (e.g., traceability link recovery). So far, there is no clear winner among these techniques. In consequence, we decided to use

¹ <http://trec.nist.gov/>

here a VSM technique, implemented in Apache Lucene². Our choice is motivated also by the fact that traditional RF methods were developed specifically for this type of IR technique.

In VSM, a vector model of a document d is a vector of words weights that span over the number of words in the corpus. The weights for each word are computed based on the *term frequency* of that word in d and the *inverse document frequency*. The similarity between two documents is computed as the cosine between their corresponding vector models.

2.3.2. RF with Rocchio

There are several options to implement an RF mechanism. One of the most popular approaches is the Rocchio relevance feedback method [26], used in conjunction with a VSM indexing technique. We implemented our own version of Rocchio, which integrates with Apache Lucene and works in the following way. When analyzing the top ranked methods, the user is asked to judge the current method as relevant, irrelevant, or neutral to the current change task. Given a set of documents D_Q encompassed by query Q , let R_Q be the subset of relevant documents and I_Q the set of irrelevant documents to the query. The original query Q can be then transformed by adding terms from R_Q and removing terms from I_Q . This mechanism is meant to bring the query closer to the relevant documents and drive it away from the irrelevant documents in the vector space. The new query is formulated as follows:

$$Q' = \alpha Q + \frac{\beta}{|R_Q|} \sum_{d \in R_Q} d - \frac{\chi}{|I_Q|} \sum_{d \in I_Q} d \quad (1)$$

where α , β , and χ are weighting parameters and d represents a document and its associated vector. The relevance feedback can be given by the user in several feedback rounds and the query is updated after each round.

The Rocchio technique is recursive. Each round, a new query is generated based on the query generated in the previous round. The three constants α , β , and χ are provided so that a level of importance can be specified by the user for the initial query, the relevant documents and the irrelevant documents. According to [9], placing emphasis on the relevant documents may improve the recall (new relevant documents may be found) while emphasizing the irrelevant documents may affect precision (false positives may be removed). Joachims recommends weighting the positive information four times higher than the negative [12]. De Lucia et al. [8] advocates using $\alpha=1$, $\beta=0.75$, and $\chi=0.25$ (i.e., relevant documents are three times more important than irrelevant ones). Our implementation

follows a similar line of thought, setting values of $\alpha=1$, $\beta=0.5$, and $\chi=0.15$ for the three weighting parameters. We tried other sets of weights ($\alpha=1$, $\beta=0.75$, and $\chi=0.25$ and $\alpha=1$, $\beta=1$, and $\chi=1$), but the final set of weights seemed to yield the best performance.

This type of query expansion is still prone to noise as certain common, but unimportant, terms may be added to the query. To filter this noise and improve precision, the system used in this paper only allows terms to be added to the query if they appear in less than 25% of the corpus.

3. Case study

As mentioned before, we developed IRRF to address several issues we learned in our experience with concept location. IR based concept location assumes the developers read the source code and reformulate the query if they did not locate the target methods. We found that most programmers are good at judging whether a method is relevant to a change or not, but their ability to formulate a good natural language query based on their knowledge of the software varies quite a bit. Developers have a hard time deciding what is wrong with their previous query and how to make it better. IRRF eliminates this step in the process and allows developers to focus on what they know best (i.e., source code rather than queries) by reformulating the queries automatically. Earlier work on traceability [7, 11] showed that RF improves an IR task, but not in all cases. Our assumption is that IRRF improves the IR based concept location (i.e., reduces developer effort) and the goal of the case study we performed is to investigate under what circumstances this is true, given that our approach has a different context than the traceability work.

3.1. Methodology

The case study consists of the reenactment of past changes in open source software (i.e., we know which methods were modified in response to the change request). The modified methods form the *change set*, and we call these methods *target methods*. This methodology has been used in previous work on evaluating concept location techniques [14, 16, 21]. In our case study, one developer performed the concept location reenactment using IRRF, given a set of change requests. He has seven years of programming experience (five in Java) and he was not familiar with the source code used in the study. For each change request his task was to locate one of the methods from the change set, based on the following scenario:

- He starts by running a query based on the change request, called the *initial query*.
- If any one of the target methods is among the top 5 methods in the ranked list of results, then he stops and

² <http://lucene.apache.org/java/docs/>

selects another change request, as RF is not needed in this case (i.e., IR based concept location will reach the method fast enough).

- Else he provides RF in several rounds. In each round, the developer marks the N top ranked methods as being *relevant* or *irrelevant*. If he cannot judge the relevancy of a method, the he marks the document as *neutral* and proceeds to the next document, increasing the size of the set of marked methods set by one.
- IRRF automatically reformulates the query based on the feedback provided by the developer and another round of feedback begins. We keep track of the number of methods marked by the developer.
- After each query is run, based on the positions of the target methods in the ranked list of search results and on the number of methods marked, the following decisions are made:
 - a. If any of the target methods is located in the top N documents, then STOP; consider IRRF successful and a target method found.
 - b. If for two consecutive feedback rounds the positions of the target methods declined in the ranked list of results, then STOP; consider that IRRF failed (i.e., the developer needs to reformulate the query manually).
 - c. If more than 50 methods were marked by the developer, then STOP; consider that IRRF failed (i.e., the developer needs to reformulate the query manually).

The values used for N vary and the performance of IRRF depends on it. The most commonly used values range from 1 to 10. We investigated the results of IRRF for three values of N : 1, 3, and 5, which are recommended values in recent studies for presenting lists of results to developers for investigation [25]. Each reenactment was done three times by the developer, the difference from case to case was the number N of marked methods in one round.

3.1.1. Data - software and change sets

We chose as the objects of the case study three open source systems: Eclipse³, JEdit⁴, and Adempiere⁵. All three systems have an active community and a rich history of changes. They all have online bug tracking systems, where bugs are reported and patches are submitted for review.

Eclipse is an integrated development environment developed in Java. For our case study, we considered version 2.0 of the system, which has approximately 2.5 millions lines of code and 7,500 classes. JEdit is an editor developed for programmers and it comes with a series of plugins which add extra features to its core

functionality. It is developed in Java and version 4.2 used in this case study has approximately 300,000 lines of code and 750 classes. Adempiere is a commons-based peer-production of open source enterprise resource planning applications. It is developed in Java and it has approximately 330,000 lines of code and 1,900 classes in version 3.1.0, which was used in our case study.

We used the history of a software system in order to extract real change sets from the source code. Specifically, we used approved patches of documented bugs for extracting the change sets. The bug descriptions are considered to be the change requests. This approach has been used in previous work on evaluating concept location techniques [14, 16, 21]. Some changes involve the addition of new methods. We do not, however, include these methods in the change sets, as they did not exist in the version that a developer would need to investigate in order to find the place to implement the change. For each of the systems, we analyzed their online defect tracking systems and manually selected a set of bugs to extract change sets for our case study.

The Eclipse community uses the open-source bug tracking system BugZilla⁶ to keep track of bugs in the system. Each bug has an associated bug report, which consists of several sections, one of which is the bug description. Sometimes the patches used to fix the bugs are also contained in the bug report, as attachments. They are usually in the form of *diff* files, containing the lines of code that changed between the version of the software where the bug was reported and the version where the bug was fixed. For our case study, we chose an initial set of 10 bugs reported in version 2.0 of the system, for which the patches were available in their bug reports.

For jEdit⁷ and Adempiere⁸, we analyzed the bug tracking systems hosted on the projects' sourceforge.net website. Both projects have systems that keep track of the patches submitted for known bugs in the source code. In these trackers, each patch has an associated report where the changes implemented in the patch are described in a *diff* file attached to the report. We selected for each system 10 initial patches for which a good description of the bug fixed by the patch was available, either in the description of the patch or in a separate bug report. All the patches we selected for jEdit were submitted and their corresponding bugs reported after version 4.2 of the system was released. For Adempiere patches were selected after the release of version 3.1.0.

³ <http://www.eclipse.org/>

⁴ <http://www.jedit.org/>

⁵ <http://www.adempiere.com/>

⁶ <https://bugs.eclipse.org/bugs/>

⁷ http://sourceforge.net/tracker/?group_id=588&atid=300588

⁸ http://sourceforge.net/tracker/?atid=879334&group_id=176962

Based on the patches reported for the three systems, we constructed the 10 change sets for each system. All change sets contained between one and six target methods.

3.1.2. Corpus creation

We extracted a corpus for each of the three systems. We used the version of the software in which the bugs chosen in the previous step were reported. We mapped each method in the source code to a document in our corpus. The Eclipse corpus has 74,996 documents, the JEdit corpus has 5,366 documents, and the Adempiere corpus has 28,622 documents. By comparison, the size of the corpora used in previous work on RF in traceability [6, 11] is in the few hundreds of documents range.

The corpora were built in the following manner:

- We extracted the methods using the Eclipse built in parser. The comments and identifiers from each method implementation were extracted.
- The identifiers were split according to common naming conventions. For example, “`setValue`”, “`set_value`”, “`SETvalue`”, etc. are all split to “`set`” and “`value`”. We kept the original identifiers in the corpus, which would favor any query containing an identifier already known by the user.
- We filtered out programming language specific keywords, as well as common English stop words⁹.
- We used the Porter stemmer¹⁰ in order to map different forms of the same lexeme to a common root.

3.1.3. Query formulation

As mentioned before, the goal of IRRF is to allow the developer not to write manually defined queries. Hence, in the case study the developer used as the initial query the bug description and bug title contained in the bug or patch reports (i.e., he copied the bug description and title). However, we eliminated any details referring to the implementation of the bug fix contained in these descriptions, prior to the study. The query was then automatically transformed following the same steps as the corpus (i.e., identifier splitting, stop word removal, stemming).

3.1.4. Assessment

Tool support in concept location is geared towards reducing developers’ effort in finding the starting point of a change. Previous work on concept location [14, 16, 21] defined and used as an efficiency measure the number of source code documents that the user has to investigate before locating the point of change. We use here the same measure with an added advantage. In previous work the cost for (re)formulating a query was never considered in evaluation (i.e., assumed to be

zero). In our case, the cost of (re)formulating a query is indeed almost zero, as the initial query is copied from the bug description and title, whereas subsequent queries are formulated automatically. The number of methods investigated is automatically tracked as they are explicitly marked by the developer.

In order to avoid the bias and the cost associated with user formulated queries, we do not consider manual query formulation as part of the methodology (see the first part of this section). We eliminate this step from both methodologies (i.e., IR and IRRS based concept location) for this case study.

For each change request, the base line is provided by the IR based concept location without query reformulation. The initial query is run and the baseline efficiency measure is the highest ranking (k) of any of the target methods. This means the user would have to investigate k methods to reach the target. For the IRRF case the efficiency measure is the number of methods marked one way or another (i.e., relevant, irrelevant, or neutral) before the target was found or until the method fails (see the methodology described above) plus the last rank of the target method. IRRF is considered to improve the baseline if its efficiency measure is lower than the baseline efficiency (i.e., fewer methods are investigated).

3.2. Results and discussion

The results presented and discussed here are aggregated and omit several intermediary steps from the study, such as: actual markings in each round for all cases, queries, intermediary ranking of target methods, complete change sets, etc. The complete data collected during the case study is available online at www.cs.wayne.edu/~severe/IRRf.

As explained in Section 3.1.1, we selected 10 changes for each system (i.e., 30 total). In 12 cases, at least one of the target methods was ranked in top 5 after the initial query, hence we did not use RF in those cases. **Error! Reference source not found.** aggregates the results of the concept location for the three systems considered, reflecting the changes for which we used RF: 7 for Eclipse, 6 for jEdit, and 5 for Adempiere.

The *Baseline* column shows the positions of the target methods in the result list after the initial query. The best rank in each case where there is more than one target method is bold. This is the efficiency measure in the baseline case (i.e., how many methods would the user need to investigate to find the best ranked target).

The IRRF columns show the positions of the target methods at the end of the IRRF location process, whether it succeeded or not (see Section 3.1 for details). N indicates the number of marked methods in

⁹ www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words

¹⁰ <http://tartarus.org/~martin/PorterStemmer/>

Table 1. Concept location results for Eclipse, jEdit and Adempiere

Eclipse					
No.	Defect Report#	Baseline	IRRF with N=1	IRRF with N=3	IRRF with N=5
1	Bug #13926	54	1 (16m/15r)	11 (51m/16r)	36 (50m/10r)
2	Bug #23140	17,42,47	99, 1 , 2 (9m/8r)	4, 1 , 2 (7m/3r)	6, 4 , 14 (9m/2r)
3	Bug #19691	1K+, 368, 531, 1K+, 108 , 139	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (2m/2r)	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (7m/2r)	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (11m/2r)
4	Bug #12118	9	1 (5m/5r)	1 (23m/8r)	4 (10m/2r)
5	Bug #17707	8	1 (2m/2r)	1 (4m/2r)	2 (7m/2r)
6	Bug #19686	428	448 (5m/5r)	3 (48m/16r)	5 (46m/9r)
7	Bug #21062	583,56	1K+, 781 (2m/2r)	604, 1 (37m/13r)	1K+, 1K+ (20m/4r)
jEdit					
1	Patch #1649033	40,87,22	70, 60, 50 (8m/7r)	39, 7, 42 (22m/7r)	30, 5 , 33 (26m/5r)
2	Patch # 1469996	296	1 (37m/36r)	289 (12m/4r)	5 (41m/9r)
3	Patch #1593900	7	1 (6m/4r)	1 (5m/2r)	1 (7m/2r)*
4	Patch # 1601830	47	216 (2m/2r)	242 (9m/3r)	146 (10m/2r)
5	Patch #1607211	354	98 (5m/5r)	3 (36m/12r)	3 (28m/6r)
6	Patch # 1275607	151	238 (4m/4r)	38 (48m/16r)	35 (50m/10r)
Adempiere					
1	Patch #1605419	15,550	1, 11 (8m/7r)	3, 109 (17m/5r)	1, 81 (12m/3r)
2	Patch #1599107	122	613 (6m/3r)	1K+ (8m/2r)	1K+ (12m/2r)
3	Patch #1599116	7	1 (3m/2r)	1 (5m/2r)	1 (7m/2r)*
4	Patch #1612136	58	141 (4m/3r)	1 (13m/5r)	1 (16m/4r)
5	Patch #1628050	52	1 (3m/3r)	2 (5m/2r)	2 (7m/2r)

Green – IRRF retrieves results more efficiently

Yellow – IRRF retrieves a better cumulative ranking of the target methods.

*IRRF performs as efficiently as the baseline

each round during IRRF. Note that only the methods for which relevance was given are counted as one of the N methods ranked in a feedback round (i.e., methods marked neutral are not counted towards the N , yet they count towards the efficiency measure). The number of methods analyzed by the developer before he stopped (i.e., the efficiency measure), either because a target method was found or because IRRF failed is reported in parenthesis (marked with m). This number includes all the methods marked by the developer in all the rounds of feedback, including also the methods marked as neutral, plus the rank of the target method in the final round. If one of the target methods was not ranked in the top 1,000 results, we denote its position as 1K+. To complete the picture, the number of feedback rounds is also reported in parenthesis (denoted with r), including the (incomplete) round when the target is found.

For example, row #2 in Eclipse, reads as follows. Baseline (**17**, 42, 47) means there are three target methods and the best ranked is on position 17. IRRF with $N=3$ (4, **1**, 2 (**7m/3r**)) means that one of the target methods (i.e., the second) was ranked #1 on the third round and the user marked a total of 7 methods to reach it (including the target method in the 3rd round).

The two numbers to compare here are: 17 in the baseline vs. 7 in the RF case. We consider that IRRF improves here and highlight the table cell with green. Cells marked with yellow show no improvement of IRRF, but they are interesting as the cumulative ranking of methods is better than in the baseline (the number of investigated methods needs to be added here to the ranks of the target methods for a proper comparison). White cells indicate cases where IRRF does not improve the baseline. The stars in the white cells indicate the cases when IRRF failed in finding the target method in a reasonable amount of steps.

The data reveals that IRRF brings improvement over the baseline in 13 of the 18 changes (i.e., at least one cell in the row is green). In 3 changes, the improvement is observed for all values of N (i.e., all three RF cells are green in these rows). RF with $N=1$ improved in 9 cases, RF with $N=3$ improved in 9 cases, and RF with $N=5$ improved in 8 cases, not all the same.

More specifically, in Eclipse for 6 out of the 7 change sets reported, IRRF retrieved one of the target methods more efficiently than the baseline. In jEdit, the ratio was 3 to 3, and in Adempiere IRRF performed better in 4 out of 5 cases. We did not observe a pattern of when one of the values of N performs better than the

other ones, nor about the magnitude of the IRRF improvement over the baseline. So, we can not formulate at this time rules such as “ $N=5$ is a better choice than $N=3$ or $N=1$ ”. Nor we can state that there is a correlation between the initial query and IRRF improvements.

One interesting phenomenon that we observed is that for one change set in jEdit and for one in Adempiere IRRF did not improve the efficiency of the baseline (based on our working definition), however it achieved a better cumulative ranking of the target methods. These two cases are marked with yellow in the table. We highlight these cases as we believe is still an indication that RF brings some added benefit in these situations.

Another interesting and rather unexpected phenomenon is that in some cases where the there are more target methods the baseline favors one of them, whereas IRRF helps retrieve another one faster. See Bug #23140 in Eclipse and Patch #1649033 in JEdit (the yellow cell). We do not speculate on this issue here, as it is orthogonal to the goal of the study, but it opens up an avenue for future research.

We identified cases when neither the ranking of the first target method, nor the cumulative ranking of IRRF was better than in the case of the baseline (i.e., all white rows in the table). Our initial assumption was “if the initial query is really poor, RF does not help much”. Unfortunately, this is not true as there were several cases where the initial query was worse, yet IRRF improved drastically (i.e., by one order of magnitude). For example, see Bug #19686 in Eclipse, Patch # 1469996, and Patch #1607211 in JEdit.

We investigated the cases with poor IRRF performance. For example, in the case of Bug #19691 in Eclipse, we found that the methods the developer would consider as being relevant based on the bug description would in fact not be relevant, even if they contained related terms from the bug description. The bug description is about exporting preferences for the team, whereas the target methods just contained “ignore” settings in the team preferences. This case highlights the difficulty of concept location in practice. Change requests are often formulated in terms different than the source code, both linguistically and logically. We can safely conclude that IRRF brings improvements over IR based concept location in many cases, but it is far from being a silver bullet.

3.3. Threats to validity

In our case study, we reenacted changes already performed in software systems and automated the process of query formulation. In practice, the user may reformulate the query along the way and IR may retrieve better results with the user re-formulated

query. Simply put, the case study approximates the situation when the developer is not good at writing queries. We argue that the opposite case does not need RF. In fact, as the results revealed, 12 of the 30 bug descriptions produced great initial queries. It is important to clearly establish the cases where explicit RF helps.

Our results are based on the feedback provided by only one user. Different people might give different feedback to IRRF.

We used only three values of N (i.e., 1, 3, and 5) in the case study and a single weighting scheme in the IRRF implementation. We are aware of the fact that other values used of N might retrieve different results. However, these values are within the range of values usually adopted in the implementation of explicit relevance feedback and represent a reasonable amount of information for a user to analyze in one round of feedback. The current set of weights used in our Rocchio implementation was chosen based on empirical evidence. Other weights could lead to slightly different results.

4. Related work

Approaches to concept location in software differ primarily on what type of information is used to guide the developer while searching the code. Dynamic techniques are based on the analysis of execution traces and focus on identifying features (i.e., concepts associated with user visible functionality of the system). Static techniques use the textual information embedded in source code (i.e., comments and identifiers) and/or structural information about the software (e.g., program dependencies). There are combined methods that use combinations of dynamic and static techniques. Given that IRRF is meant to augment lexical based static concept location, we will only refer to them in this section. A more comprehensive overview of concept location techniques is available in [21] and static techniques are discussed in [19].

Lexical base static concept location techniques rely on matching a user query to the text in the source code. Traditional searching methods are built using regular expression matching tools, such as grep. Such techniques limit user queries to be formulated as regular expressions and do not provide a ranking list of results, but rather a simple list of matches.

More sophisticated techniques rely on the use of IR methods and have the advantage (over regular expression matching) that allow the user to formulate natural language queries and the results are ranked. Marcus et al. [20] introduced such a technique, using LSI as the IR method. Our approach is based on this technique, as described in Section 2. The method was

later extended by using formal concept analysis to cluster the results of the search [22]. A different implementation of the method was done using Google Desktop Search [23], as the underlying IR engine. More recently, Lukins et al. proposed a variation of the LSI based technique, using LDA [16]. SNI AFL [29] is a related approach, which combines an IR technique with call graph information and falls under the category of combined concept location techniques.

Related to IR based concept location is work on traceability link recovery, impact analysis, and recommendation systems. We do not list here all these works, but rather explain how they relate and how they differ from our work. Software artifacts are represented in different formats and textual information is the common denominator for all of them, hence IR methods have become widely used in tools that support traceability link recovery [1, 7, 11, 15, 18] and in many recommendation systems [5]. In impact analysis [2, 4] the textual information indexed with the IR method is usually used in conjunction with structural information or historical information about changes. One of the main differences between these applications of IR vs. concept location is that they use queries extracted from different artifacts, rather than user written. This allows for automation in many cases and elimination of the user from the process. By definition these techniques retrieve usually sets of documents, hence evaluation is different than in the case of concept location.

Of particular interest to our work is the work of Hayes et al. [11] and De Lucia et al. [6, 8], which introduced RF in the context of traceability link recovery. At technology level, our work is similar to these approaches, as we use same IR methods and similar RF implementations. The difference is in the context, application, methodology, and evaluation. In [11] the problem is to reduce the number of false positives when high level requirements are traced to low level requirements. The context is the same in [8] except that several types of artifacts are considered there. In both cases, there are no user queries and the results consist of traceability matrices, which are manually evaluated. Basically, those approaches use RF to improve an essentially automated process. Due to the manual effort needed in the evaluation and availability of data, experimental data is restricted to corpora with hundreds of documents at best. In our application, we use corpora several order of magnitude larger, which is more typical for IR applications.

5. Conclusions and future work

We defined a new methodology for lexical based static concept location, which combines the IR based concept location with explicit relevance feedback. The

goal of the methodology is to alleviate the burden of query formulation on the developer.

Our case study revealed that in most cases RF reduces developer effort over the IR based concept location, in the absence of manual query reformulation. It also showed that in some cases, especially when the initial query is poor, RF does not really help. Our results are in line with previous work using RF [8, 11] in traceability link recovery. In each work RF was found to improve the performance in many cases, but not across the board in all cases. Although our context and corpora are different than the classic use of RF on text retrieval and in requirements traceability, this application is subject to some of the same limitations.

Future work will focus on specific issues. We will compare different IR methods (e.g., LSI, LDA, VSM, etc.) used in IRRF. We expect our results might be different than previous comparisons, where small corpora were used, which is often unsuitable for some statistical IR techniques. We will also experiment with more system, more change request, and different ways to build corpora. For example, we expect that if we eliminate the comments from the corpus, the queries will grow slower. On the other hand, information will be lost. We plan to analyze the trade-offs. Different weighting options in RF will be also investigated. Intuitively, favoring irrelevant documents over relevant documents should help in concept location, yet our current results support the opposite.

As far as the methodology is concerned we need to define a better heuristic that tells the user when to switch from RF to manual reformulation. During reenactment we knew the end result, so the heuristic we used in the experiment is based on this. In practice that would not work as is.

We focused here on studying methodologies rather than users. An obvious next step is to extend our research to study how developers perform RF. After all, concept location is a user driven activity.

6. References

- [1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, 28, 10, October 2002, pp. 970 - 983.
- [2] Antoniol, G., Canfora, G., Casazza, G., and Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: A Case Study", in *Proceedings 4th European Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, Feb 29 - March 03 2000, pp. 227-231.
- [3] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in *Proceedings 15th IEEE/ACM International Conference on Software Engineering (ICSE'94)* May 17-21 1993, pp. 482-498.

- [4] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories", in *Proceedings 11th IEEE International Symposium on Software Metrics (METRICS'05)*, September 19-22 2005, pp. 20-29.
- [5] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., "Hipikat: A Project Memory for Software Development", *IEEE Transactions on Software Engineering*, 31, 6, June 2005, pp. 446-465.
- [6] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?", in *Proceedings 14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006, pp. 307-316.
- [7] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *ACM Transactions on Software Engineering and Methodology*, 16, 4, 2007
- [8] De Lucia, A., Oliveto, R., and Sgueglia, P., "Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, Pennsylvania, 2006, pp. 299-309.
- [9] Dekhtyar, A., Hayes, J. H., and Larsen, J., "Make the Most of Your Time: How Should the Analyst Work with Automated Traceability Tools?", in *Proceedings 3rd International Workshop on Predictor Models in Software Engineering*, Minneapolis, MN, May 20 2007
- [10] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis", in *Proceedings 17th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 53-62.
- [11] Hayes, J. H., Dekhtyar, A., and Sundaram, S. K., "Advancing candidate link generation for requirements tracing: the study of methods", *IEEE Transactions on Software Engineering*, 32, 1, January 2006, pp. 4-19.
- [12] Joachims, T., "A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization", in *Proceedings 14th International Conference on Machine Learning*, 1997, pp. 143-151.
- [13] Landauer, T. K., Foltz, P. W., and Laham, D., "An Introduction to Latent Semantic Analysis", *Discourse Processes*, 25, 2&3, 1998, pp. 259-284.
- [14] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in *Proceedings 22nd IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, GA, November 5-9 2007, pp. 234-243.
- [15] Lormans, M. and Van Deursen, A., "Can LSI help Reconstructing Requirements Traceability in Design and Test?", in *Proceedings 10th European Conference on Software Maintenance and Reengineering*, 2006 pp. 47-56.
- [16] Lukins, S. K., Kraft, N. A., and Etzkorn, L., "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation", in *15th IEEE Working Conference on Reverse Engineering*, Antwerp, Belgium, 2008, pp. 155-164.
- [17] Manning, C. D., Raghavan, P., and Schütze, H., *Introduction to Information Retrieval*, Cambridge University Press, 2008.
- [18] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *International Journal of Software Engineering and Knowledge Engineering*, 15, 4, Oct. 2005, pp. 811-836.
- [19] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in *Proceedings 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 33-42.
- [20] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in *Proceedings 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, Delft, The Netherlands, November 9-12 2004, pp. 214-223.
- [21] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, 33, 6, June 2007, pp. 420-432.
- [22] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in *Proceedings 15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, June 2007, pp. 37-48.
- [23] Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., and Liu, D., "Source Code Exploration with Google ", in *Proceedings 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, 2006, pp. 334 - 338.
- [24] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", in *IEEE Software*, 2004, pp. 2-9.
- [25] Robillard, M., "Topology Analysis of Software dependencies", *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 17, 4, 2008, pp. 18-53.
- [26] Rocchio, J. J., "Relevance feedback in information retrieval", in *The SMART Retrieval System - Experiments in Automatic Document Processing*, Prentice Hall, 1971, pp. 313-323.
- [27] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.
- [28] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., "Locating User Functionality in Old Code", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'92)*, Orlando, FL, November 1992, pp. 200-205.
- [29] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNI AFL: Towards a Static Non-interactive Approach to Feature Location", *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 15, 2, 2006, pp. 195-226.

A Baseline Method For Search-Based Software Engineering

Gregory Gay
University of Minnesota
Minneapolis, MN
greg@greggay.com

ABSTRACT

Background: Search-based Software Engineering (SBSE) uses a variety of techniques such as evolutionary algorithms or meta-heuristic searches but lacks a standard baseline method.

Aims: The KEYS2 algorithm meets the criteria of a baseline. It is fast, stable, easy to understand, and presents results that are competitive with standard techniques.

Method: KEYS2 operates on the theory that a small subset of variables control the majority of the search space. It uses a greedy search and a Bayesian ranking heuristic to fix the values of these variables, which rapidly forces the search towards stable high-scoring areas.

Results: KEYS2 is faster than standard techniques, presents competitive results (assessed with a rank-sum test), and offers stable solutions.

Conclusions: KEYS2 is a valid candidate to serve as a baseline technique for SBSE research.

Categories and Subject Descriptors

I.2.8 [Problem Solving, Control Methods, and Search]: Heuristic methods; D.2.8 [Requirements/Specifications]: Tools

Keywords

search-based software engineering, requirements optimization, meta-heuristic search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PROMISE2010, Sep 12-13, 2010. Timisoara, Romania
Copyright 2010 ACM ISBN 978-1-4503-0404-7...\$10.00.

1. INTRODUCTION

Despite decades of research, testing, trials, and deliberation from both industry and academia, many fundamental questions of the software engineering field remain unanswered. Given the scope of many of these issues (for example, the entire space of programs developed in JAVA) or the required balancing between competing factors (i.e. the amount of money spent versus the completion of goals), many of these questions are *unanswerable*. These are the areas where search-based software engineering (SBSE) has typically excelled.

Search-based software engineering is the practice of reformulating typical software engineering issues as search problems, and applying meta-heuristic methods to find solutions. For any problem that represents a set of competing - yet equally important - factors, there will be no single perfect solution. Instead, there are likely to be several *near-optimal* solutions. In this implied space of trade-offs, meta-heuristic search techniques are ideal for sifting through a number of potential solutions for those that contain an ideal balancing of factors.

Although the concept of applying search to software engineering problems has existed for decades, the term SBSE was coined and the field was formalized in 2001 [25]. Since then, the SBSE research community has expanded rapidly. SBSE research has been applied successfully to requirements engineering [2,39,43], project cost estimation [7,10,33], testing [4,17,40], software maintenance [23,37], transformation [3,13,24] and software evolution [6] (among others).

SBSE practitioners apply a variety of techniques, among them: genetic and evolutionary algorithms [5,27], hill-climbers [32,34,36], tabu search [20,21], particle swarm optimization [12,31], and ant colony optimization [11]. Yet, there exists no agreed-upon baseline technique. A number of these approaches are common - simulated annealing and genetic algorithms are heavily favored - and a "random search" is often used as a sanity measure, but there is no standard method to use as a basis for comparison when new methods are introduced.

In 1993, Robert Holte introduced the 1R classification algorithm [28]. In comparison to many of the prevalent techniques used at the time, such as the entropy-based C4.5 classifier, 1R was incredibly simple. The learning method treated each attribute of a data set as a continuous range (rather than as discrete intervals) and ranked them according to the error rate. Although Holte did not set out with the intention of defining a baseline for the data mining research field, there are several factors that perfectly positioned 1R

for such a title:

- *Simplicity*: 1R is easy to understand, using error rate as a judgment heuristic.
- *Competitive Results*: 1R produces results that are, on average, within 5% of C4.5's on the same data sets [28].
- *Fast Runtimes*: 1R is faster than more complex techniques.
- *Stable Results*: The results produced by 1R are consistent for the same data set over multiple trials.

A baseline technique for any field *must* have some level of simplicity. The algorithm must be easy to comprehend and easy to implement. A researcher's time and effort must go into the technique meant to surpass the baseline, not the baseline itself. However, it is not enough to be simple - "dumb" ideas will yield poor results. For a algorithm to stand as a valid baseline, it must meet all four of these factors; the heuristic must be easy-to-understand, fast, stable, and produce results that are competitive with state-of-the-art techniques.

The PROMISE conference series (and, ultimately, the entire academic world) seeks "repeatable, improvable, maybe even refutable, software engineering experiments."¹ It is not enough that some techniques, such as simulated annealing or random testing, reach near-ubiquity. The existence of a valid baseline gives a common basis of comparison, facilitating the repeatability and improvement of a wide variety of software engineering experiments. Just as the data mining field has benefited from the use of 1R as a baseline [42], the SBSE research field would benefit as well.

In this research, I propose the KEYS2 algorithm [19] as a candidate baseline technique for SBSE problems. KEYS2 is based on a straightforward theory - if the behavior of a model is controlled by a small number of *key* variables, then ideal solutions can be quickly found by isolating those variables and exploring the range of their values. In a case study centered around the optimization of early life-cycle requirements models, KEYS2 is demonstrated to fit all four of the factors required from a baseline technique and to have certain advantages over other potential baseline methods, such as random search:

- KEYS2 is based on a simple theory; that is, the exploitation of a small set of important variables.
- KEYS2 produces results that are of higher quality than a simulated annealer and are competitive with a genetic algorithm (as assessed by a Mann-Whitney rank-sum test with a 95% confidence interval).
- KEYS2 is faster than standard SBSE techniques, executing almost four times faster than the genetic algorithm on the most complex models.
- KEYS2 produces stable results, and has the ability to produce partial solutions.

This work builds on previous optimization research with the KEYS algorithm.

- The original KEYS algorithm is presented in [30].
- KEYS2 is presented and both it and KEYS2 are benchmarked against a variety of search techniques, including Simulated Annealing, in [19].

¹Quoted from http://promisedata.org/?page_id=2

- The contributions of this paper include a new genetic algorithm designed to optimize DDP models, new benchmarking experiments, and an explicit focus on a SBSE research context.

2. SEARCH-BASED SOFTWARE ENG.

Many of the problems inherent to the software engineering field deal in the *balancing* of competing factors. Consider these scenarios:

- Do you want to finish a project with money left in the budget, or is it more important to deliver a robust feature set?
- If the programmers are not meeting deadlines, you could replace them with a new team. Would the additional man-hours required for training and catch-up be worth the possible benefits?
- If a longer design cycle would result in a shorter programming period, will too many defects slip by unnoticed?

In many of these cases, there is no *perfect* solution. In fact, there could be dozens of "good" solutions. Instead of finding some theoretical catch-all answer, software engineering problems are typically concerned with near-optimal solutions. That is, the subset of solutions that fall within a certain tolerance threshold. While it may be impossible or impractical to attempt to find the single *best* solution, it is certainly possible to compare two candidate solutions. This comparison forms the basis of search-based software engineering.

Search-based software engineering is a research field concerned with the reformulation of standard software engineering problems as search problems, and the application of heuristic techniques to solve said problems. Because of this implied trade-off space, meta-heuristic search-based methods are ideal for producing a set of near-optimal candidate solutions.

According to Clark and Harman [22, 25], there are four key properties that must be met before attempting to apply a SBSE solution:

- *A Large Search Space*: If there are only a small number of options to optimize, there is no need for a meta-heuristic approach. This is rarely the case, as software engineering typically deals in incredibly large search spaces (for instance, the Eclipse project contains over two million lines of code and 7500 classes).
- *Low Computational Complexity*: If the first property is met, SBSE algorithms must sample a non-trivial population. Therefore, the computational complexity of any evaluation method or fitness function has a major impact on the execution time of the search algorithm.
- *Approximate Continuity*: Any search-based optimization must rely on an objective function for guidance. Some level of continuity will ensure that such guidance is accurate.
- *No Known Optimal Solutions*: If an optimal solution to a problem is already known, then there is no need to apply search techniques.

The first and last properties are absolutely necessary for any search-based software engineering solution to succeed. The second and third *should* be met, but if they are not, it

may still be possible to formulate the problem as a search-based one [3, 24].

2.1 Simulated Annealing

Simulated annealing (SA) [32, 36] is a sophisticated hill-climbing search algorithm. Hill-climbing searches survey the immediate neighborhood and "climb" to higher-scoring states, continuing until they reach a peak (that is, an optimal area in the search space). Hill-climbers are prone to becoming stuck in local maxima, returning a sub-optimal solution. To avoid this, simulated annealing utilizes a technique from its namesake. Annealing is a metallurgy technique where a material is heated, then cooled. The heat causes the atoms in the material to wander randomly through different energy states and the cooling process increases the chances of finding a state with a lower energy than the starting position. When assessing each neighbor, SA will jump to a sub-optimal neighbor at a probability determined by the current *temperature*. When that temperature is high, SA will jump around the search space wildly. As it cools, the algorithm will stabilize into what is, ideally, an optimal area of the search space.

Simulated annealing and hill-climbers are widely used in SBSE research for several reasons. They are simple to understand, easy to implement, and are typically very fast [34]. However, they have several known flaws. While the temperature function helps to prevent the search from becoming stuck in local maxima, it does not completely mitigate this problem. In fact, past research has shown that the additional randomness in this search often leads to an unacceptable level of variance in the results [18, 19]. Thus, one must take caution when choosing a cooling strategy - cool too quickly, and it is likely that the annealer will return a sub-optimal solution.

This research uses a version of simulated annealing tuned specifically for the DDP models (discussed in Section 3). During each round, SA "picks" a neighboring set of mitigations. To calculate the configuration of this neighboring solution, a function traverses the mitigation settings of the current state and randomly flips those settings (at a 5% chance). If the neighbor has a better score, SA will move to it and set it as the current state. If no score improvement is seen, the algorithm will decide whether or not to move based on the following probability function:

$$prob(moving) = e^{((score - neighborscore) * \frac{time}{temp})} \quad (1)$$

$$temp = \frac{(maxtime - time)}{maxtime} \quad (2)$$

If the value of the `prob` function is greater than some randomly generated number, SA will move to that state regardless of its score. The algorithm will continue to operate until the number of tries is exhausted (i.e. the temperature drops to zero) or a score meets the threshold requirement.

2.2 Genetic Algorithms

Inspired by early experiments with the computational simulation of evolution [5, 27], genetic algorithms (and the broader field of evolutionary algorithms) have become one of the most famous meta-heuristics used in the search-based software engineering literature. Influenced by Darwin's Theory of Evolution, genetic algorithms take a group of candidate solutions and mutate them over several generations - filter-

ing out bad "genes" and promoting good ones. Genetic algorithms rely on four major attributes: population, selection, mutation, and crossover.

During each generation (that is, each step of the algorithm), a population of solutions is considered. Each member of the population is made up of a set of binary switches. These switches are collectively known as the chromosomes, the genetic makeup, of a particular solution state. Typically, a potential solution must be carefully translated to this binary representation. However, as the models in this research are naturally represented as a set of binary variables, no special encoding of the model states is necessary. This has an additional benefit of reducing the required setup and execution time. The GA can make use of the same fitness function that the simulated annealer and KEYS2 use.

Because the initial population is unlikely to have the "best" solution, some form of diversity must be induced into the population. This diversity is established by forming children using the crossover and mutation operations. During each generation, several "good" solutions (as scored by a predetermined fitness function) are chosen by the selection mechanism to generate new children for the next generation. The values of the binary settings in these children are determined by the crossover mechanism, which combines chromosome settings from each parent and inserts them into the offspring with probability $P_{crossover}$. A mutation mechanism takes high-quality members of the current population and moves them over to the next generation while instilling small changes in their chromosomes. Mutation is necessary to prevent the algorithm from being trapped in local maxima. To avoid the loss of good solutions, a certain number of the best results will be copied to the next generation without any sort of modification. This process is repeated with a new population each round until a certain threshold (commonly related to the performance score, number of generations, or a set time period) has passed.

To summarize, a standard genetic algorithm follows this framework:

- Evaluate each member of the population.
- Create a new populations using these scores along with the crossover and mutation mechanisms.
- Discard the old population and repeat the process.
- Stop if $time > maxtime$ (in number of generations or some real-time threshold).

The genetic algorithm used in this research creates a population of size 100 each round, and selects all population members that score within 10% of the top-scoring member (as determined by the objective function defined in Section 3) to "reproduce." The chosen *parents* populate 40% of the next population (15% through crossover, 20% through mutation, and 5% are carried over unchanged). The remaining 60% of the population are randomly generated. These percentage values were chosen at the overall best values after an exhaustive search across five models, incrementing each factor by 5% each round. Each setting can be overridden by the user. This genetic algorithm stops after the number of past generations is equal to the number of variables in the model being optimized.

2.3 KEYS2

The core premise of KEYS2 is that standard SBSE techniques perform over-elaborate searches. Suppose that the

behavior of a large system is determined by a small number of *key* variables. If so, then stable near-optimal solutions can be quickly found by finding these *keys* and then exploring the range of their values.

Within a model, there are chains of *reasons* linking select inputs to the desired goals. Some of the links clash with others. However, some of those clashes are not dependent on other clashes. In the following chains of reasoning the clashes are $\{e, \neg e\}$, $\{g, \neg g\}$ & $\{j, \neg j\}$; the independent clashes are $\{e \neg e\}$, & $\{g \neg g\}$,

$$\begin{aligned} a &\longrightarrow b \longrightarrow c \longrightarrow d \longrightarrow e \\ \text{input}_1 &\longrightarrow f \longrightarrow g \longrightarrow h \longrightarrow i \longrightarrow j \longrightarrow \text{goal} \\ \text{input}_2 &\longrightarrow k \longrightarrow \neg g \longrightarrow l \longrightarrow m \longrightarrow \neg j \longrightarrow \text{goal} \\ &\quad \neg e \end{aligned}$$

In order to optimize decision making about this model, we must first decide about these independent clashes (these are the *keys*). Returning to the above reasoning chains, any of $\{a, b, \dots, q\}$ are subject to discussion. However, most of this model is completely irrelevant to the task of $\text{input}_i \vdash \text{goal}$. For example, the $\{e, \neg e\}$ clash is irrelevant to the decision making process as no reason uses e or $\neg e$. In the context of reaching *goal* from *input*, the only important discussions are the clashes $\{g, \neg g, j, \neg j\}$. Further, since $\{j, \neg j\}$ are dependent on $\{g, \neg g\}$, then the core decision must be about variable *g* with two disputed values: true and false.

Fixing the value of these keys reduces the number of reachable states within the model. This is called the *clumping* effect. Only a small fraction of the possible states are actually reachable. The effects of clumping can be quite dramatic. Without knowledge of these keys, the above example model has 2^{20} possible states. However, in the context of $\text{input}_i \vdash \text{goal}$, that massive collection of states clumps to the following two configurations: $\{\text{input}_1, f, g, h, i, j, \text{goal}\}$ or $\{\text{input}_2, k, \neg g, l, m, \neg j, \text{goal}\}$.

The KEYS algorithm finds these key variables using a greedy search and a Bayesian ranking heuristic (called BORE). If a model contains keys then, by definition, those variables must appear in all solutions to that model. If model outputs are scored by some objective function, then the key variables are those with ranges that occur with very different frequencies in the high and the low scoring model configurations. Therefore, we need not waste time searching for the keys - rather, we just need to keep frequency counts on how often ranges appear in *best* or *rest* outputs.

The greedy search explores a space of M mitigations over the course of N eras. Initially, the entire set of mitigations is set randomly. During each era, one more mitigation is set to $M_i = X_j$, $X_j \in \{\text{true}, \text{false}\}$. In the original version of KEYS [30], the greedy search fixed the value of one variable per era. KEYS2, fixes an increasing number of variables as the search progresses. That is, KEYS2 fixes one variable in the first era, two in the second era, three in the third era, and so on

In KEYS2, each era e generates a set $\langle \text{input}, \text{score} \rangle$ as follows:

1: *MaxTries* times repeat:

- *Selected*[1...($e - 1$)] are settings from previous eras.
- *Guessed* are randomly selected values for unfixed mitigations.
- *Input* = *selected* \cup *guessed*.

```

1. Procedure KEYS
2. while FIXED_MITIGATIONS != TOTAL_MITIGATIONS
3.   for I:=1 to 100
4.     SELECTED[1...(I-1)] = best decisions up to this step
5.     GUESSED = random settings to the remaining mitigations
6.     INPUT = SELECTED + GUESSED
7.     SCORES= SCORE(INPUT)
8.   end for
9.   for J:=1 to NUM_MITIGATIONS_TO_SET
10.    TOP_MITIGATION = BORE(SCORES)
11.    SELECTED[FIXED_MITIGATIONS++] = TOP_MITIGATION
12.  end for
13. end while
14. return SELECTED

```

Figure 1: Pseudocode for KEYS

- Call *model* to compute $score = ddp(\text{input})$;
- 2: The *MaxTries* scores are divided into $\beta\%$ “best” and remainder become “rest”.
- 3: The *input* mitigation values are then scored using BORE (described below).
- 4: The top ranked mitigations are fixed and stored in *selected*[*e*].

The search moves to era $e + 1$ and repeats steps 1,2,3,4. This process stops when every mitigation has a fixed value. The exact settings for *MaxTries* and β must be set via engineering judgment. Past research has shown that, for DDP model optimization, these should be set to *MaxTries* = 100 and β = 10 [30]. For full details, see Figure 1.

KEYS ranks mitigations using a support-based Bayesian ranking measure called BORE. BORE [8] (short for “best or rest”) divides numeric scores seen over K runs and stores the top 10% in *best* and the remaining 90% scores in the set *rest* (the *best* set is computed by studying the delta of each score to the best score seen in any era). It then computes the probability that a value is found in *best* using Bayes’ theorem. The theorem uses evidence E and a prior probability $P(H)$ for hypothesis $H \in \{\text{best}, \text{rest}\}$, to calculate a posteriori probability $P(H|E) = P(E|H)P(H) / P(E)$. When applying the theorem, *likelihoods* are computed from observed frequencies. These likelihoods (called “like” below for space consideration) are then normalized to calculate probabilities. This normalization cancels out $P(E)$ in Bayes’ theorem. For example, after $K = 10,000$ runs are divided into 1,000 *best* solutions and 9,000 *rest*, the value *mitigation31 = false* might appear 10 times in the *best* solutions, but only 5 times in the *rest*. Hence:

$$\begin{aligned} E &= (\text{mitigation31} = \text{false}) \\ P(\text{best}) &= 1000/10000 = 0.1 \\ P(\text{rest}) &= 9000/10000 = 0.9 \\ freq(E|\text{best}) &= 10/1000 = 0.01 \\ freq(E|\text{rest}) &= 5/9000 = 0.00056 \\ like(\text{best}|E) &= freq(E|\text{best}) \cdot P(\text{best}) = 0.001 \\ like(\text{rest}|E) &= freq(E|\text{rest}) \cdot P(\text{rest}) = 0.000504 \\ P(\text{best}|E) &= \frac{like(\text{best}|E)}{like(\text{best}|E) + like(\text{rest}|E)} = 0.66 \quad (3) \end{aligned}$$

Previously [8], it has been found that Bayes’ theorem is a poor ranking heuristic since it is easily distracted by low frequency evidence. For example, note how the probability of E belonging to the best class is moderately high even

though its support is very low; i.e. $P(best|E) = 0.66$ but $freq(E|best) = 0.01$.

To avoid the problem of unreliable low frequency evidence, Equation 3 is augmented with a support term. Support should *increase* as the frequency of a value *increases*, i.e. $like(best|E)$ is a valid support measure. Hence, step 3 of the greedy search ranks values via

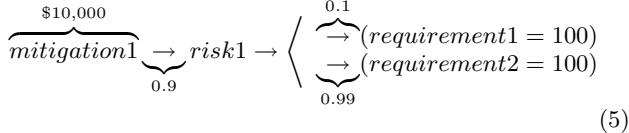
$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \quad (4)$$

3. CASE STUDY: THE DEFECT DETECTION AND PREVENTION MODEL

The Defect Detection and Prevention (DDP) requirements modeling tool [9, 14]. is used to interactively document the early life-cycle meetings conducted by "Team X" at NASA's Jet Propulsion Laboratory (JPL).

At Team X meetings, groups of up to 30 experts from various fields (propulsion, engineering, communication, navigation, science, etc) meet for short periods of time to produce a design document. This document may commit the current project to certain design choices. For example, the experts might choose solar power rather than nuclear power, or they might decide to use some particular style of guidance software. All subsequent work on the project is guided by the initial design decisions made in these mission concept documents.

The DDP model allows for the representation of the goals, risks, and risk-removing mitigations that belong to a specific project. During a Team X meeting, users of DDP explore the combinations of mitigations that will cost the least amount of money while still allowing for the completion of a large number of requirements. For example, here is a trivial DDP model where **mitigation1** costs \$10,000 to apply and each requirement is of equal value (100). Note that the mitigation can remove 90% of the risk. Also, unless mitigated, the risk will disable 10% to 99% of requirements one and two:



The other numbers show the impact of mitigations on risks, and the impact of risks on requirements. DDP propagates a series of influences over two matrices: one for *mitigations*risks* and another for *risks*requirements*.

DDP uses the following ontology:

- *Requirements* (free text) describe the objectives and constraints of the mission and its development process;
- *Weights* (numbers) of each requirement, reflecting their relative importance;
- *Risks* (free text) are events that damage the completion of requirements;
- *Mitigations*: (free text) are actions that reduce risks;
- *Costs*: (numbers) reflect the cost associated with activating a mitigation;
- *Mappings*: directed, weighted edges between requirements, mitigations, and risks that capture the quantitative relationships among them.
- *Part-of relations*: structure the collections of requirements, risks and mitigations;

1. Requirement goals:

- Spacecraft ground-based testing & flight problem monitoring
- Spacecraft experiments with on-board Intelligent Systems Health Management (ISHM)

2. Risks:

- Obstacles to spacecraft ground-based testing & flight problem monitoring
 - Customer has no, or insufficient, money available for my use
 - Difficulty of building the models and design tools
- ISHM Experiment is a failure (without necessarily causing flight failure)
- Usability, User/Recipient-system interfaces undefined
- V&V (certification path) untried and the scope is unknown
- Obstacles to spacecraft experiments with on-board ISHM
 - Bug tracking / fixes / configuration management issues, managing revisions and upgrades (multi-center tech. development issue)
 - Concern about my technology interfering with in-flight mission

3. Mitigations:

- Mission-specific actions
 - Spacecraft ground-based testing & flight problem monitoring
 - Become a team member on the operations team
 - Use Bugzilla and CVS
- Spacecraft experiments with on-board ISHM
 - Become a team member on the operations team
 - Utilize ISHM expert's experience and guidance with certification of his technology

Figure 2: Sample DDP requirements, risks, and mitigations.

Model	LOC	Objectives	Risks	Mitigations
model1.c	55	3	2	2
model2.c	272	1	30	31
model3.c	72	3	2	3
model4.c	1241	50	31	58
model5.c	1427	32	70	99

Figure 3: Details of Five DDP Models.

To improve runtimes, a compiler stores a flattened form of the DDP requirements tree in a function usable by any program written in the C language. In the compiled form, all computations are performed once and added as a constant to each reference of the requirement. The topology of the mitigation network is represented as terms in equations within the `model` function, which is called each time that a configuration of the entire model needs to be scored by a fitness function. As the models grow more complex, so do these equations. The largest real-world model used in this research, which contains 99 mitigations, generates 1427 lines of code. Figure 3 contains details on five publicly-available DDP models.

When the `model` function is called, two values are returned (the total cost of the selected mitigations and the number of reachable requirements attained). These two values are input into the following objective function in order to calculate a fitness score for the current configuration of the model. This objective function normalizes the cost and attainment values into a single score that represents the Euclidean distance to a *sweet spot* of maximum requirement attainment and minimum cost:

$$score = \sqrt{cost^2 + (attainment - 1)^2} \quad (6)$$

Here, \bar{x} is a normalized value $0 \leq \frac{x-min(x)}{max(x)-min(x)} \leq 1$. Hence, scores ranges $0 \leq score \leq \sqrt{2}$ and *lower* scores are better.

DDP is a valid choice for early life-cycle requirements optimization for three reasons:

- One potential drawback with ultra-lightweight models is that they are excessively lightweight and contain no useful information. DDP models are demonstrably useful for NASA, and clear project improvements (such as savings in power and mass) have been seen from DDP sessions at JPL. Cost savings of \$100,000 have been seen in multiple sessions, and in at least two sessions, they have exceeded \$1 million [14].
- Numerous real-world requirements models have been written in this format, and many projects are likely to use these models in the future [16]. The DDP tool can be used to document not just final decisions, but also to review the rationale that led to those decisions. Hence, it remains in use at JPL not only for its original purpose (group decision support), but also as a design rationale tool to document decisions.
- The DDP tool is representative of other requirements modeling tools in widespread use. DDP is a set of directed influences expressed in a rigid hierarchy and controlled by a set of equations. At this level of abstraction, DDP is just another form of QOC [41] or a variant of Mylopoulos' soft goal graphs [38].

Like any interesting search-based software engineering problem, DDP models represent a space of competing factors. In this case, the trade-off is between the budget of a project and the attainment of goals. Using the four properties of SBSE problems (as discussed in Section 2), it can be shown that the optimization of DDP models is a valid application area for SBSE techniques.

- Typical DDP models present a massive search space, encompassing hundreds to thousands of possible combinations of mitigation settings. One collection of model settings must comment on dozens of individual mitigations. In one known model, there are over 2^{99} ($mitigation\ values^{number\ of\ mitigations}$) possible combinations.
- With such a large number of combinations in the search space, any candidate solution must execute in a short time (to meet the second condition of a SBSE problems). As will be shown in Section 4.2, standard SBSE techniques execute in one second or less, implying a low computational complexity in the model assessment method. Furthermore, past research [19] has demonstrated that the proposed baseline method, KEYS2, operates in low-order polynomial time ($O(N^2)$).

- While DDP models do not represent a continuous search space (any model with *true/false* statements is, by definition, not continuous), this is not a problem. The objective function used to score a DDP configuration represents an approximately continuous trade-off between project costs and feature attainment. Therefore, it provides the necessary guidance needed to meet the third condition.
- Finally, DDP models have no known optimal solution. In fact, any solution is dependent on the specific features of the project being modeled. As such, a search method *must* be employed in order to calculate some set of *near-optimal solutions*.

By meeting all four of these conditions, the optimization of these early life-cycle DDP models is a valid avenue for search-base software engineering research.

4. EXPERIMENTS & RESULTS

While the "simplicity" of the KEYS2 algorithm is arguably subjective, the other three requirements of a baseline algorithm can be assessed qualitatively. In order to demonstrate that KEYS2 is a valid candidate for a baseline technique, it must be experimentally shown to be:

- Results that are competitive with those of state-of-the-art techniques.
- As fast as competing algorithms.
- A low level of variance in its results.

In the following experiments, three algorithms (KEYS2, a simulated annealer, and a genetic algorithm) will be used to optimize three of the five publicly-available DDP models². Note that:

- Models one and three are trivially small. They were used to debug source code, but have been excluded from the following experiments.
- Model 4 was previously discussed in detail [35]. Model 5 was optimized in [15].
- All five models were optimized by the original KEYS algorithm in [30]. However, that paper presented no comparison results.
- KEYS2 was benchmarked against a variety of algorithms, including simulated annealing, on models 2,4, and 5 in [19]. However, no results were reused. All results presented in this paper are from new trials.

4.1 Result Quality

By definition, the purpose of a baseline is to provide a *starting goal*. It should set some kind of bar that a newly-developed technique must pass before it can be considered for real-world use. However, while the goal of any experiment will be to "beat" the baseline score, that does not mean that a baseline should be some sort of "straw man." A baseline technique might not yield results that surpass the state-of-the-art, but it should at least be *competitive* with them.

In order to assess the overall result quality, each algorithm was executed 1000 times per model. These results are plotted in Figure 4. In each graph, the x-axis represents the "attainment" (the number of project objectives that were successfully completed in the final configuration of the model).

²These models are available from the PROMISE repository at <http://promisedata.org/?cat=133>

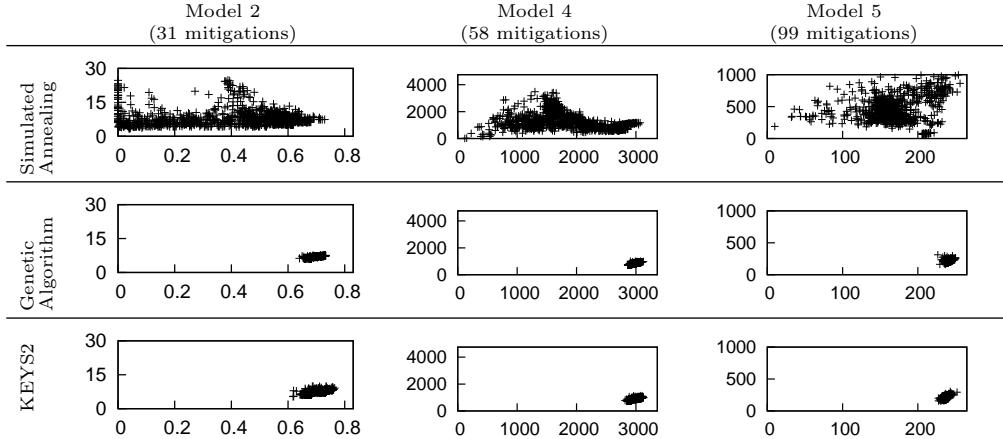


Figure 4: Results of each algorithm for each model (1000 trials per algorithm/model pairing). The y-axis shows cost (in thousands) and the x-axis shows attainment. Better solutions fall towards the bottom right of each plot (low costs and high attainment) and are clumped tightly together (low variance).

The y-axis is the cost, in thousands, of the mitigations employed by that model configuration. Better results are those that appear in the bottom-right corner. These are the configurations resulting in a low cost and high attainment of goals. It follows that the worst results are in the top-left, the area of high cost and low attainment.

A visual inspection gives a very clear idea of the result quality. The simulated annealing method can instantly be eliminated from the comparison. Its results, for every model, fall all over the graph. Very few of its conclusions result in an ideal balance of the cost and attainment. Even if some of its results are ideal, the massive level of variance eliminates it from the competition. Both KEYS2 and the genetic algorithm fare better, with the vast majority of their results concentrated in the ideal low cost, high attainment area.

To confirm this visual analysis, the distribution of values for each algorithm and each model were compared using a non-parametric rank-sum test (Mann-Whitney) with a 95% confidence interval. Separate tests were completed for cost and attainment for each of the three models. The wins, ties, and losses were aggregated and are listed in Figure 5.

Algorithm	Ties	Wins	Losses	Score (wins-losses)
Genetic Algorithm	0	7	5	2
KEYS2	0	7	5	2
Simulated Annealing	0	4	8	-4

Figure 5: Win-loss-tie results from a Mann-Whitney rank-sum test with a 95% confidence interval.

As expected, the wide range of results for simulated annealing hurt its overall result quality. The algorithm lost twice as often as it won in comparison tests. KEYS2 and the genetic algorithm yielded the exact same number of wins and losses. While they never tied, the fact that they traded wins and losses at such equal parity shows that KEYS2 is extremely competitive with the most commonly used SBSE techniques.

4.2 Runtimes

For both researchers and industrial practitioners, time is limited and expensive. By asking any of these individuals to run a baseline algorithm for comparison purposes, we are implicitly asking them to spend more time considering a problem. Therefore, it is crucial that any baseline technique be fast, even real-time if possible.

In order to compare execution times, each algorithm was executed for each model 100 times. The total time for these 100 trials was recorded using the Unix `time` command, and the "real time" value was divided by 100 in order to calculate the average execution time.

All three algorithms were executed under the same operating conditions on the same machine - a Dell XPS workstation with a 2.40 GHz Intel Core2 Duo CPU and 2 GB of memory running the Ubuntu 8.04 operating system.

Algorithm	Model 2	Model 4	Model 5
simulated annealing	0.40967	0.94350	0.64050
genetic algorithm	0.00976	0.04625	0.09948
KEYS2	0.00430	0.01407	0.02695

Figure 6: Average runtimes for each algorithm on each model, in seconds, averaged over 100 runs.

The results of this experiment can be seen in Figure 6. The simulated annealer clearly experiences some difficulty in optimizing these models. It takes orders of magnitude more time to complete a single run than other algorithms.

The remaining two algorithms are incredibly fast, both returning results in under a tenth of a second on average. A closer look shows that KEYS2 is still much faster than the genetic algorithm, completing each trial between 2.27 to 3.69 times faster than its competitor.

4.3 Stability

The results shown in Section 4.1 give some idea of the variance in the performance of each of the compared techniques. From a visual inspection of Figure 4, it is clear that the annealer's final values are unpredictable. Very few of

its results fall in the ideal bottom-right corner (where the project leads pay a little to achieve a lot). The final score values of both the genetic algorithm and KEYS2 fall in a much smaller cluster. To gain a clearer view of the result variance of each algorithm, we will examine the range of values output by each algorithm for the largest model (model 5 from Section 3).

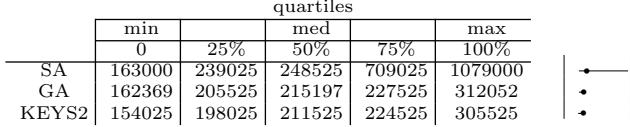


Figure 7: Quartile charts of the final “cost” results, taken from 1000 executions of each algorithm on model 5. Lines represent the full range of values, with the dot representing the median.

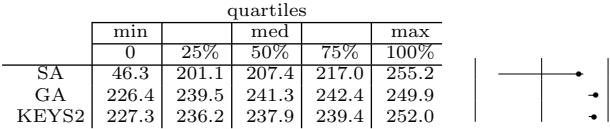


Figure 8: Quartile charts of the final “attainment” results, taken from 1000 executions of each algorithm on model 5. Lines represent the full range of values, with the dot representing the median.

Figures 7 and 8 show quartile breakdowns of the value distributions for the cost and attainment results on model 5. Simulated annealing reaches median values that are weaker than, but not too far from, the other two algorithms. However, the full range of its results is much wider than that of KEYS2 or the genetic algorithm. Simulated annealing demonstrates an unacceptable level of variance. KEYS2 and the GA yield similar results. The GA has a slightly tighter range of results with a spread (75th quartile - 25th) of 22000 to 26500 for cost and 2.9 to 3.2 on attainment.

From this, it can be concluded that KEYS2 demonstrates the consistency required of a baseline method. However, its very design provides one additional degree of stability. Recall that each round, KEYS2 seeks out the most important variable and fixes its value. Thus, by providing a linear ranking of model variables, KEYS2 could be stopped at any point during its execution. Unlike the other two algorithms, KEYS2 can provide partial results. Because it generates 100 random configurations at each stage of execution, the variance of those partial decisions can be measured.

Figure 9 demonstrates the internal stability during one execution of KEYS2 on model 5. At any stage, the spread (i.e. the variance) of KEYS internal decisions can be measured. It can clearly be seen that KEYS2 rapidly plateaus towards stable, high-scoring results for both the cost and attainment of goals. The ability to generate partial *or* complete solutions, both with a low degree of variance, lend credence to the use of KEYS2 as an experimental baseline.

5. DISCUSSION & CONCLUSIONS

Despite the extensive body of research that has been invested in the search-based software engineering field over

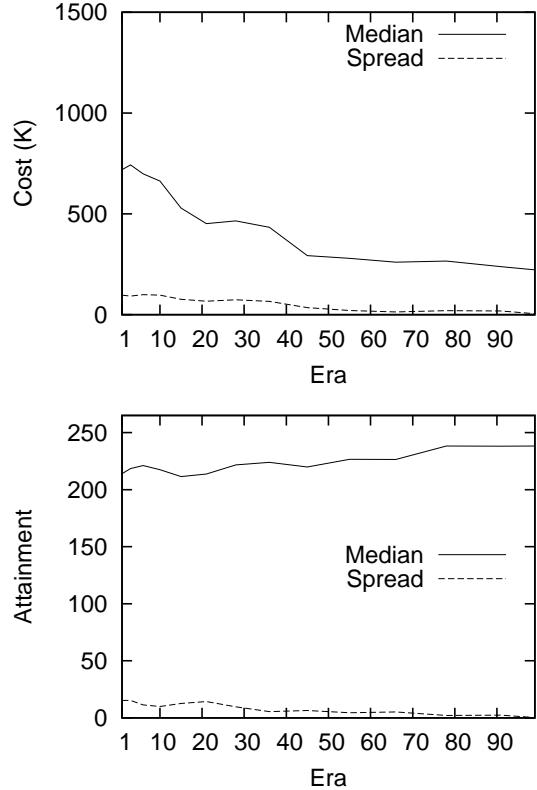


Figure 9: Median and spread of internal decisions made by KEYS2 during one trial. X-axis shows the number of rounds passed. “Median” shows the 50th percentile of the measured value seen in the 100 random configurations made during each round. “Spread” shows the difference between the 75th and 50th percentile.

the past few years, there is still no agreed-upon basis for comparison between the disparate research tracts. With all of the different improvements to and implementations of the “standard” techniques, it has become difficult to achieve the PROMISE community’s goal of “repeatable, improvable, maybe even refutable, software engineering experiments.”

For the further expansion of the body of knowledge, it is necessary for the SBSE field to adopt some sort of baseline technique. Just as data mining research has benefited from Holte’s 1R algorithm [28], the SBSE community would benefit from a common baseline. Such a technique would allow comparisons between seemingly incomparable algorithms, would more easily allow for the repetition and improvement of existing experiments, and would give a performance goal to be beaten by newly-developed methods.

However, not just any algorithm can be chosen as a baseline. A baseline technique must meet several criteria:

- *Simplicity:* The method must be easily understood and easily implemented. If not, researchers will waste crucial time and brainpower that would be better spent on their own ideas.
- *Competitive Results:* Although a baseline is meant to be “beaten,” it should not be a straw man. It must

yield results within the same neighborhood as the state-of-the-art.

- *Fast Runtimes:* By asking a researcher to execute a baseline algorithm, you are implicitly asking them to devote more time to their experiments. Thus, an ideal baseline technique should execute in an almost unnoticeable amount of time.
- *Stable Results:* A baseline technique must be reliable and trustworthy. It must return results that fall within a small range of fitness values.

It is not enough for a technique to meet one or two of these criteria. A fast algorithm is pointless if its results are seemingly random. Likewise, a simple method is not useful if its results are poor. A baseline algorithm should be fast, reliable, competitive, and be straight-forward to implement.

Out of the commonly-used approaches in the SBSE field, random search is likely the closest to an agreed-upon baseline. Random search is used by many researchers as a "sanity check," a de facto bar to beat with their algorithm of choice [26]. However, random search is a weak approach. Its very nature implies that it will eventually violate either the second or third of the previously-specified properties of a baseline. Although it *may* find an ideal solution in a short amount of time, there is no way to ensure that it *always* will. Because it picks inputs at random, there will be a massive amount of variance in its final results (for a similar effect, see the simulated annealing results in Sections 4.1 and 4.3). There is some empirical evidence that, given enough time, random testing will deliver predictable results [29]. However, such a time threshold will violate the second guideline, that any baseline execute quickly. Many researchers seem to have little confidence in random search, treating it as a straw man. As Harman notes, "It is something of a 'sanity check'; in any optimization problem worthy of study, the chosen technique should be able to convincingly outperform random search. [26]" The use of random search is an ongoing debate in the SBSE community [1]. I would suggest, despite the popularity of random search, a stronger baseline algorithm should be chosen.

The KEYS2 algorithm [19, 30] meets all of these criteria. It is based on a straightforward theory, that a small number of important variables control the vast majority of the search space. A search that rapidly isolates these variables and fixes their values will quickly plateau towards stable, optimal solutions. In a case study centered around the DDP early life-cycle requirements models [9, 14], KEYS2 was demonstrated to fulfill the other three requirements.

- In Section 4.1, it is shown that KEYS2 and a genetic algorithm are statistically identical in terms of quality. They yielded the same number of wins and losses in a series of rank-sum tests.
- Section 4.2 shows that KEYS2 executes three or more times faster than other SBSE techniques.
- Finally, Section 4.3 demonstrates that KEYS2 yields stable results. KEYS2 can also output partial results, and the variance and spread of these can be examined as well.

Thus, KEYS2 is a candidate to serve as a baseline technique for the SBSE field.

I am not presumptuous enough to insist that KEYS2 be the absolute baseline or even that it is the best candidate. However, it is crucially important to debate this topic. Even

amongst "standard" techniques like simulated annealing, there are thousands of slightly different implementations and tweaks. Direct comparisons are almost impossible. For the goal of open research to become possible, a simple (yet competitive) technique should be adopted and made available to the research community. Such a baseline is necessary for the growth of this research field, and the PROMISE community is in an ideal position to promote its use.

6. REFERENCES

- [1] A. Arcuri, M. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 219–230, New York, NY, USA, 2010. ACM.
- [2] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14), December 2001.
- [3] A. Baresel, D. Wendell Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, 2004.
- [4] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1329–1336. Morgan Kaufmann Publishers, 2002.
- [5] N. Aall Barricelli. Esempi numerici di processi di evoluzione. *Mehodos*, pages 45–68, 1954.
- [6] T. Van Belle and D. Ackley. Code factoring and the evolution of evolvability. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1383–1390. Morgan Kaufmann Publishers, 2002.
- [7] C. Burgess and M. Lefley. Can genetic programming improve software effort estimation? a comparative evaluation. *Information and Software Technology*, 43(14):863–873, 2001.
- [8] R. Clark. Faster treatment learning, Computer Science, Portland State University. Master's thesis, 2005.
- [9] S.L. Cornford, M.S. Feather, and K.A. Hicks. DDP a tool for life-cycle risk management. In *IEEE Aerospace Conference, Big Sky, Montana*, pages 441–451, March 2001.
- [10] J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43(1):61–72, 2001.
- [11] M. Dorigo and L. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computing*, 1:53–66, 1997.
- [12] R.C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *6th International Symposium on Micromachine Human Science*, pages 39–43, 1995.
- [13] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In *In 4th International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 65–71,

- Los Alamitos, CA, USA, 2004. IEEE Computer Society Press.
- [14] M. Feather, S. Cornford, K. Hicks, J. Kiper, and T. Menzies. Application of a broad-spectrum quantitative requirements model to early-lifecycle decision making. *IEEE Software*, 2008. Available from <http://menzies.us/pdf/08ddp.pdf>.
 - [15] M.S. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002. Available from <http://menzies.us/pdf/02re02.pdf>.
 - [16] M.S. Feather, S. Uckun, and K.A. Hicks. Technology maturation of integrated system health management. In *Space Technology and Applications International Forum (STAIF-2008) Albuquerque, USA*, February 2008.
 - [17] M. Claudia Figueiredo, P. Emer, and S. Regina Vergilio. GPTesT: A testing tool based on genetic programming. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1343–1350. Morgan Kaufmann Publishers, 2002.
 - [18] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Accepted for Automated Software Engg.*, 17(4), 2010.
 - [19] G. Gay, T. Menzies, O. Jalali, G. Mundy, B. Gilkerson, M. Feather, and J. Kiper. Finding robust solutions in requirements models. *Automated Software Engg.*, 17(1):87–116, 2010.
 - [20] F. Glover. Tabu search - part 1. *ORSA Journal on Computing*, 1:190–206, 1989.
 - [21] F. Glover. Tabu search - part 2. *ORSA Journal on Computing*, 2:4–32, 1990.
 - [22] M Harman and J. Clark. Metrics are fitness functions too. In *10th International Software Metrics Symposium (METRICS 2004)*, 2004), pages = 58–69, location = Chicago, IL, USA, publisher = IEEE Computer Society Press, address = Los Alamitos, CA, USA,.
 - [23] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358. Morgan Kaufmann, July 2002.
 - [24] M. Harman, L. Hu, R. Mark Hierons, J. Wegener, H. Stamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.
 - [25] M. Harman and B.F. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
 - [26] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, 2010.
 - [27] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.
 - [28] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.
 - [29] Ciupa I., A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 2009.
 - [30] O. Jalali, T. Menzies, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings of the PROMISE 2008 Workshop (ICSE)*, 2008. Available from <http://menzies.us/pdf/08keys.pdf>.
 - [31] J. Kennedy and R.C. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.
 - [32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, Number 4598, 13 May 1983, 220, 4598:671–680, 1983.
 - [33] C. Kirsopp, M. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1367–1374. Morgan Kaufmann Publishers, 2002.
 - [34] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.
 - [35] T. Menzies, J. Kiper, and M. Feather. Improved software engineering decision support through automatic argument reduction tools. In *SEDECS'2003: the 2nd International Workshop on Software Engineering Decision Support (part of SEKE2003)*, June 2003. Available from <http://menzies.us/pdf/03star1.pdf>.
 - [36] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, 1953.
 - [37] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1375–1382. Morgan Kaufmann Publishers, 2002.
 - [38] J. Mylopoulos, L. Cheng, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, January 1999.
 - [39] A. Ngo-The and G. Ruhe. Optimized resource allocation for software release planning. *Software Engineering, IEEE Transactions on*, 35(1):109–123, Jan.-Feb. 2009.
 - [40] S.L. Rhys, S. Pouling, and J.A. Clark. Using automated search to generate test data for matlab. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1697–1704, New York, NY, USA, 2009. ACM.
 - [41] S. Buckingham Shum and N. Hammond. Argumentation-based design rationale: What use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, 1994.
 - [42] I. H. Witten and E. Frank. *Data mining. 2nd edition*. Morgan Kaufmann, Los Altos, US, 2005.

- [43] Y. Zhang, M. Harman, and S.A. Mansouri. The multi-objective next release problem. In *In ACM Genetic and Evolutionary Computation Conference (GECCO 2007*, page 11, 2007.

APPENDIX

Acquiring the Algorithms

In the spirit of the PROMISE conference series, all of the algorithm implementations and data used in this research are available for public use.

All five DDP models are available from the PROMISE repository at <http://www.promisedata.org/?cat=133>.

KEYS2 can be downloaded from <http://www.unbox.org/wisp/var/greg/keys/2.6>. The genetic algorithm can be found at <http://www.unbox.org/wisp/var/greg/genefreeze> and the simulated annealing is located at <http://www.unbox.org/wisp/tags/ddpExperiment>.

When to Use Data from Other Projects for Effort Estimation

Ekrem Kocaguneli,
Gregory Gay
Tim Menzies
LCSEE, WVU, USA
ekocagun@mix.wvu.edu,
greg@greggay.com,
tim@menzies.us

Ye Yang
Institute of Software
Chinese Academy of Sciences
Beijing, People's Republic of
China
ye@itechs.iscas.ac.cn

Jacky W. Keung
School of Computer Science
and Engineering
University of New South Wales
Sydney, Australia
jacky.keung@nicta.com.au

ABSTRACT

Collecting the data required for quality prediction *within* a development team is time-consuming and expensive. An alternative to make predictions using data that *crosses* from other projects or even other companies. We show that with/without *relevancy filtering*, imported data performs the same/worse (respectively) than using local data. Therefore, we recommend the use of relevancy filtering whenever generating estimates using data from another project.

Categories and Subject Descriptors: D.2.9 [Software Engineering]: Effort Estimation

General Terms: Economics, Measurement

Keywords: Effort estimation, data mining, cross, within

1. INTRODUCTION

When data is scarce *within* one project, it is tempting to use data imported from other projects. Such *cross-project* data exist; for example the PROMISE repository [2] offers a dozen effort estimation data sets for public access.

A recent survey paper has evaluated *within* or *cross* data for effort estimation [5]. They concluded that they could not make a conclusion; that the current findings are contradictory about the relative merits of *within* or *cross* data.

In other work [10], we have shown that it is acceptable to use *cross* data sources for defect prediction, providing that data has been pre-processed by some sort of *relevancy filtering*. Given a large training set, such relevancy filters select a small subset relevant to the current test case. Such filtering removes training instances that create noise in the estimation process, leaving a body of data that, in theory, follows the principle of locality.

The success of relevancy filtering for defect prediction prompts us to apply it to effort estimation. To the best of our knowledge, this is the first exploration in the effort estimation community of the effects of relevancy filtering when applied to *cross* and *within* project data. We show that *cross* data can usually attain estimation accuracies just as high as those of *within* data, provided that a relevancy filter is applied to the data, prior to making estimates.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

Copyright 2010 ACM 978-1-4503-0116-9/10/09 ...\$10.00.

2. RELEVANCY FILTERING

Our relevancy filter extends standard analogy-based estimation methods (which we call ABE0). ABE0 generates estimates for a test project by gathering evidence from the effort values of similar projects in some training set. By analyzing the previous research of experts like Shepperd et. al. [9], Mendes et. al. [8] and Li et. al. [7] on the field of analogy-based estimation, we can come up with a baseline technique:

- Build a training data from rows of past projects;
- The columns of this set are composed of *independent* variables (the features that define projects) and a *dependent* variable (the recorded effort value).
- Decide on how many similar projects (*analogies*) to use when examining a new test instance, i.e. *k*-values.
- For each test instance, select *k* analogies from training.
 - While selecting analogies, use a similarity measure (such as the Euclidean distance of features).
 - Before calculating similarity, apply a scaling measure on independent features to equalize their influence on this similarity measure.
- Use the effort values of the *k* nearest analogies to calculate an effort estimate.

We can refer to this baseline framework as ABE0. For the similarity measure, we use Euclidean distance:

$$Distance = \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \quad (1)$$

ABE0 returns the median of the efforts in the *k* nearest analogies.

Our relevancy filter is a small variant of ABE0. It is a two-pass system. Pass 1 removes the training instances implicated in poor decisions; pass 2 selects those instances nearest the test instance.

In pass 1, the training projects are used to generate a binary tree. The leaves of this binary tree are formed by the individual training projects, which are then greedily clustered in tuples to form the parent levels. This binary tree, which we will call BT1, is then traversed upwards from the root to height level one (one higher level than the leaves). The variance of the effort values in each subtree (the performance variance) is then recorded and normalized to a 0-1 interval. Pass one prunes all sub-trees with a variance greater than 10% of the maximum variance seen in any tree.

The leaves of the remaining sub-trees are the *survivors* of pass one. These move to pass 2 where the survivors are used to build a second binary tree (called BT2). BT2 is generated and traversed by test instances in the same fashion as BT1. This time, while traversing the tree, instead of storing the variances of sub-trees,

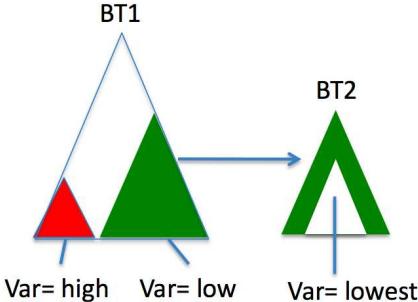


Figure 1: Two pass relevancy filtering. Each tree BT1 and BT2 are binary cluster trees. The red sub-tree is pruned in pass one due to high variance. The remaining subtrees (shown in green) form the right-hand tree. In pass two, test instances start at the root of this tree and traverse to the nearest child (and so on, recursively). While the sub-tree variance continues to decrease, the traversal continues. Estimates are generated from the median of the instances of the right-hand-side sub-tree with lowest variance.

we use the variance as a decision criterion. If the variance of the current tree is larger than its sub-trees, then continue to move down the subtree; otherwise, stop moving and select the instances in the current tree as the relevant instances and adapt them for estimation. Relevancy filtering is visualized in Figure 1.

This filter is similar to the NN-filter used by Turhan et.al. [10], except that there is no need to pre-specify the number of analogies k to be used for estimation. Each test instance selects its own relevant analogies by traversing to different sub-trees of BT2.

For a detailed discussion on the rationale behind this filter, see [6]. All we need to say here is that this filter is known to generate low errors for ABE0-style effort estimation [6]. Hence, it is a suitable tool for the rest of this study.

3. METHODOLOGY

In our research, we have used subsets of three commonly-used datasets in software effort estimation research: Nasa93, the original Cocomo81 [1], and Desharnais [4].

We will denote the subsets of Nasa93 as Nasa93c1, Nasa93c2 and Nasa93c5. Nasa93c1, Nasa93c2 and Nasa93c5 contain projects from different NASA development centers around the United States (denoted as development centers 1, 2 and 5 in the complete dataset). In a similar fashion, subsets of Cocomo81 will be denoted as Coc81o, Coc81e and Coc81s (for organic, embedded, and semidetached). Finally, the Desharnais dataset is split into three different subsets: DesL1, DesL2 and DesL3 (languages 1, 2 and 3 respectively). Since each of these subsets have certain common criteria (the development center, development mode, or development language), each subset will be treated as a separate *within* dataset. All of the datasets used in this research are available in PROMISE data repository [2].

For each of the three main datasets (Nasa93, Cocomo81 and Desharnais) in our research, we have conducted *within* and *cross* experiments. Each subset became a *within* dataset that contains projects sharing the particular characteristics of a single development firm.

To understand the *within* and *cross* data formation, assume that a dataset X with its three subsets X_1 , X_2 and X_3 is under consideration. For *within* experiments, relevancy filtering described is

applied on each one of X_1 , X_2 and X_3 separately and the median of the filtered project instances in the training set is stored as the effort estimate for the test instance. For the separation of training and testing sets, the leave-one-out method is used. Leave-one-out selects one instance out of a dataset of n instances as the test instance and uses the remaining $n - 1$ instances as the training set.

For the *cross* experiments, one of X_1 , X_2 or X_3 is chosen as the test set and the combination of the remaining two forms the *cross* dataset for training. This time, the relevancy filtering is applied on the *cross* dataset, and the estimations for projects in the test set are stored.

Each of the *within* and *cross* experiments are repeated twenty times in order to remove any bias that would otherwise be brought by a particular test and training set combination.

In order to compare the performance of *within* and *cross* datasets, we have used two measures: the magnitude of relative error (MRE) and win-tie-loss values generated by a statistical rank-sum test.

$$MRE = \frac{|actual_i - predicted_i|}{actual_i} \quad (2)$$

Using a Mann-Whitney test (95%), we checked how often one treatment won/lost/tied with the others. Here, a “tie” means that they are not statistically significant different. If statistically different, then the method with a lower median MRE score gets one more “win” and the other method gets one more “loss”.

4. RESULTS

In our experiment, we analyzed 3 datasets * 3 subsets = 9 treatments. We evaluated *cross* and *within* performances of each particular dataset, subject to statistical tests, with and without relevancy filtering.

4.1 Without Relevancy Filtering

In this first experiment, we have 9 treatments and for each treatment we observe the estimation performances when *within* and *cross* datasets are used. For this purpose we used a linear regression model. Two-pass filtering was not applied.

In *cross* experiments, for each data set, we selected one of the 3 subsets as the test set and the remaining two as the train set. We then built a linear regression model on the *cross* data and applied this model on the test set. For the *within* dataset we also used a linear regression model. The test case selection for *within* experiment is performed in accordance with leave-one-out method, which picks up one of the instances in the dataset as the test set and uses the remaining instances as the train set. The linear regression model that is built on the train set is then tested on the single test instance. After applying linear regression model on *within* and *cross* datasets, we calculated the win-tie-loss values for each treatment.

As shown in Figure 2, we see that in a minority of cases ($\frac{4}{9}$, see Nasa93c5, Coc81e, Coc81s and DesL2), *cross* and *within* data perform just as well as each other. In the majority case ($\frac{5}{9}$, see Nasa93c1, Nasa93c2, Coc81o, DesL1, DesL3), *within* performed better than *cross* data. That is, in the absence of relevancy filtering, the *within* datasets yield significantly lower MRE values in majority of cases.

4.2 With Relevancy Filtering

This section shows that for each data set, the application of relevancy filtering reverses the conclusion of the previous section; i.e. the *cross* data becomes useful for estimating the local project.

Figure 3, shows the the win-tie-loss values for the subsets of Nasa93. The greedy clustering algorithm of the two pass relevancy filtering uses some non-determinism (when breaking ties be-

Data set	Train Set	Test Set	Method	Win	Tie	Loss
Nasa93	Nasa93c1	Leave-one-out test instance	Within	1		
	Nasa93c2 and Nasa93c5	Nasa93c1	Cross		1	
	Nasa93c2	Leave-one-out test instance	Within	1		
	Nasa93c1 and Nasa93c5	Nasa93c2	Cross		1	
	Nasa93c5	Leave-one-out test instance	Within	1		
	Nasa93c1 and Nasa93c2	Nasa93c5	Cross		1	
Cocomo81	Coc81o	Leave-one-out test instance	Within	1		
	Coc81e and Coc81s	Coc81o	Cross		1	
	Coc81e	Leave-one-out test instance	Within		1	
	Coc81o and Coc81s	Coc81e	Cross		1	
	Coc81s	Leave-one-out test instance	Within		1	
	Coc81o and Coc81e	Coc81s	Cross		1	
Desharnais	DesL1	Leave-one-out test instance	Within	1		
	DesL2 and DesL3	DesL1	Cross		1	
	DesL2	Leave-one-out test instance	Within		1	
	DesL1 and DesL3	DesL2	Cross		1	
	DesL3	Leave-one-out test instance	Within	1		
	DesL1 and DesL2	DesL3	Cross		1	

Figure 2: MRE win-tie-loss results without relevancy filtering. Every odd and even line is a pair of experiments. In each pair, there is a *within* and a *cross* experiment. In *cross* experiment, a linear regression model is built on *cross* data and tested on the *within* data. In *within* experiment, the test instance is selected with leave-one-out, and a linear regression model is built on the remaining instances and tested on the selected test instance. A “1” denotes which item in the pair won, lost or tied.

tween instances of similar distances), so we repeat these experiments twenty times.

This results shows us that, in all three treatments, the *tie* values are quite high. This indicates that, for at least 75% of the tests, there is no statistical difference between filtered *cross* and *within* results. In short, for Nasa93, the performance of *cross* data (filtered for relevancy) is indistinguishable from the performance of *within* data.

Dataset	Method	Win	Tie	Loss
Nasa93c1	within	3	15	2
Nasa93c2 and Nasa93c5	cross	2	15	3
Nasa93c2	within	3	17	0
Nasa93c1 and Nasa93c5	cross	0	17	3
Nasa93c5	within	1	19	0
Nasa93c1 and Nasa93c2	cross	0	19	1

Figure 3: MRE win-tie-loss values for Nasa93 from 20 randomized assessments. In all treatments *tie* values are quite high. For Nasa93, the performance of *cross* data is mostly same as *within* data.

Figure 4 shows the win-tie-loss values for the subsets of Cocomo81. In two out of the three treatments the *tie* values are 19, which tells us that for these treatments, *within* and *cross* performance are almost identical. However, the first treatment shows a preference for *within* data on thirteen of the twenty tests.

The win-tie-loss values for subsets of Desharnais are given in Figure 5. The derived results for the Desharnais subsets are similar to those of Cocomo81 treatments: Two out of the three treatments show identical *tie* values of 19, which again suggests that the performance of filtered *cross* datasets is statistically identical to *within* datasets. However, in one of the treatments, *within* outperforms *cross* on sixteen of the twenty trials.

In summary, with relevancy filtering, in the majority case ($\frac{7}{9}$ treatments) the *cross* data performs as well as the *within* data for ef-

Dataset	Method	Win	Tie	Loss
Coc81o	within	13	7	0
Coc81e and Coc81s	cross	0	7	13
Coc81e	within	1	19	0
Coc81o and Coc81s	cross	0	19	1
Coc81s	within	0	20	0
Coc81o and Coc81e	cross	0	20	0

Figure 4: MRE win-tie-loss values for Cocomo81 from 20 randomized assessments. In 2 treatments *cross* data is the same as the *within* data. However, in the case of Coc81o, *within* outperforms *cross* data.

Dataset	Method	Win	Tie	Loss
DesL1	within	1	19	0
DesL2 and DesL3	cross	0	19	1
DesL2	within	1	19	0
DesL1 and DesL3	cross	0	19	1
DesL3	within	16	4	0
DesL1 and DesL2	cross	0	4	16

Figure 5: MRE win-tie-loss values for Desharnais from 20 randomized assessments. In the case of DesL3 the *within* data is much better than the *cross* data. For other treatments, *within* and *cross* data are statistically the same.

fort estimation. There are only two treatments, DesL3 and Coc81o, where *within* performance was significantly better than *cross* performance. A possible explanation for those two scenarios may be hidden in the dataset size or in the quality of the *within* datasets, but the currently-available information makes it difficult to suggest any conclusive reason for the situation.

Cross Dataset	Test Set	Instances selected in BT2		
Nasa93c2 and Nasa93c5 Nasa93c1 and Nasa93c5 Nasa93c1 and Nasa93c2	Nasa93c1 Nasa93c2 Nasa93c5	From Nasa93c1	From Nasa93c2	From Nasa93c5
		0	2.4	1.2
		1.4	0	1.8
Coc81e and Coc81s Coc81o and Coc81s Coc81o and Coc81e	Coc81o Coc81e Coc81s	From Coc81o	From Coc81e	From Coc81s
		0	2.0	1.3
		3.6	0	1.4
DesL2 and DesL3 DesL1 and DesL3 DesL1 and DesL2	DesL1 DesL2 DesL3	From DesL1	From DesL2	From DesL3
		0	2.3	0.7
		2.1	0	0.1
		3.2	1.6	0

Figure 6: Mean number of instances used for estimation after filtering in 20 runs. Cross datasets are combinations of two *within* datasets tested on another *within* dataset.

5. DISCUSSION

Figure 6 shows the mean number of instances used for analogy-based estimation by our two-pass relevancy filtering algorithm. Surprisingly, the number of selected analogies is very small: mean value around 3. Further, while exceptions exist, the selected analogies come from multiple other projects. For example, for the Nasa93 dataset, the data relevant to center c1 came $\frac{2}{3}$ -rds and $\frac{1}{3}$ -rd from centers c2 and c5 (respectively).

This suggests that we should revisit what we mean by *within* and *cross*:

- Effort estimation functions on a theory of *locality*; i.e. new projects follow similar practices to historical projects and should require a similar amount of effort. As Chen et. al. [3] have shown, inconsistencies in data collection across multiple companies create locality-specific biases in *cross* data sets. Such biases result in an unacceptable amount of variance in the effort calculations.
- Figure 6 is saying that *similar* projects may not come from the same geographical location. Our relevance filters hunted out handfuls of *similar* projects from other subsets. Perhaps software differs on many dimensions (such as where it was written), and the most important dimensions for finding *similar* projects may not be mere geography.

Given the complex multi-dimensional nature of the software creation process, the geographical dimension may be less important than other factors. The most similar software to what you are writing now may not be in the next office. Rather, it may be in an office on the other side of the world.

Going forward, we would like to learn exactly why an instance is deemed “relevant” or “irrelevant” by our filter. In other words, we would like to know exactly which features are most influential when assigning relevancy. The ability to identify these exact dimensions would make the selection of appropriate projects easier for any institution that uses *cross* data. It would also lead to (a) more accurate filtering techniques; and (b) a better understanding of the structure of software projects including where to find data most relevant to some current project.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China under Grant Nos. 60873072 and 90718042.

6. REFERENCES

- [1] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [2] G. Boetticher, T. Menzies, and T. Ostrand. PROMISE repository of empirical software engineering data, 2007.
- [3] J. Chen, Y. Yang, V. Nguyen, and Q. Li. Reducing the local bias in calibrating the general cocomo. *International Forum on COCOMO and Systems-Software Cost Modeling*, 2009.
- [4] J. Desharnais. Analyse statistique de la productivité des projets informatique à partie de la technique des point des fonction. Master’s thesis, Univ. of Montreal, 1989.
- [5] B. A. Kitchenham, E. Mendes, and G. H. Travassos. Cross versus within-company cost estimation studies: A systematic review. *IEEE Trans. Softw. Eng.*, 33(5):316–329, 2007.
- [6] E. Kocaguneli. Better methods for configuring case-based reasoning systems. Master’s thesis, 2010.
- [7] Y. Li, M. Xie, and T. Goh. A study of project selection and feature weighting for analogy based software cost estimation. *Journal of Systems and Software*, 82:241–252, 2009.
- [8] E. Mendes, I. D. Watson, C. Triggs, N. Mosley, and S. Counsell. A comparative study of cost estimation models for web hypermedia applications. *Empirical Software Engineering*, 8(2):163–196, 2003.
- [9] M. Shepperd, C. Schofield, and B. Kitchenham. Effort estimation using analogy. In *International Conference on Software Engineering*, pages 170–178, 1996.
- [10] B. Turhan, T. Menzies, A. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14:540 – 578, 2009.

Finding Robust Solutions in Requirements Models

Gregory Gay¹, Tim Menzies¹, Omid Jalali¹, Gregory Mundy²,
Beau Gilkerson¹, Martin Feather³, and James Kiper⁴

¹ West Virginia University, Morgantown, WV, USA

² Alderson-Broaddus College, Philippi, WV

³ Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA

⁴ Dept. of Computer Science and Systems Analysis, Miami University, Oxford, OH, USA

greg@greggay.com, tim@menzies.us, jalali.omid@gmail.com,
beau.gilkerson@gmail.com, mundyge@ab.edu,
martin.s.feather@jpl.nasa.gov, kiperjd@muohio.edu

Abstract. Solutions to non-linear requirements engineering problems may be “brittle”; i.e. small changes may dramatically alter solution effectiveness. Hence, it is not enough to just generate solutions to requirements problems- we must also assess solution robustness. The KEYS2 algorithm can generate *decision ordering diagrams*. Once generated, these diagrams can assess solution robustness in linear time. In experiments with real-world requirements engineering models, we show that KEYS2 can generate decision ordering diagrams in $O(N^2)$. When assessed in terms of terms of (a) reducing inference times, (b) increasing solution quality, and (c) decreasing the variance of the generated solution, KEYS2 out-performs other search algorithms (simulated annealing, ASTAR, MaxWalkSat).

1 Introduction

Consider a “requirements model” where stakeholders write:

- Their various goals;
- Their different methods to reach those goals;
- Their view of the possible risks that compromise those goals;
- What mitigations they believe might reduce those risks.

A “solution” to such models is a *least* cost set of mitigations that reduce the *most* risks, thereby enabling the *most* requirements. In theory, software can find the solution that best exploits and most satisfies the various goals of our different stakeholders. Such tools might find good solutions that were missed by stakeholders. Finding solutions to these requirements models is a non-linear optimization problem (*minimize* the sum of the mitigation costs while *maximizing* the number of achieved requirements).

There are many heuristic methods that can generate solutions to non-linear problems (see *Related Work*, below). However, such heuristic methods can be *brittle*; i.e. small changes may dramatically alter the effectiveness of the solution. Therefore, it important to understand the *neighborhood around the solution*. A naive approach to understanding the neighborhood might be to run a system N times then report:

- the solutions appearing in more than (say) $\frac{N}{2}$ cases;
- results with a $\pm 95\%$ confidence interval.

Note that both these approaches requires multiple runs of an analysis method. Multiple runs are undesirable since, in our experience [20], stakeholders often ask questions across a range of “scenarios”; i.e. hard-wired constraints that cannot be changed in that scenario. For example, three scenarios might be “what can be achieved assuming a maximum budget of one, two, or five million dollars?”. Scenario analysis can be time consuming. Reflecting over (say) $d = 10$ possible decisions a statistically significant number of times (e.g. $N = 20$) requires up to $20 * 2^{10} > 20,000$ repeats of the analysis.

Therefore, this paper proposes a rapid method for exploring decision neighborhoods. *Decision ordering diagrams* are a visual representation of the effects of changing a solution. We show below that:

- Using these diagrams, the region around a solution can be explored in linear time.
- A greedy Bayesian-based method called KEYS2 can generate the decision ordering diagrams in $O(N^2)$ time.
- KEYS2 yields solutions of higher quality than several other methods (simulated annealing, MaxWalkSat, ASTAR).
- Also, the variance of the solutions found by KEYS2 is less (and hence, better) than those found by the other methods.

This paper is structured as follows:

- After some background notes on the solution robustness, we describe the *DDP* requirements engineering tool used at NASA’s Jet Propulsion Laboratory (the case studies for this paper come from real-world early lifecycle DDP models);
- DDP inputs and outputs are then reviewed;
- Next, decision ordering diagrams are introduced;
- We then define and compare five different algorithms for generating solutions from DDP models: KEYS, KEYS2, Simulated Annealing, MaxWalkSat, ASTAR. Experimental results will show that KEYS2 out-performs the other methods (measured in terms of quickly generating high quality solutions that allow us to reflect over solution robustness).
- Finally, we offer some notes on related work and conclusions.

This paper extends a prior publication [39] in several ways:

- That paper did not address concerns of solution robustness.
- That paper did not explore a range of alternate algorithms.
- This paper introduces KEYS2, which is an improved version of KEYS.,
- This paper offers extensive notes on related work.

2 Background

According to Harman [30], understanding the neighborhood of solutions is an open and pressing issue in search-based software engineering (SBSE). He argues that many software engineering problems are over-constrained and no precise solution over all variables is achievable; therefore partial solutions based on heuristic search methods are preferred. *Solution robustness* is a major problem for such partial heuristic searches.

The results of such partial heuristic search may be “brittle”; i.e. small changes to the search results may dramatically alter the effectiveness of the solution [31].

When offering partial solutions, it is very important to also offer insight into the space of options around the proposed solution. Such neighborhood information is very useful for managers with only partial control over their projects since it can give them confidence that even if only some of their recommendations are effected, then at least the range of outcomes is well understood. Harman [30] comments that understanding the neighborhood of our solutions is an open and pressing issue in search-based software engineering (SBSE):

“In some software engineering applications, solution robustness may be as important as solution functionality. For example, it may be better to locate an area of the search space that is rich in fit solutions, rather than identifying an even better solution that is surrounded by a set of far less fit solutions.”

“Hitherto, research on SBSE has tended to focus on the production of the fittest possible results. However, many application areas require solutions in a search space that may be subject to change. This makes robustness a natural second order property to which the research community could and should turn its attention [30].”

This paper reports a set of experiments on AI search for robust solutions. Our experiments address two important concerns. Firstly, *is demonstrating solution robustness a time consuming task?* Secondly, is it necessary, as Harman suggests that *solution quality must be traded off against solution robustness*? That is, in our search for the conclusions that were stable within their local neighborhood, would we have to reject better solutions because they are less stable across their local neighborhood?

At least for the NASA models described in the next section, both these concerns are unfounded. KEYS2 terminates in hundredths of a second (whereas our prior implementations took minutes to terminate [22]). Also, the solutions found by KEYS2 were not only of highest quality, they were also exhibited the lowest variance. Further, KEYS2 generates the decision ordering diagrams that can assess solution robustness in linear time.

3 Requirements Modeling Using “DDP”

This section introduces the DDP requirements modeling tool [15, 20], used to interactively document the “Team-X” early lifecycle meetings at NASA’s Jet Propulsion Laboratory (JPL). These meetings are the source of the real-world requirements models used in this paper.

At “Team X” meetings, a large and diverse group of up to 30 experts from various fields (propulsion, engineering, communication, navigation, science, etc) meet for very short periods of time (hours to days) to produce a “mission concept” document. This document may commit the project to, say, solar power rather nuclear power or a particular style of guidance software. All of the subsequent work is based on the initial decisions documented in the mission concept.

-
1. Requirement goals:
 - Spacecraft ground-based testing & flight problem monitoring
 - Spacecraft experiments with on-board Intelligent Systems Health Management (ISHM)
 2. Risks:
 - Obstacles to spacecraft ground-based testing & flight problem monitoring
 - Customer has no, or insufficient, money available for our use
 - Difficulty of building the models / design tools
 - ISHM Experiment is a failure (without necessarily causing flight failure)
 - Usability, User/Recipient-system interfaces undefined
 - V&V (certification path) untried and scope unknown
 - Obstacles to Spacecraft experiments with on-board ISHM
 - Bug tracking / fixes / configuration management issues, Managing revisions and upgrades (multi-center tech. development issue)
 - Concern about our technology interfering with in-flight mission
 3. Mitigations:
 - Mission-specific actions
 - Spacecraft ground-based testing & flight problem monitoring
 - Become a team member on the operations team
 - Use Bugzilla and CVS
 - Spacecraft experiments with on-board ISHM
 - Become a team member on the operations team
 - Utilize xyz's experience and guidance with certification of his technology
-

Fig. 1. Sample DDP requirements, risks, mitigations.

DDP allows for the representation of the goals, risks, and risk-removing mitigations that belong to a specific project. During a Team X meeting, users of DDP explore combinations of mitigations that cost the least and support the most number of requirements. DDP propagate influences over matrices. For example, here is a trivial DDP model where *mitigation1* costs \$10,000 to apply and each requirement is of equal value (100). Note that the mitigation can remove 90% of the risk. Also, unless mitigated, the risk will disable 10% to 99% of requirements one and two (respectively):

$$\overbrace{\text{mitigation1}}^{\$10,000} \rightarrow \text{risk1} \rightarrow \begin{cases} \nearrow 0.1 \\ \searrow 0.9 \end{cases} \begin{cases} \nearrow (requirement1 = 100) \\ \searrow (requirement2 = 100) \end{cases} \quad (1)$$

The other numbers show the impact of mitigations on risks, and the impact of risks on requirements. The core of DDP is two matrices: one for *mitigations*risks* and another for *risks*requirements*.

DDP is used as follows. A dozen experts, or more, gather together for short, intensive knowledge acquisition sessions (typically, 3 to 4 half-day sessions). These sessions *must* be short since it is hard to gather together these experts for more than a very short period of time. The DDP tool supports a graphical interface for the rapid entry of the assertions. Such rapid entry is essential, time is crucial and no tool should slow the debate.

Therefore, DDP uses a lightweight representations for its model. Such representations are essential for early lifecycle decision making since only high-level assertions can be collected in short knowledge acquisition sessions (if the assertions get more elaborate, then experts may be unable to understand technical arguments from outside their own field of expertise). Therefore, DDP uses the following lightweight ontology:

- *Requirements* (free text) describing the objectives and constraints of the mission and its development process;
- *Weights* (numbers) of each requirements, reflecting their relative importance;
- *Risks* (free text) are events that damage requirements;
- *Mitigations*: (free text) are actions that reduce risks;
- *Costs*: (numbers) effort associated with mitigations, and repair costs for correcting Risks detected by Mitigations;
- *Mappings*: directed edges between requirements, mitigations, and risks that capture quantitative relationships among them.
- *Part-of relations* structure the collections of requirements, risks and mitigations;

Note that DDP models are the same as the “requirements models” we defined in the introduction. For examples of risks, requirements, and mitigations, see Figure 1. For an example of the network of connections between risks and requirements and mitigations, see Figure 2.

Sometimes, we are asked if the analysis of DDP requirements models is a real problem. The usual question is something like: “With these ultra-lightweight languages, aren’t all open issues just obvious?”. Such a question is usually informed by the small model fragments that appear in the ultra-lightweight modeling literature. Those sample model fragments are typically selected according to their ability to fit on a page or to succinctly illustrate some point of the authors. Real world ultra-lightweight models can be much more complex, paradoxically perhaps due to their simplicity: if a model is easy to write then it is easy to write a lot of it. Figure 2, for example, was generated in under a week by four people discussing one project. It is complex and densely-connected (a close inspection of the left and right hand sides of Figure 2 reveals the requirements and fault trees that inter-connect concepts in this model) and it is, by no means, the biggest or most complex DDP model that has ever been built.

We base our experimentation around DDP for three reasons. Firstly, one potential drawback with ultra-lightweight models is that they are excessively lightweight and contain no useful information. DDP’s models are demonstrably useful (that is, we are optimizing a real-world problem of some value). Clear project improvements have been seen from DDP sessions at JPL. Cost savings in at least two sessions have exceeded \$1 million, while savings of over \$100,000 have resulted in others [20]. Cost savings are not the only benefits of these DDP sessions. Numerous design improvements such as savings of power or mass have come out of DDP sessions. Likewise, a shifting of risks has been seen from uncertain architectural ones to more predictable and manageable ones. At some of these meetings, non-obvious significant risks have been identified and subsequently mitigated.

Our second reason to use DDP is that we can access numerous real-world requirements models written in this format, both now and in the future. The DDP tool can be used to document not just final decisions but also to review the rationale that led to those

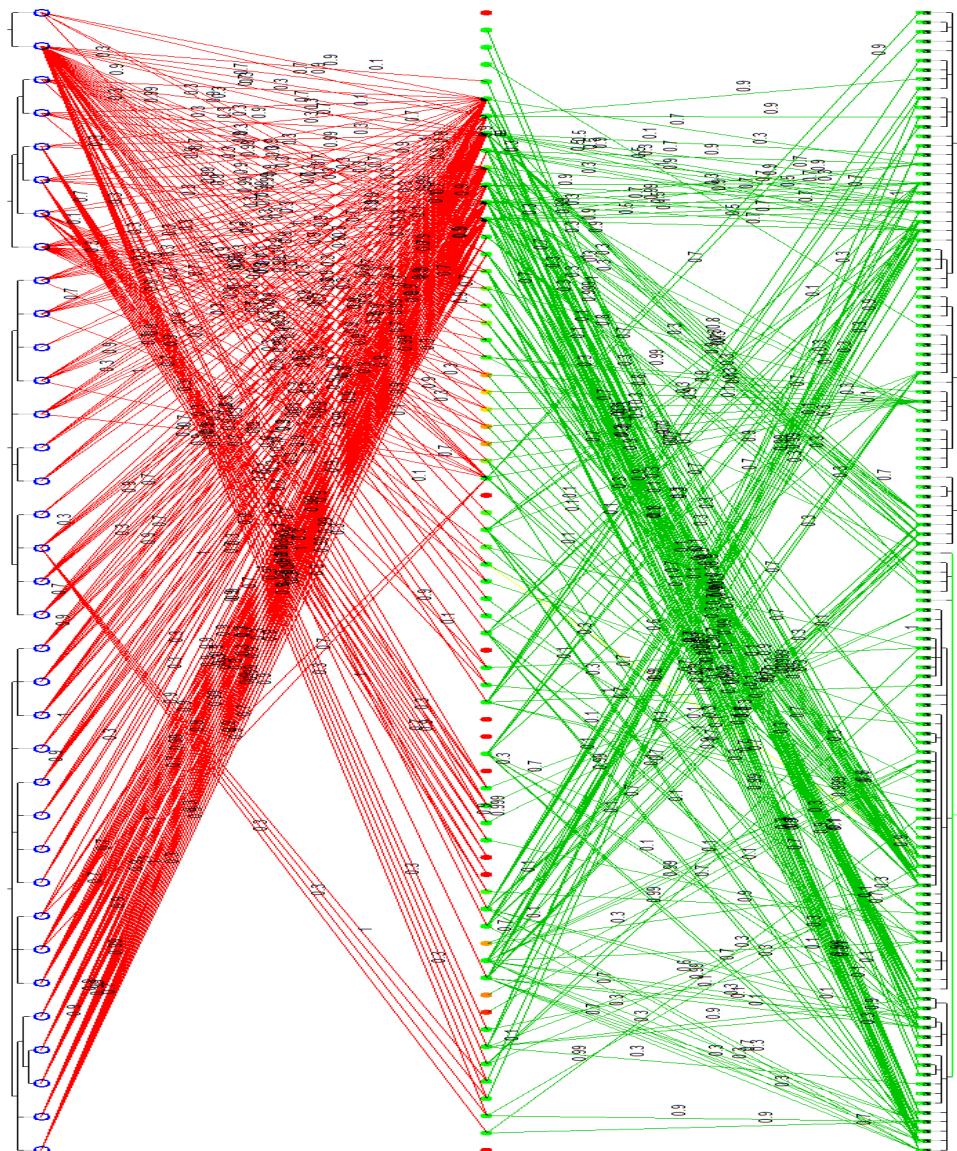


Fig. 2. An example of a model formed by the DDP tool. Red lines connect risks (middle) to requirements (left). Green lines connect mitigations (right) to the risks.

decisions. Hence, DDP remains in use at JPL: not only for its original purpose (group decision support), but also as a design rationale tool to document decisions. Recent DDP sessions included:

- An identification of the challenges of intelligent systems health management (ISHM) technology maturation (to determine the most cost-effective approach to achieving maturation) [23];
- A study on the selection and planning of deployment of prototype software [21].

Our third, and most important reason to use DDP in our research is that the tool is representative of other requirements modeling tools in widespread use. At its core, DDP is a set of influences expressed in a hierarchy, augmented with the occasional equation. Edges in the hierarchy have weights that strengthen or weaken influences that flow along those edges. At this level of abstraction, DDP is just another form of QOC [64] or a quantitative variant of Mylopoulos' s qualitative soft goal graphs [54].

4 Model Inputs and Outputs

Before describing experimental comparisons of different methods for generating decision ordering diagrams, we will first offer more details on the DDP models.

4.1 Pre-Processing

To enable fast runtimes, a simple compiler exports the DDP models into a form accessible by our algorithms. This compiler stores a flattened form of the DDP requirements tree. In our compiled form, all computations are performed once and added as a constant to each reference of the requirement. For example, the compiler converts the trivial model of Equation 1 into `setupModel` and `model` functions similar to those in Figure 3. The `setupModel` function is called only once and sets several constant values. The `model` function is called whenever new cost and attainment values are needed. The topology of the mitigation network is represented as terms in equations within these functions. As our models grow more complex, so do these equations. For example, our biggest model, which contains 99 mitigations, generates 1427 lines of code. Figure 4 compares the largest model to four other real-world DDP models.

Currently it takes about two seconds to compile a model with 50 requirements, 31 risks, and 58 mitigations. This compilation only has to happen once, after which we will run our $2^{|d|}$ what-if scenarios. While this is not a significant bottleneck, the current compiler (written in unoptimized Visual Basic code) can certainly be sped up. Experts usually change a small portion of the model then run $2^{|d|}$ what-if scenarios to understand the impact of that change. Therefore, an incremental compiler (that only updates changed portions) would run much faster than a full compilation of the entire DDP model.

4.2 Objective Function

When the `model` function is called, a pairing of the total cost of the selected mitigations and the number of reachable requirements (attainment) is returned. All of our

```

#include "model.h"

#define M_COUNT 2
#define O_COUNT 3
#define R_COUNT 2

struct ddpStruct
{
    float oWeight[O_COUNT+1];
    float oAttainment[O_COUNT+1];
    float oAtRiskProp[O_COUNT+1];
    float rAPL[R_COUNT+1];
    float rLikelihood[R_COUNT+1];
    float mCost[M_COUNT+1];
    float roImpact[R_COUNT+1][O_COUNT+1];
    float mrEffect[M_COUNT+1][R_COUNT+1];
};

ddpStruct *ddpData;

void setupModel(void)
{
    ddpData = (ddpStruct *) malloc(sizeof(ddpStruct));
    ddpData->mCost[1]=11;
    ddpData->mCost[2]=22;
    ddpData->rAPL[1]=1;
    ddpData->rAPL[2]=1;
    ddpData->oWeight[1]=1;
    ddpData->oWeight[2]=2;
    ddpData->oWeight[3]=3;
    ddpData->roImpact[1][1] = 0.1;
    ddpData->roImpact[1][2] = 0.3;
    ddpData->roImpact[2][1] = 0.2;
    ddpData->mrEffect[1][1] = 0.9;
    ddpData->mrEffect[1][2] = 0.3;
    ddpData->mrEffect[2][1] = 0.4;
}

void model(float *cost, float *att, float m[])
{
    float costTotal, attTotal;
    ddpData->rLikelihood[1] = ddpData->rAPL[1] * (1 - m[1] * ddpData->mrEffect[1][1])
        * (1 - m[2] * ddpData->mrEffect[2][1]);
    ddpData->rLikelihood[2] = ddpData->rAPL[2] * (1 - m[1] * ddpData->mrEffect[1][2]);
    ddpData->oAtRiskProp[1] = (ddpData->rLikelihood[1] * ddpData->roImpact[1][1])
        + (ddpData->rLikelihood[2] * ddpData->roImpact[2][1]);
    ddpData->oAtRiskProp[2] = (ddpData->rLikelihood[1] * ddpData->roImpact[1][2]);
    ddpData->oAtRiskProp[3] = 0;
    ddpData->oAttainment[1] = ddpData->oWeight[1] * (1 - minValue(1, ddpData->oAtRiskProp[1]));
    ddpData->oAttainment[2] = ddpData->oWeight[2] * (1 - minValue(1, ddpData->oAtRiskProp[2]));
    ddpData->oAttainment[3] = ddpData->oWeight[3] * (1 - minValue(1, ddpData->oAtRiskProp[3]));
    attTotal = ddpData->oAttainment[1] + ddpData->oAttainment[2] + ddpData->oAttainment[3];
    costTotal = m[1] * ddpData->mCost[1] + m[2] * ddpData->mCost[2];

    *cost = costTotal;
    *att = attTotal;
}

```

Fig. 3. A trivial DDP model after knowledge compilation

Model	LOC	Objectives	Risks	Mitigations
model1.c	55	3	2	2
model2.c	272	1	30	31
model3.c	72	3	2	3
model4.c	1241	50	31	58
model5.c	1427	32	70	99

Fig. 4. Details of Five DDP Models.

algorithms then use that information to obtain a “score” for the current set of mitigations. The two numbers are normalized to a single score that represents the distance to a *sweet spot* of maximum requirement attainment and minimum cost:

$$score = \sqrt{cost^2 + (attainment - 1)^2} \quad (2)$$

Here, \bar{x} is a normalized value $0 \leq \frac{x-min(x)}{max(x)-min(x)} \leq 1$. Hence, our scores ranges $0 \leq score \leq \sqrt{2}$ and *lower* scores are *better*.

4.3 Decision Ordering Diagrams

The objective function described above summarizes *one* call to a DDP model. This section describes *decision ordering diagrams*, which are a tool for summarizing the results of thousands of calls to DDP models.

Consider some recommendation for changes to a project that requires decisions d of size $|d|$. In the general case, d is a subset of the space of all solutions D ($d \subseteq D$). When checking for solution robustness, or reflecting over modifications to d , a stakeholder may need to consider up to $d' \subseteq N^{|d|}$ possibilities (and $N = 2$ for binary decisions of the form “should I or should I not do this”). This can be a slow process, especially if evaluating each decision requires invoking a complex and slow simulator.

Decision ordering diagrams are a linear time method for studying the robustness and neighborhood of a set of decisions. The diagrams assume that some method could offer a *linear ordering* of the decisions $x \in d$ ranked from *most-important* to *least-important*. They also assume that some method offers information on the effects of applying the top-ranked $1 \leq x \leq |d|$ decisions (e.g. the median and variance seen in the model’s objective function after applying solution $\{d_1..d_x\}$). For example, the *decision ordering diagram* of Figure 5 shows such a linear ordering (this figure presents *benefit* and *cost* results). In that figure:

- The x-axis denotes the number of decisions made.
- The y-axis shows performance statistics of an objective function seen after imposing the conjunction of decisions $1 \leq i \leq x$.

For performance, we run some objective function and report the median (50th percentile) and *spread* (the range given by the 75th percentile - the 50th percentile). We use median and spread to avoid any parametric assumptions.

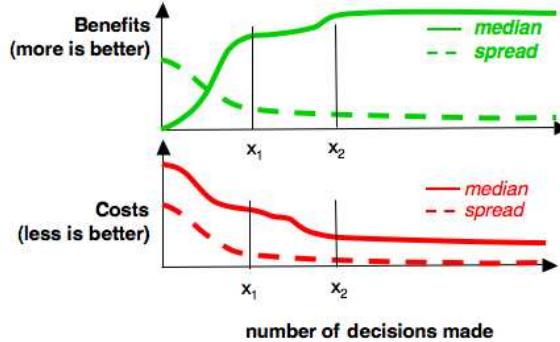


Fig. 5. A Decision Ordering Diagram. The median and spread plots show 50%-the percentile and the (75-50)%-th percentile range (respectively) values generated from some objective function.

These diagrams can comment on the robustness and neighborhood of solution $\{d_1..d_x\}$ as follows:

- By considering the variance of the performance statistics after applying $\{d_1..d_x\}$.
- By comparing the results of using the first x decisions to that of using the first $x - 1$ or $x + 1$ actions.

The neighborhood of a solution that uses decisions $\{d_1..d_x\}$ are solutions that use the decisions $\{d_1..d_{x \pm j}\}$. Since j is bounded $0 \leq |j| - 1$, this means that *reflecting over solution neighborhoods takes time linear on the number of decisions d*.

Decision ordering diagrams are a natural representation for “trade studies,” the activity of a multidisciplinary team to identify the most balanced technical solution among a set of proposed viable solutions [2]. For example, minimum costs and maximum benefits are achieved at point x_2 of Figure 5. However, after applying only half the decisions (see x_1) most of the benefits could be achieved, albeit at a somewhat higher cost.

Decision ordering diagrams are *useful* under at least three conditions:

- The scores output by the objective functions are *well-behaved*; i.e. move smoothly to a plateau.
- The decisions *tame* the variance; i.e. the spread falls to value much lower than then median (otherwise, it is hard to show that decisions have any effect on the system performance).
- The are generated in a *timely* manner. Fast runtimes are required in order to keep up with fast moving discussion.

According to these definitions, Figure 5 is a *useful* decision ordering diagram if it can be generated in a *timely* manner.

It is an open issue if real worlds requirements models generate useful decision ordering diagrams. The following experiments test if, in practice, decision ordering diagrams generated from real world requirements models are *timely* to generate while being *well-behaved* and *tame*.

5 Searching for Solutions

Our experiments compare the results of numerous algorithms. We selected these comparison algorithms with much care. Numerous researchers have stressed the difficulties associated with comparing radically different algorithms. For example, Uribe and Stickel [67] tried to make some general statement about the value of Davis-Putnam proof (DP) procedures or binary-decision diagrams (BDD) for constraint satisfaction problems. In the end, they doubted that it was fair to compare algorithms that perform constraint satisfaction and no search (like BDDs) and methods that perform search and no constraint satisfaction (like DP). For this reason, model checking researchers like Holzmann (pers. communication) eschew comparisons of tools like SPIN [35], which are search-based, with tools like NuSMV [11], which are BDD-based. Hence we take care to only select algorithms which are similar to KEYS.

In terms of the Gu et al. survey [27], our selected algorithms (simulated annealing, ASTAR and MaxWalkSat) share four properties with KEYS and KEYS2. They are each discrete, sequential, *unconstrained* algorithms (constrained algorithms work towards a pre-determined number of possible solutions while unconstrained methods are allowed to adjust to the goal space).

For full details on simulated annealing, ASTAR, and MaxFunWalk, see below. We observe that these algorithms share the property that at each step of their processing, they comment on all model inputs. KEYS2, on the other hand, explores the consequences of setting only a subset of the possible inputs.

5.1 SA: Simulated Annealing

Simulated Annealing is a classic stochastic search algorithm. It was first described in 1953 [51] and refined in 1983 [43]. The algorithm’s namesake, annealing, is a technique from metallurgy, where a material is heated, then cooled. The heat causes the atoms in the material to wander randomly through different energy states and the cooling process increases the chances of finding a state with a lower energy than the starting position.

For each round, SA “picks” a neighboring set of mitigations. To calculate this neighbor, a function traverses the mitigation settings of the current state and randomly flips those mitigations (at a 5% chance). If the neighbor has a better score, SA will move to it and set it as the current state. If it isn’t better, the algorithm will decide whether to move to it based on the mathematical function:

$$prob(w, x, y, temp(y, z)) = e^{((w-x)*\frac{y}{temp(y, z)})} \quad (3)$$

$$temp(y, z) = \frac{(z - y)}{z} \quad (4)$$

If the value of the `prob` function is greater than a randomly generated number, SA will move to that state anyways. This randomness is the very cornerstone of the Simulated Annealing algorithm. Initially, the “atoms” (current solutions) will take large random jumps, sometimes to even sub-optimal new solutions. These random jumps allow simulated annealing to sample a large part of the search space, while avoiding being trapped in local minima. Eventually, the “atoms” will cool and stabilize and the search will converge to a simple hill climber.

```

1. Procedure SA
2. MITIGATIONS:= set of mitigations
3. SCORE:= score of MITIGATIONS
4. while TIME < MAX_TIME && SCORE < MIN_SCORE //minScore is a constant score (threshold)
5.   find a NEIGHBOR close to MITIGATIONS
6.   NEIGHBOR_SCORE:= score of NEIGHBOR
7.   if NEIGHBOR_SCORE > SCORE
8.     MITIGATIONS:= NEIGHBOR
9.     SCORE:= NEIGHBOR_SCORE
10.    else if prob(SCORE, NEIGHBOR_SCORE, TIME, temp(TIME, MAX_TIME)) > RANDOM
11.      MITIGATIONS:= NEIGHBOR
12.      SCORE:= NEIGHBOR_SCORE
13.    TIME++
14.  end while
15. return MITIGATIONS

```

Fig. 6. Pseudocode for SA

As shown in line 4 of Figure 6, the algorithm will continue to operate until the number of tries is exhausted or a score meets the threshold requirement.

5.2 MaxFunWalk

The design of simulated annealing dates back to the 1950s. In order to benchmark our own search engine (KEYS2) against a more state-of-the-art algorithm, we implemented the variant of MaxWalkSat described below.

WalkSat is a local search method designed to address the problem of boolean satisfiability [42]. MaxWalkSat is a variant of that algorithm that applies weights to each clause in a conjunctive normal form equation [62]. While WalkSat tries to satisfy the entire set of clauses, MaxWalkSat tries to maximize the sum of the weights of the satisfied clauses.

In one respect, both algorithms can be viewed as a variant of simulated annealing. Whereas simulated annealing always selects the next solution randomly, the WalkSat algorithms will *sometimes* perform random selection while, other times, conduct a local search to find the next best setting to one variable.

MaxFunWalk is a generalization of MaxWalkSat:

- MaxWalkSat is defined over CNF formulae. The success of a collection of variable settings is determined by how many clauses are “satisfiable” (defined using standard boolean truth tables).
- MaxWalkFun, on the other hand, assumes that there exist an arbitrary function that can assess a collection of variable settings. Here, we use the DDP model as a assessment function.

Note that *MaxWalkFun* = *MaxWalkSat* if the assessment is conducted via a logical truth table.

The MaxFunWalk procedure is shown in Figure 7. When run, the user supplies an ideal cost and attainment. This setting is normalized, scored, and set as a goal threshold. If the current setting of mitigations satisfies that threshold, the algorithm terminates.

```

1. Procedure MaxFunWalk
2. for TRIES:=1 to MAX-TRIES
3.   SELECTION:=A randomly generate assignment of mitigations
4.   for CHANGED:=1 to MAX-CHANGES
5.     if SCORE satisfies THRESHOLD return
6.     CHOSEN:= a random selection of mitigations from SELECTION
7.     with probability P
8.       flip a random setting in CHOSEN
9.     with probability (P-1)
10.      flip a setting in CHOSEN that maximizes SCORE
11.   end for
12. end for
13. return BESTSCORE

```

Fig. 7. Pseudocode for MaxFunWalk

MaxFunWalk begins by randomly setting every mitigation. From there, it will attempt to make a *single* change until the threshold is met or the allowed number of changes runs out (100 by default). A random subset of mitigations is chosen and a random number P between 0 and 1 is generated. The value of P will decide the form that the change takes:

- $P \leq \alpha$: A stochastic decision is made. A setting is changed completely at random within the set CHOSEN.
- $P > \alpha$: Local search is utilized. Each mitigation in CHOSEN is tested until one is found that improves the current score.

The best setting of α is domain-specific. For this study, we used $\alpha = 0.3$.

If the threshold is not met by the time that the allowed number of changes is exhausted, the set of mitigations is completely reset and the algorithm starts over. This measure allows the algorithm to avoid becoming trapped in local maxima. For the DDP models, we found that the number of retries has little effect on solution quality.

If the threshold is never met, MaxFunWalk will reset and continue to make changes until the maximum number of allowed resets is exhausted. At that point, it will return the best settings found.

As an additional measure to improve the results found by MaxFunWalk, a heuristic was implemented to limit the number of mitigations that could be set at one time. If too many are set, the algorithm will turn off a few in an effort to bring the cost factor down while minimizing the effect on the attainment.

5.3 A* (ASTAR)

A* is a best-first path finding algorithm that uses distance from origin (G) and estimated cost to goal (H) to find the best path [33]. The algorithm is widely used [36, 57, 59, 66].

A* is a natural choice for DDP optimization since the objective function described above is actually a Euclidean distance measure to the desired goal of maximum attainment and minimum costs. Hence, for the second portion of the ASTAR heuristic, we can make direct use of Equation 2.

The ASTAR algorithm keeps a *closed* list in order to prevent backtracking. We begin by adding the starting state to the closed list. In each “round,” a list of neighbors is populated from the series of possible states reached by making a change to a single mitigation. If that neighbor is not on the closed list, two calculations are made:

- G = Distance from the start to the current state plus the additional distance between the current state and that neighbor.
- H = Distance from that neighbor to the goal (an ideal spot, usually 0 cost and a high attainment). For DDP models, we use Equation 2 to compute H.

The *best* neighbor is the one with the lowest F=G+H. The algorithm “travels” to that neighbor and adds it to the closed list. Part of the optimality of the A* algorithm is that the distance to the goal is underestimated. Thus, the final goal is never actually reached by ASTAR. Our implementation terminates once it stops finding better solutions for a total of ten rounds. This number was chosen to give ample time for ASTAR to become “unstuck” if it hits a corner early on.

```

1. Procedure ASTAR
2. CURRENT_POSITION:= Starting assignment of mitigations
3. CLOSED[0]:= Add starting position to closed list
4.
5. while END:= false
6.   NEIGHBOR_LIST:=list of neighbors
7.   for each NEIGHBOR in NEIGHBOR_LIST
8.     if NEIGHBOR is not in CLOSED
9.       G:=distance from start
10.      H:=distance to goal
11.      F:=G+H
12.      if F<BEST_F
13.        BEST_NEIGHBOR:=NEIGHBOR
14.   end for
15.   CURRENT_POSITION:= BEST_NEIGHBOR
16.   CLOSED[++]:=Add new state to closed list
17.   if STUCK
18.     END:= true
19. end while
20. return CURRENT_POSITION

```

Fig. 8. Pseudocode for ASTAR

5.4 KEYS and KEYS2

The core premise of KEYS and KEYS2 is that the above algorithms perform over-elaborate searches. Suppose that the behavior of a large system is determined by a small number of *key* variables. If so, then a very rapid search for solutions can be found by (a) finding these *keys* then (b) explore the ranges of the key variables.

As documented in our *Related Work* section, this notion of *keys* has been discovered and rediscovered many times by many researchers. Historically, finding the keys

has seen to be a very hard task. For example, finding the keys is analogous to finding the *minimal environments* of DeKleer' ATMS algorithm [17]. Formally, this *logical abduction*, which is an NP-hard task [8].

Our method for finding the keys uses a Bayesian sampling method. If a model contains keys then, by definition, those variables must appear in all solutions to that model. If model outputs are scored by some oracle, then the key variables are those with ranges that occur with very different frequencies in high/low scored model outputs. Therefore, we need not search for the keys- rather, we just need to keep frequency counts on how often ranges appear in *best* or *rest* outputs.

KEYS contains an implementation of this Bayesian sampling method. It has two main components - a greedy search and the BORE ranking heuristic. The greedy search explores a space of M mitigations over the course of M “eras.” Initially, the entire set of mitigations is set randomly. During each era, one more mitigation is set to $M_i = X_j$, $X_j \in \{\text{true}, \text{false}\}$. In the original version of KEYS [48], the greedy search fixes one variable per era. A newer variant, KEYS2, fixes an increasing number of variables as the search progresses (see below for details).

In KEYS (and KEYS2), each era e generates a set $\langle \text{input}, \text{score} \rangle$ as follows:

- 1: MaxTries times repeat:
 - $\text{Selected}[1\dots(e-1)]$ are settings from previous eras.
 - Guessed are randomly selected values for unfixed mitigations.
 - $\text{Input} = \text{selected} \cup \text{guessed}$.
 - Call model_1 to compute $\text{score} = \text{ddp}(\text{input})$;
- 2: The MaxTries scores are divided into $\beta\%$ “best” and remainder become “rest”.
- 3: The input mitigation values are then scored using BORE (described below).
- 4: The top ranked mitigations (the default is one, but the user may fix multiple mitigations at once) are fixed and stored in $\text{selected}[e]$.

```

1. Procedure KEYS
2. while FIXED_MITIGATIONS != TOTAL_MITIGATIONS
3.   for I:=1 to 100
4.     SELECTED[1\dots(I-1)] = best decisions up to this step
5.     GUESSED = random settings to the remaining mitigations
6.     INPUT = SELECTED + GUESSED
7.     SCORES= SCORE(INPUT)
8.   end for
9.   for J:=1 to NUM_MITIGATIONS_TO_SET
10.    TOP_MITIGATION = BORE(SCORES)
11.    SELECTED[FIXED_MITIGATIONS++] = TOP_MITIGATION
12.  end for
13. end while
14. return SELECTED

```

Fig. 9. Pseudocode for KEYS

The search moves to era $e + 1$ and repeats steps 1,2,3,4. This process stops when every mitigation has a setting. The exact settings for MaxTries and β must be set via

engineering judgment. After some experimentation, we used $MaxTries = 100$ and $\beta = 10$. For full details, see Figure 9.

KEYS ranks mitigations using a support-based Bayesian ranking measure called BORE. BORE [12] (short for “best or rest”) divides numeric scores seen over K runs and stores the top 10% in *best* and the remaining 90% scores in the set *rest* (the *best* set is computed by studying the delta of each score to the best score seen in any era). It then computes the probability that a value is found in *best* using Bayes theorem. The theorem uses evidence E and a prior probability $P(H)$ for hypothesis $H \in \{\text{best}, \text{rest}\}$, to calculate a posteriori probability $P(H|E) = P(E|H)P(H) / P(E)$. When applying the theorem, *likelihoods* are computed from observed frequencies. These likelihoods (called “like” below) are then normalized to create probabilities. This normalization cancels out $P(E)$ in Bayes theorem. For example, after $K = 10,000$ runs are divided into 1,000 *best* solutions and 9,000 *rest*, the value *mitigation31 = false* might appear 10 times in the *best* solutions, but only 5 times in the *rest*. Hence:

$$\begin{aligned}
E &= (\text{mitigation31} = \text{false}) \\
P(\text{best}) &= 1000/10000 = 0.1 \\
P(\text{rest}) &= 9000/10000 = 0.9 \\
\text{freq}(E|\text{best}) &= 10/1000 = 0.01 \\
\text{freq}(E|\text{rest}) &= 5/9000 = 0.00056 \\
\text{like}(\text{best}|E) &= \text{freq}(E|\text{best}) \cdot P(\text{best}) = 0.001 \\
\text{like}(\text{rest}|E) &= \text{freq}(E|\text{rest}) \cdot P(\text{rest}) = 0.000504 \\
P(\text{best}|E) &= \frac{\text{like}(\text{best}|E)}{\text{like}(\text{best}|E) + \text{like}(\text{rest}|E)} = 0.66
\end{aligned} \tag{5}$$

Previously [12], we have found that Bayes theorem is a poor ranking heuristic since it is easily distracted by low frequency evidence. For example, note how the probability of E belonging to the best class is moderately high even though its support is very low; i.e. $P(\text{best}|E) = 0.66$ but $\text{freq}(E|\text{best}) = 0.01$.

To avoid the problem of unreliable low frequency evidence, we augment Equation 5 with a support term. Support should *increase* as the frequency of a value *increases*, i.e. $\text{like}(\text{best}|E)$ is a valid support measure. Hence, step 3 of our greedy search ranks values via

$$P(\text{best}|E) * \text{support}(\text{best}|E) = \frac{\text{like}(\text{best}|E)^2}{\text{like}(\text{best}|E) + \text{like}(\text{rest}|E)} \tag{6}$$

For each era, KEYS samples the DDP models and fixes the top $N = 1$ settings. KEYS2 assigns progressively larger values. In era 1, KEYS2 behaves exactly the same as KEYS while in (say) era 3, KEYS2 will fix the top 3 ranked ranges. Since it sets more variables at each era, KEYS2 terminates earlier than KEYS.

Note that decision ordering diagrams could be directly generated during execution, just by collection statistics from the SCORES array used in line 7 of Figure 9.

6 Results

Each of the above algorithms was tested on the five models of Figure 4. Note that:

- Models one and three are trivially small. They were used them to debug our code, but not in the core experiments. We report our results using models two, four and five since they are large enough to stress test real-time optimization.
- Model 4 was discussed in [49] in detail. The largest, model 5 was optimized previously in [22]. At that time (2002), it took 300 seconds to generate solutions using our old, very slow, rule learning method.

We also studied how well KEYS and KEYS2 scale to larger models. Further, we instrumented KEYS and KEYS2 to generate decision ordering diagrams. The results from all of these experiments are shown below.

6.1 Attainment and Costs

We ran all of our algorithms 1000 times on each model. This number was chosen because it yielded enough data points to give a clear picture of the span of results. At the same time, it is a low enough number that we can generate a set of results in a fairly short time span.

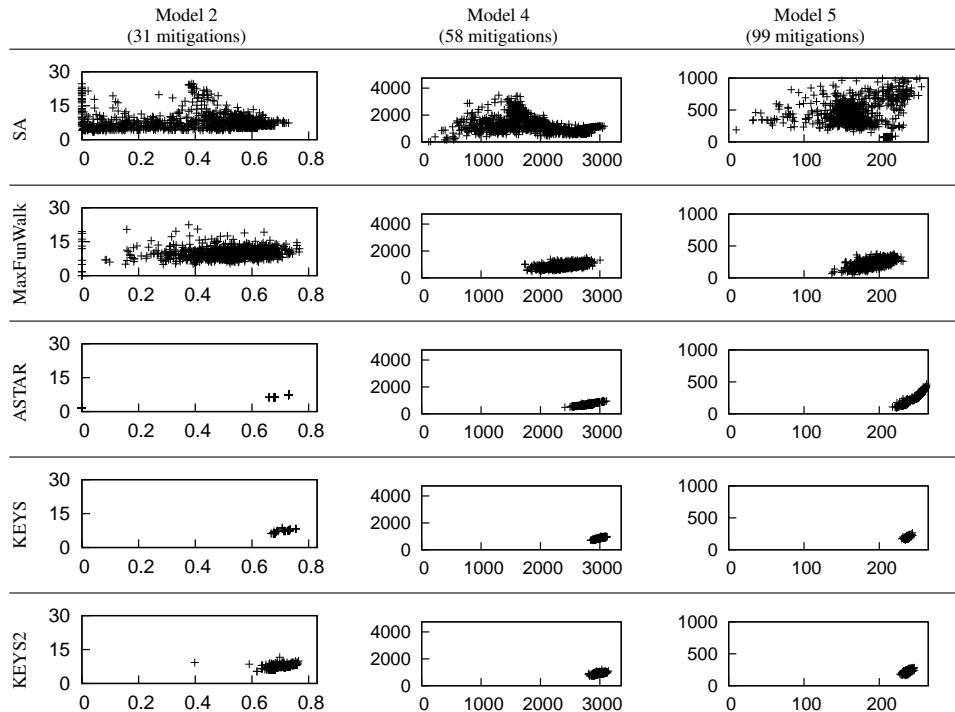


Fig. 10. 1000 results of running five algorithms on three models (15,000 runs in all). The y-axis shows cost and the x-axis shows attainment. The size of each model is measured in number of mitigations. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also better solutions exhibit less variance, i.e. are clumped tighter together.

	Model 2 (31 mitigations)	Model 4 (58 mitigations)	Model 5 (99 mitigations)
SA	0.577	1.258	0.854
MaxFunWalk	0.122	0.429	0.398
ASTAR	0.003	0.017	0.048
KEYS	0.011	0.053	0.115
KEYS2	0.006	0.018	0.038

Fig. 11. Runtimes in seconds, averaged over 100 runs, measured using the “real time” value from the Unix `times` command. The size of each model is measured in number of mitigations (and for more details on model size, see Figure 4).

The results are pictured in Figure 10. Attainment is along the x-axis and cost (in thousands) is along the y-axis. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also, better solutions exhibit less variance; that is, the results are clumped closely together.

These graphs give a clear picture of the results obtained by our various algorithms. Two methods are clearly inferior:

- Simulated annealing exhibits the worst variance, lowest attainments, and highest costs.
- MaxFunWalk is better than SA (less variance, lower costs, higher attainment) but its variance is still far too high to use in any critical situation.

As to the others:

- On larger models such as model4 and model5, KEYS and KEYS2 exhibit lower variance, lower costs, and higher attainments than ASTAR.
- On smaller models such as model2, ASTAR usually produces higher attainments and lower variance than the KEYS algorithms (this advantage disappears on the larger models). However, observe the results near the (0, 0) point of model2’s ASTAR results: sometimes ASTAR’s heuristic search failed completely for that model

6.2 Runtime Analysis

Measured in terms of attainment and cost, there is little difference between KEYS and KEYS2. However, as shown by Figure 11, KEYS2 runs twice to three times as fast as its predecessor. Interestingly, Figure 11 ranks two of the algorithms in a similar order to Figure 10:

- Simulated annealing is clearly the slowest;
- MaxFunWalk is somewhat better but not as fast as the other algorithms.

As to ASTAR versus KEYS or KEYS2:

- ASTAR is faster than KEYS;
- and KEYS2 runs in time comparable to ASTAR.

Measured purely in terms of runtimes, there is little to recommend KEYS2 over ASTAR. However, ASTAR’s heuristic guesses were sometimes observed to be sub-optimal (recall the above discussion on the (0, 0) results in model2’s ASTAR results). Such sub-optimality was never observed for KEYS2.

model	expansion	model size	Runtime (secs)		$\frac{\text{KEYS}}{\text{KEYS2}}$
			KEYS	KEYS2	
2	1	62	0.01	0.01	1.07
2	2	124	0.03	0.02	1.23
4	1	139	0.04	0.02	2.29
5	1	201	0.13	0.04	3.18
2	4	248	0.10	0.05	2.09
4	2	278	0.17	0.05	3.48
5	2	402	0.50	0.12	4.26
2	8	496	0.44	0.14	3.21
4	4	556	0.73	0.16	4.66
5	4	804	1.98	0.38	5.21
4	8	1112	2.97	0.52	5.71
5	8	1608	8.06	1.35	5.96

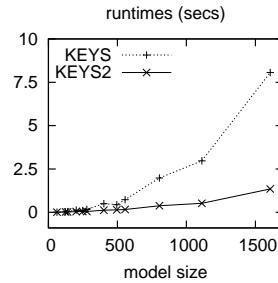


Fig. 12. Runtimes KEYS vs KEYS2 (medians over 1000 repeats) as models increase in size. The “model” number in column one corresponds to Figure 4. The “expansion factor” of column two shows how much the instance generator expanded the model. The “model sizes” of column three are the sum of mitigations, requirements, and risks seen in the expanded model.

model	expansion	model size	Calls to model		$\frac{\text{KEYS}}{\text{KEYS2}}$
			KEYS	KEYS2	
2	1	62	3100	800	3.9
2	2	124	6200	1100	5.6
4	1	139	5800	1100	5.3
5	1	201	9900	1400	7.1
2	4	248	12400	1600	7.8
4	2	278	11600	1500	7.7
5	2	402	19800	2000	9.9
2	8	496	24800	2200	11.3
4	4	556	23200	2200	10.5
5	4	804	39600	2800	14.1
4	8	1112	46400	3000	15.5
5	8	1608	79200	4000	19.8

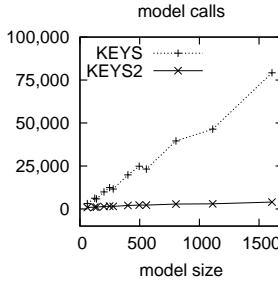


Fig. 13. Number of model calls made by KEYS vs KEYS2 (medians over 1000 results) as models increase in size. This figure uses the same column structure as Figure 12.

6.3 Scale-Up Studies

Figure 12 and Figure 13 shows the effect of changing the size of the model on the number of times that the model is asked to generate a score for both KEYS and KEYS2. To generate these graph, an *instance generator* was created that:

- Examined the real-world DDP models of Figure 4;
- Extracted statistics related to the different types of nodes (mitigations or risks or requirements) and the number of edges between different types of nodes;
- Used those statistics to build random models that were 1,2,4 and 8 times larger than the original models.

Figure 14 shows the results of curve fitting to the plots of Figure 12 and Figure 13. The KEYS and KEYS2 performance curves fit a low-order polynomial (of degree two) with very high coefficients of determination ($R^2 \geq 0.98$).

Figure 14 suggests that we could scale *either* KEYS or KEYS2 to larger models. However we still recommend KEYS2. The column marked $\frac{\text{KEYS}}{\text{KEYS2}}$ in Figure 13 shows the ratio of the number of calls made by KEYS vs KEYS2. As models get larger, the

	KEYS		KEYS2	
	runtimes	model calls	runtimes	model calls
exponential	0.82	0.83	0.88	0.93
polynomial (of degree 2)	0.99	0.99	0.99	0.98

Fig. 14. Coefficients of determination R^2 of KEYS/KEYS2 performance figures, fitted to two different functions: exponential or polynomial of degree two. Higher values indicate a better curve fit. In all cases, the best fit is not exponential.

number of calls to the model are an order of magnitude greater in KEYS than in KEYS2. If applied to models with slower runtimes than DDP, then this order of magnitude is highly undesirable.

6.4 Decision Ordering Algorithms

The decision ordering diagrams of Figure 15 show the effects of the decisions made by KEYS and KEYS2. For both algorithms, at $x = 0$, all of the mitigations in the model are set at random. During each subsequent era, more mitigations are fixed (KEYS sets one at a time, KEYS2 an incrementally increasing number). The lines in each of these plots show the median and spread seen in the 100 calls to the `model` function during each round.

Note that the these diagrams are *tame* and *well-behaved*:

- *Tame*: The “spread” values quickly shrink to a small fraction of the median.
- *Well-behaved*: The median values move smoothly to a plateau of best performance (high attainment, low costs).

On termination (at maximum value of x), KEYS and KEYS2 arrive at nearly identical median results (caveat: for `model2`, KEYS2 attains slightly more requirements at a slightly higher cost than KEYS). The spread plots for both algorithms are almost indistinguishable (exception: in `model2`, the KEYS2 spread is less than KEYS). That is, KEYS2 achieves the same results as KEYS, but (as shown in Figure 11 and Figure 12) it does so in less time.

A core assumption of this work is the “keys” concept; i.e. a small number of variables set the remaining model variables. Figure 15 offers significant support for this assumption: observe how most of the improvement in costs and attainments were achieved after KEYS and KEYS2 made only a handful of decisions (often ten or fewer).

On another matter, it is insightful to reflect on the effectiveness of different algorithms for generating decision ordering diagrams. KEYS2 is the most direct and fastest method. As mentioned above, all of the required information can be collected during one execution. On the other hand, simulated annealing, ASTAR, and MaxWalkSat would require a post-processor to generate the diagrams:

- Given D possible decisions, At each era, KEYS and KEYS2 collects statistics on partial solutions where $1, 2, 3, \dots |d|$ variables are fixed (where d is the set of decisions) while the remaining $D - d$ decisions are made at random.

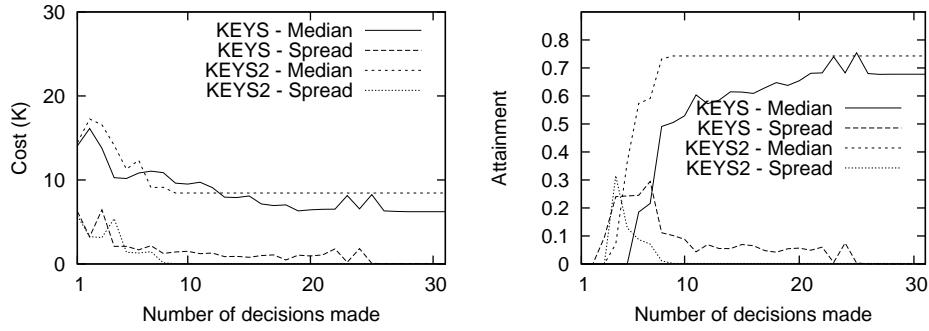


Figure 15a: Internal Decisions on Model 2.

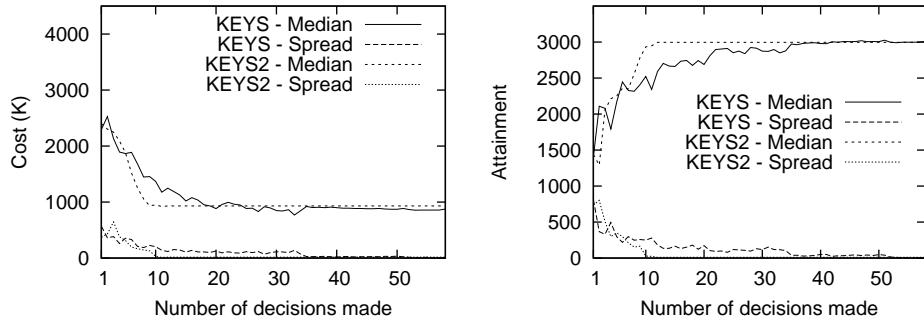


Figure 15b: Internal Decisions on Model 4.

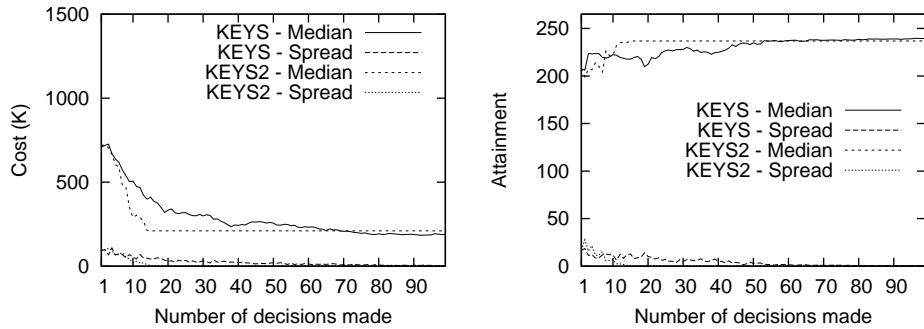


Figure 15c: Internal Decisions on Model 5.

Fig. 15. Median and spread of partial solutions learned by KEYS and KEYS2. X-axis shows the number of decisions made. “Median” shows the 50-th percentile of the measured value seen in 100 runs at each era. “Spread” shows the difference between the 75th and 50th percentile.

- ASTAR, Simulated Annealing, and MaxFunWalk work with full solutions since at each step they offer settings to all $d_i \in D$ variables. In the current form, they cannot comment on partial solutions. Modified forms of these algorithms could add in extra instrumentation and extra post-processing to comment on partial solutions using methods like feature subset selection [28] or a sensitivity analysis [61].

7 Related Work

7.1 Early vs Later Life-cycle Requirements Engineering

The case studies presented in this paper come from the NASA Jet Propulsion Lab’s Team X meetings. Team X conducts early life-cycle requirement discussions.

Once a system is running, released, and being maintained or extended, another problem is *release planning*; i.e. what features to add to the next N releases. To solve this problem, an inference engine must reason about how functionality extensions to current software can best satisfy outstanding stakeholder requirements. The challenge of release planning is that the benefits of added functionality must be weighed against the cost of implementing those extensions.

Several approaches have been applied to this problem including:

- The OPTIMIZE tool of Ngo-The and Ruhe [56], which combines linear programming with genetic programming to optimize release plans for software projects
- The weighted Pareto optimal genetic algorithm approach of Zhang et al. [70]

(See also the earlier comparison of exact vs greedy algorithms by Bagnall et al. [5]).

Without further experimentation, we cannot assert that KEYS2 will work as well on later life-cycle models (such as those used in release planning) as it did above (on the earlier life-cycle Team X models). However, at this time, we can see no reason why KEYS2 would not work as a non-linear optimizer of these later life-cycle models. This could be a productive area for future work.

7.2 Other Optimizers

As documented by the search-based SE literature [13, 31, 32, 58] and Gu et al [27], there are many possible optimization methods. For example:

- Gradient descent methods assume that an objective function $F(X)$ is differentiable at any single point N . A Taylor-series approximation of $F(X)$ can be shown to decrease fastest if the negative gradient ($-\Delta F(N)$) is followed from point N .
- Sequential methods run on one CPU while parallel methods spread the work over a distributed CPU farm.
- Discrete methods assume model variables have a finite range (that may be quite small) while continuous methods assume numeric values with a very large (possibly infinite) range.
- The search-based SE literature prefers meta-heuristic methods like simulated annealing, genetic algorithms and tabu search.

- Some methods map discrete values true/false into a continuous range 1/0 and then use integer programming methods like CPLEX [53] to achieve results.
- Other methods find overlaps in goal expressions and generate a binary decision diagram (BDD) where parent nodes store the overlap of children nodes.

This list is hardly exhaustive: Gu et al. list hundreds of other methods and no single paper can experiment with them all. All the algorithms studied here are discrete and sequential. We are currently exploring parallel versions of our optimizers but, so far, the communication overhead outweighs the benefits of parallelism.

As to the general class of gradient descent methods, we do not use them since they assume the objective function being optimizing is essentially continuous. Any model with an “if” statement in it is not continuous since, at the “if” point, the program’s behavior may become discontinuous. The requirements models studied here are discontinuous about every subset of every possible mitigation.

As to the more specific class of integer programming methods, we do not explore them here for two reasons. Coarfa et al. [14] found that integer programming-based approaches ran an order of magnitude slower than discrete methods like the MaxWalkSat and KEYS2 algorithms that we use. Similar results have been reported by Gu et.al where discrete methods ran one hundred times faster than integer programming [27].

Harman offers another reason to avoid integer programming methods. In his search-based SE manifest, Harman [31] argues that many SE problems are over-constrained and so there may exist no precise solution that covers all constraints. A complete solution over all variables is hence impossible and partial solution based on heuristic search methods are preferred. Such methods may not be complete; however, as Clarke et al remark, “...software engineers face problems which consist, not in finding *the* solution, but rather, in engineering an *acceptable* or *near-optimal solution* from a large number of alternatives.” [13]

7.3 Models of Requirements Engineering

DDP is a ultra-lightweight modeling tool. The value of ultra-lightweight ontologies in early life cycle modeling is widely recognized. For example Mylopoulos’ *soft-goal* graphs [54, 55] represent knowledge about non-functional requirements. Primitives in soft goal modeling include statements of partial influence such as *helps* and *hurts*. Another commonly used framework in the design rationale community is a “questions-options-criteria” (QOC) graph [64]. In QOC graphs:

- *Questions* suggest *options*. Deciding on one option can raise other questions;
- Options shown in a box denote *selected options*;
- Options are assessed by *criteria*;
- Criteria are gradual knowledge; i.e. they *tend/reject to support* options.

QOCs can succinctly summarize lengthy debates; e.g. the 480 sentences uttered in a debate on interface options can be displayed in a QOC graph on a single page [45]. Saaty’s Analytic Hierarchy Process (AHP) [60] is a variant of QOC.

While DDP shares many of the design aspects of softgoals & QOC & AHP, it differs in its representations and inference method. As explained above around Equation 1,

whereas AHP and QOC and softgoals propagate influences over hierarchies, DDP propagate influences over matrices.

7.4 Formal Models of Requirements Engineering

Zave & Jackson [69] define requirements engineering as finding the specification S for the domain assumptions K that satisfies the given requirements R ; i.e.

$$\text{find } S \text{ such that } S \vdash R \quad (7)$$

Jureta, Mylopoulos & Faulkner [41] (hereafter JMF) take issue with Equation 7, saying that it implicitly assume that K, S, R are precise and complete enough for the satisfaction relation to hold. More specifically, JMF complain that Equation 7 does not permit partial fulfillment of (some) non-functional requirements. Also, the Zave&Jackson definition does not allow any preference ordering of specification_1 over specification_2 . JMF offer a replacement ontology where classical inference is replaced with operators that supports the generation and ranking of subsets of domain assumptions that lead to maximal (w.r.t. size) subsets of the possible goals, and softgoal quality criteria⁵.

DDP reinterprets “ \vdash ” in Equation 7 as an inference across numeric quantities, rather than the inference over discrete logical variables suggested by Zave&Jackson. Hence, it can achieve the same goals as JMF (ranking of partial solutions with weighted goals) without requiring the JMF ontology.

7.5 Requirements Analysis Tools

There exist many powerful requirements analysis tools including continuous simulation (also called system dynamics) [1, 65], state-based simulation (including petri net and data flow approaches) [3, 29, 47], hybrid-simulation (combining discrete event simulation and systems dynamics) [19, 46, 63], logic-based and qualitative-based methods [7, chapter 20] [37], and rule-based simulations [52]. One can find these models being used in the requirements phase (i.e. the DDP tool described below), design refactoring using patterns [25], software integration [18], model-based security [40], and performance assessment [6]. Many researchers have proposed support environments to help explore the increasingly complex models that engineers are developing. Gray et al [26] have developed the Constraint-Specification Aspect Weaver (C-Saw), which uses aspect-oriented approaches [24] to help engineers in the process of model transformation. Cai and Sullivan [9] describe a formal method and tool called *Simon* that “supports interactive construction of formal models, derives and displays design structure matrices... and supports simple design impact analysis.” Other tools of note are lightweight formal methods such as ALLOY [38] and SCR [34] as well as UML tools that allow for the execution of life cycle specifications (e.g. CADENA [10]).

⁵ According to JMF: “a salient characteristic of softgoals is that they cannot be satisfied to the ideal extent, not only because of subjectivity, but also because the ideal level of satisfaction is beyond the resources available to (and including) the system. It is therefore said that a softgoal is not satisfied, but satisfied.”

Many of the above tools were built to maximize the expressive power of the representation language or the constraint language used to express invariants. What distinguishes our work is that we are *willing to trade off representational or constraint expressiveness for faster runtimes*. There exists a class of ultra-lightweight model languages which, as we show above, can be processed very quickly. Any of the tools listed in the last paragraph are also candidate solutions to the problem explored in this paper, if it can be shown that their processing can generate tame and well-behaved decision ordering diagrams in a timely manner.

7.6 Other Work on “Keys”

Elsewhere [50], we have documented dozens of papers that have reported the keys effect (that a small number of variables set the rest) under different names including *narrow*s, *master-variables*, *back doors*, and *feature subset selection*:

- Amarel [4] observed that search problems contain narrow sets of variables or collars that must be used in any solution. In such a search space, what matters is not so much how you get to these collars, but what decision you make when you get there. Amarel defined macros encoding paths between *narrow*s, effectively permitting a search engine to jump between them.
- In a similar theoretical analysis, Menzies & Singh [50] computed the odds of a system selecting solutions to goals using complex, or simpler, sets of preconditions. In their simulations, they found that a system will naturally select for tiny sets of preconditions (a.k.a. the keys) at a very high probability.
- Numerous researchers have examined *feature subset selection*; i.e. what happens when a data miner deliberately ignores some of the variables in the training data. For example, Kohavi and John [44] showed in numerous datasets that as few as 20% of the variables are *key* - the remaining 80% of variables can be ignored without degrading a learner’s classification accuracy.
- Williams et.al. [68] discuss how to use keys (which they call “back doors”) to optimize search. Constraining these back doors also constrains the rest of the program. So, to quickly search a program, they suggest imposing some set values on the key variables. They showed that setting the keys can reduce the solution time of certain hard problems from exponential to polytime, provided that the keys can be cheaply located, an issue on which Williams et.al. are curiously silent.
- Crawford and Baker [16] compared the performance of a complete TABLEAU prover to a very simple randomized search engine called ISAMP. Both algorithms assign a value to one variable, then infer some consequence of that assignment with forward checking. If contradictions are detected, TABLEAU backtracks while ISAMP simply starts over and re-assigns other variables randomly. Incredibly, ISAMP took *less* time than TABLEAU to find *more* solutions using just a small number of tries. Crawford and Baker hypothesized that a small set of *master variables* set the rest and that solutions are not uniformly distributed throughout the search space. TABLEAU’s depth-first search sometimes drove the algorithm into regions containing no solutions. On the other hand, ISAMP’s randomized sampling effectively searches in a smaller space.

In summary, the core assumption of our algorithms are supported in many domains.

8 Conclusion

Requirements tools such as the DDP tool (used at NASA for early lifecycle discussions), contain a shared group memory that stores all of the requirements, risks, and mitigations of each member of the group. Software tools can explore this shared memory to find consequences and interactions that may have been overlooked.

Studying that group memory is a non-linear optimization task: possible benefits must be traded off against the increased cost of applying various mitigations. Harman [31] cautions that solutions to non-linear problems may be “brittle” - small changes to the search results may dramatically alter the effectiveness of the solution. Hence, when reporting an analysis of this shared group memory, it is vitally important to comment on the robustness of the solution.

Decision ordering diagrams are a solution robustness assessment method. The diagrams rank all of the possible decisions from most-to-least influential. Each point x on the diagrams shows the effects on imposing the conjunction of decisions $1 \leq j \leq x$. These diagrams can comment on the robustness and neighborhood of solution $\{d_1..d_x\}$ using two operators:

1. By considering the variance of the performance statistics after applying $\{d_1..d_x\}$.
2. By comparing the results of using the first x decisions to that of using the first $x - 1$ or $x + 1$ actions.

Since the diagrams are sorted, this analysis of robustness and neighborhood takes, at most, time linear of the number of decisions. That is, theoretically, it takes linear time to *use* a decision ordering diagram (see §4.3).

Empirically, it take low-order polynomial time to *generate* a decision ordering diagram using KEYS2. This algorithm makes the “key” assumption (that a small group of variables set everything else) and uses Bayesian ranking mechanism to quickly find those keys. As discussed above in the *Related Work* section, this assumption holds over a wide range of models used in a wide range of domains. This “keys” assumption can be remarkably effective: our empirical results show that KEY2 can generate decision ordering diagrams faster than the other algorithms studied here. Better yet, curve fits to our empirical results show that KEYS runs in low-order polynomial time (of degree two) and so should scale to very large models.

Prior to this work, our two pre-experimental concerns were that:

- We would need to trade solution robustness against solution quality. More robust solutions may not have the highest quality.
- Demonstrating solution robustness requires multiple calls to an analysis procedure.

At least for the models studied here, neither concern was realized. KEYS2 generated the highest quality solutions (lowest cost, highest attainments) and did so more quickly than the other methods.

In §4.3 it was argued that decision ordering diagrams are useful when they are *timely* to generate while being *well-behaved* and *tame*. KEYS2’s results are the most *timely*

(fastest to generate) of all of the methods studied here. As to the other criteria, Figure 15 shows that KEYS2’s decision ordering diagrams:

- Move smoothly to a plateau with only a small amount of “jitter”;
- Have very low spreads, compared to the median results.

That is, at least for the models explored here, KEYS2 generated decision ordering diagrams they are both *well-behaved* and *tame*.

In summary, we recommend KEYS2 for generating decision ordering diagrams since, apart from the (slightly slower) KEYS algorithm, we are unaware of other search-based software engineering methods that enable such a rapid reflection of solution robustness.

9 Acknowledgments

This research was conducted at West Virginia University, the Jet Propulsion Laboratory under a contract with the National Aeronautics and Space administration, Alderson-Broaddus College, and Miami University. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government

References

1. T. Abdel-Hamid and S. Madnick. *Software Project Dynamics: An Integrated Approach*. Prentice-Hall Software Series, 1991.
2. F. A. Administration. System engineering manual version 3.1, section 4.6: Trade studies, 2006. Available from http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/operations/sysengsaf/seman/SEM3.1/Section%204.6.pdf.
3. M. Akhavi and W. Wilson. Dynamic simulation of software process models. In *Proceedings of the 5th Software Engineering Process Group National Meeting (Held at Costa Mesa, California, April 26 - 29)*. Software engineering Institute. Carnegie Mellon University, 1993.
4. S. Amarel. Program synthesis as a theory formation task: Problem representations and solution methods. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach: Volume II*, pages 499–569. Kaufmann, Los Altos, CA, 1986.
5. A. Bagnall, V. Rayward-Smith, and I. Whittle. The next release problem. *Information and Software Technology*, 43(14), December 2001.
6. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5), May 2004.
7. I. Bratko. *Prolog Programming for Artificial Intelligence. (third edition)*. Addison-Wesley, 2001.
8. T. Bylander, D. Allemand, M. Tanner, and J. Josephson. The Computational Complexity of Abduction. *Artificial Intelligence*, 49:25–60, 1991.
9. Y. Cai and K. J. Sullivan. Simon: modeling and analysis of design space structures. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 329–332, New York, NY, USA, 2005. ACM Press.

10. A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. Calm and cadena: Meta-modeling for component-based product-line development. *IEEE Computer*, 39(2), Februry 2006. Available from <http://projects.cis.ksu.edu/docman/view.php?7/129/CALM-Cadena-IEEE-Computer-Feb-2006.pdf>.
11. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification*, 2002.
12. R. Clark. Faster treatment learning, Computer Science, Portland State University. Master's thesis, 2005.
13. J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Man-coridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings-Software*, 150(3):161–175, 2003.
14. C. Coarfa, D. D. Demopoulos, A. San, M. Aguirre, D. Subramanian, and M. Y. Vardi. Random 3-sat: The plot thickens. In *In Principles and Practice of Constraint Programming*, pages 143–159, 2000. Availabe from <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3662>.
15. S. Cornford, M. Feather, and K. Hicks. DDP a tool for life-cycle risk management. In *IEEE Aerospace Conference, Big Sky, Montana*, pages 441–451, March 2001.
16. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.
17. J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.
18. P. Denno, M. P. Steves, D. Libes, and E. J. Barkmeyer. Model-drven integration using existing models. *IEEE Software*, 20(5):59–63, Sept.-Oct. 2003.
19. P. Donzelli and G. Iazeolla. Hybrid simulation modelling of the software process. *Journal of Systems and Software*, 59(3), December 2001.
20. M. Feather, S. Cornford, K. Hicks, J. Kiper, and T. Menzies. Application of a broad-spectrum quantitative requirements model to early-lifecycle decision making. *IEEE Software*, 2008. Available from <http://menzies.us/pdf/08ddp.pdf>.
21. M. Feather, K. Hicks, R. Mackey, and S. Uckun. Guiding technology deployment decisions using a quantitative requirements analysis technique. In *IEEE International Conference on Requirements Engineering, Industrial Practice and Experience track Barcelona, Spain*, 2008.
22. M. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002. Available from <http://menzies.us/pdf/02re02.pdf>.
23. M. Feather, S. Uckun, and K. Hicks. Technology maturation of integrated system health management. In *Space Technology and Applications International Forum (STAIF-2008) Albuquerque, USA*, February 2008.
24. R. E. Filman. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.
25. R. France, S. Ghosh, E. Song, and D. Kim. A metamodeling approach to pattern-based moel refractoringt. *IEEE Software*, 20(5):52–58, Sept.-Oct. 2003.
26. J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering. *IEEE Computer*, 39(2):51–58, February 2006.
27. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.
28. M. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions On Knowledge And Data Engineering*, 15(6):1437– 1447, 2003. Available from <http://www.cs.waikato.ac.nz/~mhall/HallHolmesTKDE.pdf>.

29. D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
30. M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering, ICSE'07*. 2007.
31. M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
32. M. Harman and J. Wegener. Getting results from search-based approaches to software engineering. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 728–729, Washington, DC, USA, 2004. IEEE Computer Society.
33. P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
34. C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, January 2002. Available from <http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heimtmeier-encse.p%df>.
35. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
36. Y. Hui, E. Prakash, and N. Chaudhari. Game ai: artificial intelligence for 3d path finding. In *TENCON 2004. 2004 IEEE Region 10 Conference*, volume 2, pages 306–309, 2004.
37. Y. Iwasaki. Qualitative physics. In P. C. A. Barr and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, pages 323–413. Addison Wesley, 1989.
38. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
39. O. Jalali, T. Menzies, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings of the PROMISE 2008 Workshop (ICSE)*, 2008. Available from <http://menzies.us/pdf/08keys.pdf>.
40. J. Jerjens and J. Fox. Tools for model-based security engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 819–822, New York, NY, USA, 2006. ACM Press.
41. I. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the core ontology and problem in requirements engineering. In *International Requirements Engineering, 2008. RE '08. 16th IEEE*, pages 71–80, Sept. 2008.
42. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, Aug. 4–8 1996. AAAI Press / MIT Press. Available from <http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps>.
43. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
44. R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.
45. A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options and criteria: Elements of design space analysis. In T. Moran and J. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 53–106. Lawerence Erlbaum Associates, 1996.
46. R. Martin and R. D. M. Application of a hybrid process simulation model to a software development project. *Journal of Systems and Software*, 59(3), 2001.
47. R. Martin and D. M. Raffo. A model of the software development process using both continuous and discrete models. *International Journal of Software Process Improvement and Practice*, June/July 2000.
48. T. Menzies, O. Jalali, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings PROMISE '08 (ICSE)*, 2008.

49. T. Menzies, J. Kiper, and M. Feather. Improved software engineering decision support through automatic argument reduction tools. In *SEDECS'2003: the 2nd International Workshop on Software Engineering Decision Support (part of SEKE2003)*, June 2003. Available from <http://menzies.us/pdf/03star1.pdf>.
50. T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In M. Madravio, editor, *Soft Computing in Software Engineering*. Springer-Verlag, 2003. Available from <http://menzies.us/pdf/03maybe.pdf>.
51. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, 1953.
52. P. Mi and W. Scacchi. A knowledge-based environment for modeling and simulation software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, pages 283–294, September 1990.
53. H. Mittelmann. Recent benchmarks of optimization software. In *22nd Euorpean Conference on Operational Research*, 2007.
54. J. Mylopoulos, L. Cheng, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, January 1999.
55. J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions of Software Engineering*, 18(6):483–497, June 1992.
56. A. Ngo-The and G. Ruhe. Optimized resource allocation for software release planning. *Software Engineering, IEEE Transactions on*, 35(1):109–123, Jan.-Feb. 2009.
57. J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
58. L. Rela. Evolutionary computing in search-based software engineering. Master’s thesis, Lappeenranta University of Technology, 2004.
59. S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.
60. T. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill, 1980.
61. A. Saltelli, K. Chan, and E. Scott. *Sensitivity Analysis*. Wiley, 2000.
62. B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In M. Trick and D. S. Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.
63. S. Setamanit, W. Wakeland, and D.Raffo. Using simulation to evaluate global software development task allocation strategies. *Software Process: Improvement and Practice*, (Forthcoming), 2007.
64. S. B. Shum and N. Hammond. Argumentation-based design rationale: What use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, 1994.
65. H. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill, 2000.
66. B. Stout. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, (7), 1997.
67. T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *In Proc. of the 1st International Conference on Constraints in Computational Logics*, pages 34–49. Springer-Verlag, 1994.
68. R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI 2003*, 2003. <http://www.cs.cornell.edu/gomes/FILES/backdoors.pdf>.
69. P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.
70. H. Zhang and X. Zhang. Comments on ‘data mining static code attributes to learn defect predictors’. *IEEE Transactions on Software Engineering*, September 2007.

Obtaining our System

We have placed on-line all the materials required for other researchers to conduct further investigation into this problem. All the code, Makefiles, scripts, and so on used in this paper are available at <http://unbox.org/wisp/tags/ddpExperiment/install>. For security reasons, all the available JPL requirements models have been “sanitized”; i.e. all words replaced with anonymous variables.

Sharing Experiments Using Open Source Software



Adam Nelson, Tim Menzies, Gregory Gay*,*

Lane Department of Computer Science and Electrical Engineering, West Virginia University,
Morgantown, WV, USA

KEY WORDS: open source, data mining

SUMMARY

When researchers want to repeat, improve or refute prior conclusions, it is useful to have a complete and operational description of prior experiments. If those descriptions are overly long or complex, then sharing their details may not be informative.

OURMINE is a scripting environment for the development and deployment of data mining experiments. Using OURMINE, data mining novices can specify and execute intricate experiments, while researchers can publish their complete experimental rig alongside their conclusions. This is achievable because of OURMINE's succinctness. For example, this paper presents two experiments documented in the OURMINE syntax. Thus, the brevity and simplicity of OURMINE recommends it as a better tool for documenting, executing, and sharing data mining experiments.

1. Introduction

Since 2004, the authors have been involved in an open source data mining experiment. Researchers submitting to the PROMISE conference on predictive models in software engineering are encouraged to offer not just conclusions, but also the data they used to generate those conclusions. The result is nearly 100 data sets, all on-line, freely available for download †.

The premise of this paper is that after *data sharing* comes *experiment sharing*; i.e. repositories should grow to include not just data, but the code required to run experiments over that data. This simplifies the goal of letting other researchers repeat, or even extend, data mining experiments of others. Repeating experiments is not only essential in confirming previous results, but also in providing an insightful understanding of the fundamentals

*Correspondence to: Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA

*E-mail: rabbituckman@gmail.com, tim@menzies.us, gregoryg@csee.wvu.edu

†<http://promisedata.org/data>

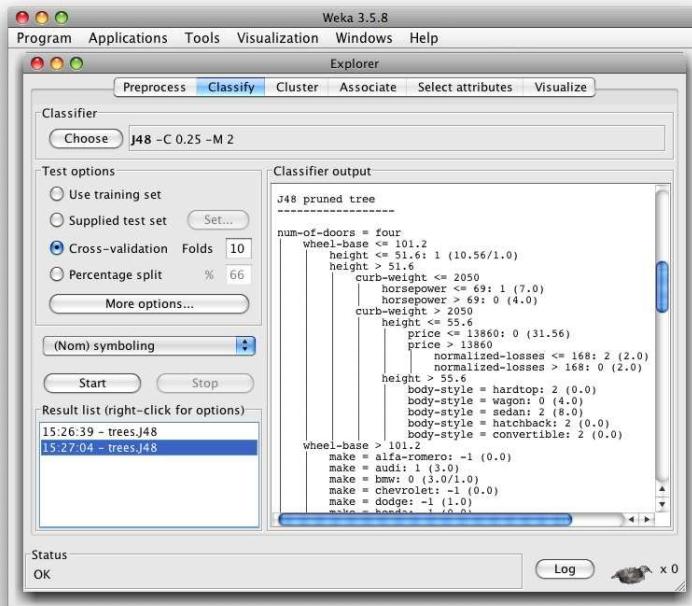


Figure 1. The WEKA toolkit running the J48 decision tree learner.

involving those experiments. Thus, it is important to have a prepared set of clearly defined and operational experiments that can be utilized in order to achieve similar prior results.

Consequently, we have been assessing data mining toolkits such as the WEKA[‡] tool of Figure 1, “R”[§], the ORANGE[¶] tool of Figure 2, RAPID-I^{||}, MLC++ **, and MATLAB^{††}. We find that they are not suitable for publishing experiments. Our complaint with these tools is same as that offered by Ritthoff et al. [21]:

- Proprietary tools such as MATLAB are not freely available.
- Real-world experiments are more complex than running one algorithm.

[‡]<http://www.cs.waikato.ac.nz/ml/weka/>

[§]<http://www.r-project.org/>

[¶]<http://magix.fri.uni-lj.si/orange/>

^{||}<http://rapid-i.com/>

^{**}<http://www.sgi.com/tech/mlc/>

^{††}<http://www.mathworks.com>

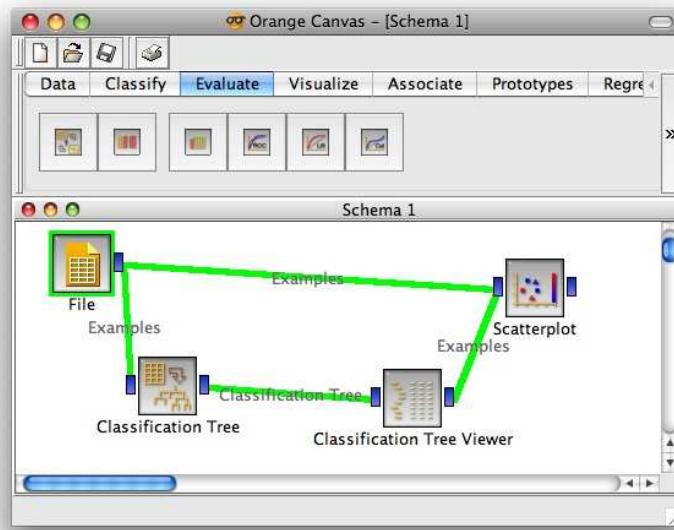


Figure 2. Orange's visual programming environment.

- Rather, such experiments or applications require *intricate combinations* of a large number of tools that include data miners, data pre-processors and report regenerators.

However, the need to “wire up” data miners within a multitude of other tools has been addressed in many ways. In WEKA’s visual *knowledge flow* programming environment, for example, nodes represent different pre-processors/ data miners/ etc while arcs represent how data flows between them. A similar visual environment is offered by many other tools including ORANGE (see Figure 2). The YALE tool (now RAPID-I) built by Ritthoff et al. formalizes these visual environments by generating them from XML schemas describing *operator trees* like Figure 3.

Ritthoff et al. argues (and we agree) that the standard interface of (say) WEKA does not support the rapid generation of these intricate combinations. In our experience these visual environments are either overly elaborate, discouraging or distracting:

- These standard data mining toolkits may be overly elaborate. As shown throughout this paper, very simple UNIX scripting tools suffice for the specification, execution, and communication of experiments.
- Some data mining novices are discouraged by the tool complexity. These novices shy away from extensive modification and experimentation.

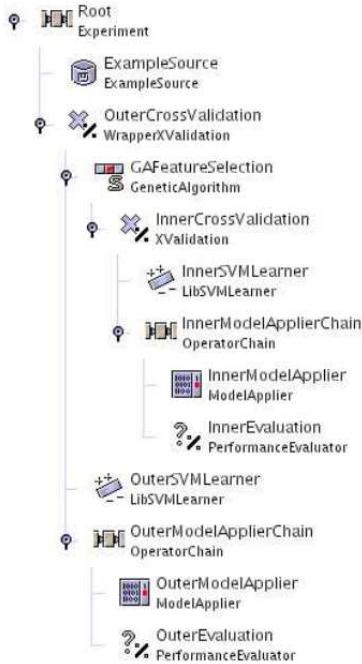


Figure 3. Rapid-I's operator trees. [17]. Internally, this tree is a nested XML expression that is traversed top-down to complete an experiment.

- Other developers may become so enamored with the impressive software engineering inside these tools that they waste much time building environments to support data mining, but never get around to the data mining itself.
- According to Menzies [12], while visual systems provide motivation for beginners, they fail in providing a good explanation device for many software engineering and knowledge engineering problems, as many of these problems are not inherently spatial. For example, suppose an entity-relationship diagram is drawn on a piece of paper. While inferences can be made from the diagram, they are not entirely dependent on the physical location of (e.g.) an entity.

A similar experience is reported by our colleagues in WVU Department of Statistics. In order to support teaching, they hide these data mining tools inside high-level scripting environments. Their scripting environments shield the user from the complexities of “R” by defining high-level LISP code that, internally, calls “R”. While a powerful language, we find that LISP can be arcane to many audiences.

OURMINE is a data mining scripting environment. The current kit has tools written in BASH/ GAWK/ JAVA/ PERL/ and there is no technical block to adding other tools written in other languages. Other toolkits impose strict limitations of the usable languages:

- MLC++ requires C++
- Extensions to WEKA must be written in JAVA.

Our preference for BASH [19]/GAWK [1] over, say, LISP is partially a matter of taste but we defend that selection as follows. Once a student learns, for example, RAPID-I's XML configuration tricks, then those learned skills are highly specific to that toolkit. On the other hand, once a student learns BASH/GAWK methods for data pre-processing and reporting, they can apply those scripting tricks to any number of future UNIX-based applications.

This paper introduces OURMINE as follows:

- First, we describe the base tool and offer some samples of coding in OURMINE;
- Next, we demonstrate OURMINE's ability to succinctly document even complex experiments.
 - OURMINE is a tool for both practitioners and researchers. Our first demonstration experiment is practitioner-focused and shows how OURMINE can be used in an industrial setting. Following experiments demonstrate OURMINE as an adequate tool for current research.

This paper is an extension of a prior publication [5]:

- This paper details functionality added since that last paper was published;
- This paper adds two new experiments to the description of OURMINE (see §3.1 and §3.3).

2. OURMINE

OURMINE was developed to help graduate students at West Virginia University document and execute their data mining experiments. The toolkit uses UNIX shell scripting. As a result, any tool that can be executed from a command prompt can be seamlessly combined with other tools.

For example, Figure 4 shows a simple bash function used in OURMINE to clean text data before conducting any experiments using it. Line 5 passes text from a file, performing tokenization, removing capitals and unimportant words found in a stop list, and then in the next line performing Porter's stemming algorithm on the result.

OURMINE allows connectivity between tools written in various languages as long as there is always a command-line API available for each tool. For example, the modules of Figure 4 are written using BASH, Awk and Perl.

The following sections describe OURMINE's functions and applications.

2.1. Built-in Data and Functions

In order to encourage more experimentation, the default OURMINE installation comes with numerous data sets:

```

1 clean(){
2     local docdir=$1
3     local out=$2
4
5     for file in $docdir/*; do
6         cat $file | tokes | caps | stops $Lists/stops.txt > tmp
7         stems tmp >> $out
8         rm tmp
9     done
9 }

```

Figure 4. An OURMINE function to clean text documents and collect the results. *Tokes* is a tokenizer; *caps* sends all words to lower case; *stops* removes the stop words listed in “\$Lists/stops.txt”; and *stems* performs Porter’s stemming algorithm (removes confusing suffixes).

- *Text mining data sets*: including STEP data sets (numeric): ap203, ap214, bbc, bbcsport, law, 20 Newsgroup subsets [sb-3-2, sb-8-2, ss-3-2, sl-8-2]^{††}
- *Discrete UCI Machine Learning Repository datasets*: anneal, colic, hepatitis, kr-vs-kp, mushroom, sick, waveform-5000, audiology, credit-a, glass, hypothyroid, labor, pcolic, sonar, vehicle, weather, autos, credit-g, heart-c, ionosphere, letter, primary-tumor, soybean, vote, weather.nominal,breast-cancer, diabetes, heart-h, iris, lymph, segment, splice, vowel;
- *Numeric UCI Machine Learning Repository datasets*: auto93, baskball, cholesterol, detroit, fruitfly, longley, pbc, quake, sleep, autoHorse, bodyfat, cleveland, echoMonths, gascons, lowbwt, pharynx, schlvote, strike, autoMpg, bolts, cloud, elusage, housing, mbagrade, pollution, sensory, veteran, autoPrice, breastTumor, cpu, fishcatch, hungarian, meta, pwLinear, servo, vineyard.
- The defect prediction data sets from the *PROMISE repository*: CM1, KC1, KC2, KC3, MC2, MW1, PC1

OURMINE also comes with a variety of built-in functions to perform data mining and text mining tasks. For a complete list, see the appendix.

2.2. Learning and Teaching with OURMINE

Data mining concepts become complex when implemented in a complex manner. For this reason, OURMINE utilizes simple scripting tools (written mostly in BASH or GAWK) to better convey the inner-workings of these concepts. For instance, Figure 5 shows a GAWK implementation used by OURMINE to determine the TF-IDF [20] (Term Frequency * Inverse Document Frequency, described below) values of each term in a document. This script is simple and concise, while a C++ or Java implementation would be large and overly complex. An

^{††}<http://mlg.ucd.ie/datasets>

```

function train() { #update counters for all words in the record
    Docs++;
    for(I=1;I<NF;I++) {
        if( ++In[$I,Docs]==1)
            Doc[$I]++;
        Word[$I]++;
        Words++ }
    }
function tfidf(i) { #compute tfidf for one word
    return Word[i]/Words*log(Docs/Doc[i])
}

```

Figure 5. A GAWK implementation of TF-IDF.

additional example demonstrating the brevity of OURMINE script can be seen in Figure 6, which is a complete experiment whose form can easily be taught and duplicated in future experiments.

Another reason to prefer scripting in OURMINE over the complexity of RAPID-I, WEKA, “R”, etc, is that it reveals the inherent simplicity of many of our data mining methods. For example, Figure 7 shows a GAWK implementation of a Naive Bayes classifier for discrete data where the last column stores the class symbol. This tiny script is no mere toy- it successfully executes on very large data sets such as those seen in the 2001 KDD cup and in [18]. WEKA cannot process these large data sets since it always loads its data into RAM. Figure 7, on the other hand, only requires memory enough to store one instance as well as the frequency counts in the hash table “*F*”.

More importantly, in terms of teaching, Figure 7 is easily customizable. Figure 8 shows four warm-up exercises for novice data miners that (a) introduce them to basic data mining concepts and (b) show them how easy it is to script their own data miner: Each of these tasks requires changes to less than 10 lines from Figure 7. The simplicity of these customizations fosters a spirit of “this is easy” for novice data miners. This in turn empowers them to design their own extensive and elaborate experiments.

Also from the teaching perspective, demonstrating on-the-fly a particular data mining concept helps not only to solidify this concept, but also gets the student accustomed to using OURMINE as a tool in a data mining course. As an example, if a Naive Bayes classifier is introduced as a topic in the class, an instructor can show the workings of the classifier by hand, and then immediately afterwards complement this by running Naive Bayes on a small data set in OURMINE. Also, since most of OURMINE does not use pre-compiled code, an instructor can make live changes to the scripts and quickly show the results.

We are not alone in favoring GAWK for teaching purposes. Ronald Loui uses GAWK to teaching artificial intelligence at Washington University in St. Louis. He writes:

```

1 demo004(){
2     local out=$Save/demo004-results.csv
3     [ -f $out ] && echo "Caution: File exists!" || demo004worker $out
4 }

5 # run learners and perform analysis
6 demo004worker(){

7     local learners="nb j48"
8     local data="$Data/discrete/iris.arff"
9     local bins=10
10    local runs=5
11    local out=$1

12    cd $Tmp
13    (echo "#data,run,bin,learner,goal,a,b,c,d,acc,pd,pf,prec,bal"
14    for((run=1;run<=$runs;run++)); do
15        for dat in $data; do

16            blab "data='basename $dat',run=$run"
17            for((bin=1;bin<=$bins;bin++)); do

18                rm -rf test.lisp test.arff train.lisp train.arff
19                makeTrainAndTest $dat $bin $bin
20                goals='cat $dat | getClasses --brief'

21                for learner in $learners; do

22                    $learner train.arff test.arff | gotwant > produced.dat
23                    for goal in $goals; do

24                        cat produced.dat |
25                        abcd --prefix "'basename $dat',$run,$bin,$learner,$goal" \
26                            --goal "$goal" \
27                            --decimals 1
28                        done
29                    done
30                done
31                blabln
32            done
33        done | sort -t, -r -n -k 11,11) | malign > $out

34    winLossTie --input $out --test w --fields 14 --key 4 --perform 11
35 }

```

Figure 6. A demo OURMINE experiment. This worker function begins by being called by the top level function *demo004* on lines 1-4. Noteworthy sections of the demo code are at: line 19, where training sets and test sets are built from 90% and 10% of the data respectively, lines 25-27 in which values such as *pd*, *pf* and *balance* are computed via the *abcd* function that computes values from the confusion matrix, and line 34 in which a *Wilcoxon* test is performed on each learner in the experiment using *pd* as the performance measure.

```

#naive bayes classifier in gawk
#usage: gawk -F, -f nbc.awk Pass=1 train.csv Pass=2 test.csv

Pass==1 {train()}
Pass==2 {print $NF "|" classify()}

function train(    i,h) {
    Total++;
    h=$NF;      # the hypothesis is in the last column
    H[h]++;
    # remember how often we have seen "h"
    for(i=1;i<=NF;i++) {
        if ($i=="?")
            continue;      # skip unknown values
        Freq[h,i,$i]++;
        if (++Seen[i,$i]==1)
            Attr[i]++; # remember unique values
    }
    function classify(      i,temp,what,like,h) {
        like = -100000;      # smaller than any log
        for(h in H) {        # for every hypothesis, do...
            temp=log(H[h]/Total); # logs stop numeric errors
            for(i=1;i<NF;i++) {
                if ( $i=="?")
                    continue;      # skip unknown values
                temp += log((Freq[h,i,$i]+1)/(H[h]+Attr[NF])) }
                if ( temp >= like ) { # better hypothesis
                    like = temp
                    what=h}
            }
            return what;
    }
}

```

Figure 7. A Naive Bayes classifier for a CSV file, where the class label is found in the last column.

There is no issue of user-interface. This forces the programmer to return to the question of what the program does, not how it looks. There is no time spent programming a binsort when the data can be shipped to /bin/sort in no time. [9]

Function documentation provides a way for newcomers to OURMINE to not only get to know the workings of each function, but also add to and modify the current documentation. Instead of asking the user to implement a more complicated “man page”, OURMINE uses a very simple system consisting of keywords such as *name*, *args* and *eg* to represent a function name, its arguments and an example of how to use it. Using this documentation is simple. Entering *funs* at the OURMINE prompt provides a sorted list of all available functions in ourmine. Then, by typing *help X*, where *X* is the name of the function, information about that function is printed to the screen. See Figure 9 for an example of viewing the help document for the function *j4810*. Documentation for a function is added by supplying a text file to the *helpdocs* directory in OURMINE named after the function.

1. Modify Figure 7 so that there is no train/test data. Instead, make it an incremental learning. Hint: 1) call the functions *train*, then *classify* on every line of input. 2) The order is important: always *train* before *classifying* so the the results are always on unseen data.
2. Convert Figure 7 into HYPERPIPES [3]. Hint: 1) add globals *Max[h, i]* and *Min[h, i]* to keep the max/min values seen in every column “*i*” and every hypothesis class “*h*”. 2) Test instance belongs to the class that most overlaps the attributes in the test instance. So, for all attributes in the test set, sum the returned values from *contains1*:

```
function contains1(h,i,val,numerip) {
    if(numerip)
        return Max[h,i] >= val && Min[h,i] <= val
    else return (h,i,value) in Seen
}
```
3. Use Figure 7 to implement an anomaly detector. Hint: 1) make all training examples get the same class; 2) an anomalous test instance has a likelihood $\frac{1}{\alpha}$ of the mean likelihood seen during training (*alpha* needs tuning but *alpha* = 50 is often useful).
4. Using your solution to #1, create an incremental version of HYPERPIPES and an anomaly detector.

Figure 8. Four Introductory OURMINE programming exercises.

```
Function: j4810
Arguments: <data (arff)>
Example(s): j4810 weather.arff
Description: Uses a j48 decision tree learner on the input data

Function Code:
=====
j4810 () {
    local learner=weka.classifiers.trees.J48
    $Weka $learner -C 0.25 -M 2 -i -t $1
}
```

Figure 9. Function help in OURMINE.

3. Using Ourmine for Industrial and Research Purposes

OURMINE is not just a simple demonstration system for novice data miners. It is also a tool for commercial practitioners and researchers alike. In order to demonstrate OURMINE’s utility for practitioners, our first demonstration experiment is an example of selecting the right learner and sample size for a particular domain.

- In part A of this first experiment, we show a very simple example of how practitioners can “tune” learners to data sets in order to achieve better results. This is shown through the comparison of learners (Naive Bayes, J48, One-r, etc.) training on raw data, and on discretized data.

-
- In part B of this experiment, we demonstrate how *early* our predictors can be applied. This is accomplished through incremental, random sampling of the data, to which the winning method from Part A is applied. The objective here is to report the smallest number of cases required in order to learn an adequate theory.

As software engineering researchers, we use OURMINE to generate publications in leading software engineering journals and conferences. For example, in the last three years, we have used OURMINE in this way to generate [2,5,13,15,16,22]. In order to demonstrate OURMINE's application to research, we present here two further experiments. Experiment #2 reproduces an important recent result by Turhan et al. [22]. Kitchenham et al. report that there is no clear consensus on the relative value of effort estimation models learned from either local or imported data [7]. Since this matter has not been previously studied in the defect prediction literature, Turhan et al. conducted a series of experiments with ten data sets from different companies. In their *within-company* (WC) experiments, defect predictors were learned from 90% of the local data, then tested on the remaining 10%. In the *cross-company* (CC) experiments, defect predictors were learned from nine companies then tested on the tenth. Turhan et al. found that the CC predictions were useless since they suffered from very high false alarm rates. However, after applying a simple *relevancy filter*, they found that imported CC learning yielded predictors that were nearly as good as the local WC learning. That is, using these results, we can say that organizations can make quality predictions about their source code, even when they lack local historical logs.

Experiment #3 checks a recent text mining result from an as-yet-unpublished WVU masters thesis. Matheny [10] benchmarked various lightweight learning methods (TF*IDF, the GENIC stochastic clusterer) against other, slower, more rigorous learning methods (PCA, K-means). As expected, the rigorous learning methods ran much slower than the stochastic methods. But, unexpectedly, Matheny found that the lightweight methods perform nearly as well as the rigorous methods. That is, using these results, we can say that text mining methods can scale to much larger data sets.

The rest of this section describes the use of OURMINE to reproduce Experiment #1, #2 and #3.

3.1. Experiment I: Commissioning a Learner

OURMINE's use is not only important to researchers, but also to practitioners. To demonstrate this, two very simple experiments were conducted using seven PROMISE data sets (CM1, KC1, KC2, KC3, MC2, MW1, PC1). The aim of this experiment is to commission a data mining system for a local site. When selecting the right data miners for a particular source of data, three issues are:

1. What learners to use?
2. What discretizers to use?
3. How much data is required for adequate learning?

Given software being released in several stages, OURMINE can be used on stage 1 data to find the right answers to the above questions. These answers can be applied on stages 2,3,4, etc.

In Part A of this experiment, four learners (Naive Bayes, J48, ADTree, One-R) are trained using undiscretized data, and then the same learners are trained on discretized data for a total of eight combinations. While this represents a highly simplified proof-of-concept, from it we can illustrate how our learners can be integrated with other techniques to find the best treatment for the data set(s) of concern. Practitioners can then short-circuit their experiments by only performing further analyses on methods deemed worthwhile for the current corpus. In turn, this decreases the amount of time in which results can be obtained.

In Part B of the experiment, the winning treatment from Part A is used with randomly selected, incremental sizes of training data. In this experiment, it can be shown how early our quality predictors can be applied based on the smallest number of training examples required to learn an adequate theory. Predictors were trained using $N = 100$, $N = 200$, $N = 300\dots$ randomly selected instances up to $N = 1000$. For each training set size N , 10 experiments were conducted using $|Train| = 90\% * N$, and $|Test| = 100$. Both *Train* and *Test* were selected at random for each experiment. Our random selection process is modeled after an experiment conducted by Menzies et. al. [14]. In that paper, it was found that performance changed little regardless of whether fewer instances were selected (e.g. 100), or much larger sizes on the order of several thousand. Additionally, it was determined that larger training sizes were actually a disadvantage, as variance increased with increasing training set sizes.

3.1.1. Results

Our results for Part A and B of this experiment use pd (probability of detection) values sorted in decreasing order, and pf (probability of false alarm) values sorted in increasing order. These values assess the classification performance of a binary detector, and are calculated as follows. If $\{a, b, c, d\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by the detector then $pd = recall = \frac{d}{b+d}$ and $pf = \frac{c}{a+c}$. Note that a higher pd is better, while a lower pf is better.

The last column of each figure shows quartile charts of the methods' pd and pf values. The black dot in the center of each plot represents the median value, and the line going from left to right from this dot show the second and third quartile respectively.

Column one of each figure gives a method its rank based on a Mann-Whitney test at 95% confidence. A rank is determined by how many times a learner or learner/filter wins compared to another. The method that wins the most number of times is given the highest rank.

Figure 10 and Figure 11 show the results from Part A of the experiment. As can be seen, ADTrees (Alternating Decision Trees) [4] trained on discretized data yields the best results:

- Mann-Whitney test ranking (95% confidence): ADTrees + discretization provides the highest rank for both PD and PF results
- Medians & Variance: ADTrees + discretization results in the highest median/lowest variance for PD, and the lowest variance for PF, with only a 1% increase over the lowest median

Results from Part B of the experiment are shown in Figure 12 and Figure 13. Using the winning method from Part A, or learning using ADTrees on discretized data, results show that unquestionably a training size of $N = 600$ instances is the best number to train our defect

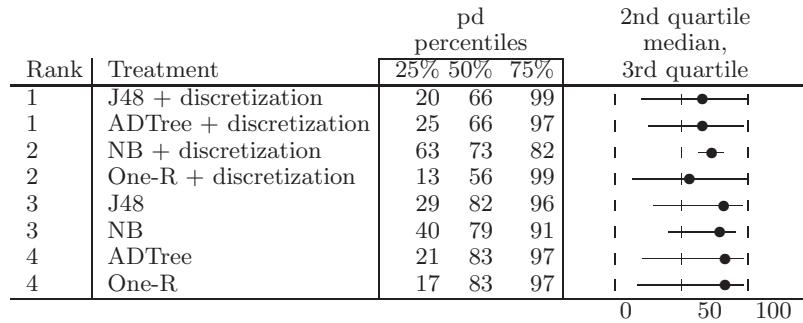


Figure 10. Experiment #1 - Part A - (Learner tuning). Probability of Detection (PD) results, sorted by rank then median values.

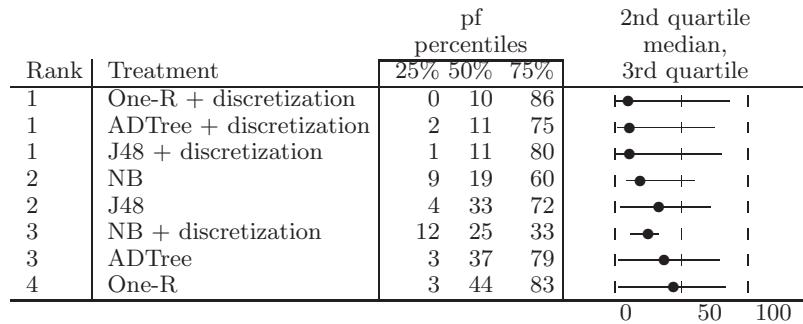


Figure 11. Experiment #1 - Part A - (Learner tuning). Probability of False Alarm (PF) results, sorted by rank then median values.

predictors using the aforementioned experiment setup. This is clear when considering $N = 600$ maintains the highest Mann-Whitney ranking (again using 95% percent confidence), as well as the highest PD and lowest PF medians.

More importantly, perhaps, it should be noted that while the most beneficial number of training instances remains 600, we can still consider a significantly smaller value of, say, $N = 300$ without much loss in performance; a training size of just 300 yields a loss in PD ranking of only one, with a 3% decrease in median, while PF ranking is identical to our winning size and sporting medians of a mere 3% increase.

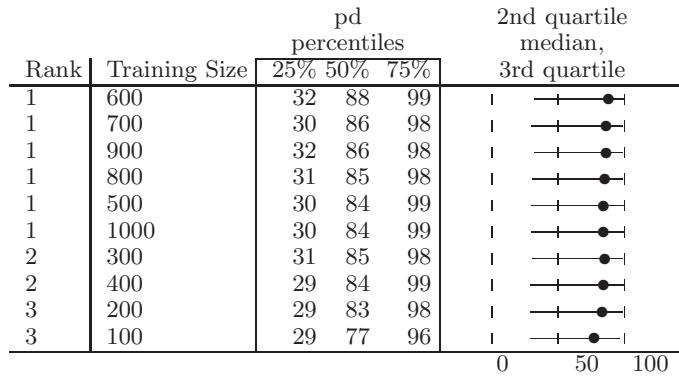


Figure 12. Experiment #1 - Part B - (Random Sampling). Probability of Detection (PD) results, sorted by rank then median values.

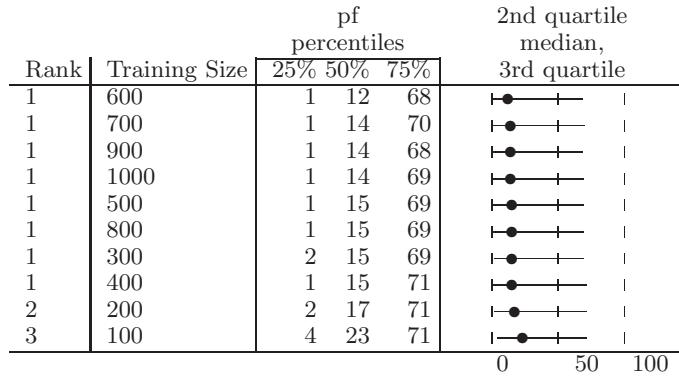


Figure 13. Experiment #1 - Part B - (Random Sampling). Probability of False Alarm (PF) results, sorted by rank then median values.

3.2. Experiment II: Within vs Cross-Company Data

OURMINE was used to reproduce Turhan et al.'s experiment - with a Naive Bayes classifier in conjunction with a k-Nearest Neighbor (k-NN) relevancy filter. Relevancy filtering is used to group similar instances together in order to obtain a learning set that is homogeneous with the testing set. Thus, by using a training set that shares similar characteristics with the testing

set, it is assumed that a bias in the model will be introduced. The k-NN filter works as follows. For each instance in the test set:

- the k nearest neighbors in the training set are chosen.
- duplicates are removed
- the remaining instances are used as the new training set.

3.2.1. Building the Experiment

The entire script used to conduct this experiment is shown in Figure 14.

To begin, the data in this study were the same as used by Gay et al. [5]; seven PROMISE defect data sets (CM1, KC1, KC2, KC3, MC2, MW1, PC1) were used to build seven combined data sets each containing $\frac{6}{7}$ -th of the data. For instance, the file *combined_PC1.arff* contains all seven data sets *except* PC1. This is used as the training set for the cross-company (CC) data. For example, if we wished to learn on all training data except for PC1, this would be a valid data set representation for a cross-company experiment.

Next, as can be seen in line 15 of Figure 14, a 10-way cross validation was conducted by calling *makeTrainAndTest*, which is a built-in OURMINE function that randomly shuffles the data and constructs both a test set, containing 10% of the data, and a training set containing 90% of the data. This was repeated ten times, and the resulting data was used in proceeding studies. For instance, in lines 18-24, the original test and training sets are used for the first WC study. However, in the WC experiment using relevancy filtering (lines 25-31), the same test set is used, but with the newly created training set. Lines 32-38 show the first CC study. This study is identical to the WC study except that as we saw before, we use *combined_X.arff* files, instead of *shared_X.arff*.

We chose to use a Naive Bayes classifier for this study because this is what was chosen in the original experiment conducted by Turhan et al. in [22], as well as because Naive Bayes has been shown to be competitive on PROMISE defect data against other learners [8].

3.2.2. Results

Our results for this experiment can be found in Figure 15 and Figure 16. The following are important conclusions derived from these results:

- When CC data is used, relevancy filtering is crucial. According to our results, cross-company data with no filtering yields the worst *pd* and *pf* values.
- When relevancy filtering is performed on this CC data, we obtain better *pd* and *pf* results than using just CC and Naive Bayes.
- When considering only filtered data or only unfiltered data, the highest *pd* and lowest *pf* values are obtained by using WC data as opposed to CC data. This suggests that WC data gives the best results.

These finds were consistent with Turhan et al.'s results:

- Significantly better defect predictors are produced from using WC data.

```

1 promiseDefectFilterExp(){
2 local learners="nb"
3 local datanames="CM1 KC1 KC2 KC3 MC2 MW1 PC1"
4 local bins=10
5 local runs=10
6 local out=$Save/defects.csv
7 for((run=1;run<=$runs;run++)); do
8   for dat in $datanames; do
9     combined=$Data/promise/combined_${dat}.arff
10    shared=$Data/promise/shared_${dat}.arff
11    blabln "data=${dat} run=$run"
12    for((bin=1;bin<=$bins;bin++)); do
13      rm -rf test.lisp test.arff train.lisp train.arff
14      cat $shared |
15      logArff 0.0001 "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" > logged.arff
16      makeTrainAndTest logged.arff $bins $bin
17      goals='cat $shared | getClasses --brief'
18
19      for learner in $learners; do
20        blabln "WC"
21        $learner train_shared.arff test_shared.arff | gotwant > produced.dat
22        for goal in $goals; do
23          extractGoals goal "${dat},${run},${bin},WC,$learner,$goal" `pwd`/produced.dat
24        done
25        blabln "WCkNN"
26        rm -rf knn.arff
27        $Clusterers -knn 10 test_shared.arff train_shared.arff knn.arff
28        $learner knn.arff test_shared.arff | gotwant > produced.dat
29        for goal in $goals; do
30          extractGoals goal "${dat},${run},${bin},WCkNN,$learner,$goal" `pwd`/produced.dat
31        done
32        blabln "CC"
33        makeTrainCombined $combined > com.arff
34        cat com.arff | logArff 0.0001 "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" > logged.arff
35        $learner logged.arff test_shared.arff | gotwant > produced.dat
36        for goal in $goals; do
37          extractGoals goal "${dat},${run},${bin},CC,$learner,$goal" `pwd`/produced.dat
38        done
39        blabln "CkNN"
40        makeTrainCombined $combined > com.arff
41        cat com.arff |
42        logArff 0.0001 "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" > logged.arff
43        $Clusterers -knn 10 test_shared.arff logged.arff knn.arff
44        $learner knn.arff test_shared.arff | gotwant > produced.dat
45        for goal in $goals; do
46          extractGoals goal "${dat},${run},${bin},CkNN,$learner,$goal" `pwd`/produced.dat
47        done
48      done
49    done
50  done ) | malign | sort -t, -r -n -k 12,12 > $out

```

Figure 14. The OURMINE script used in conducting the WC vs. CC experiment.

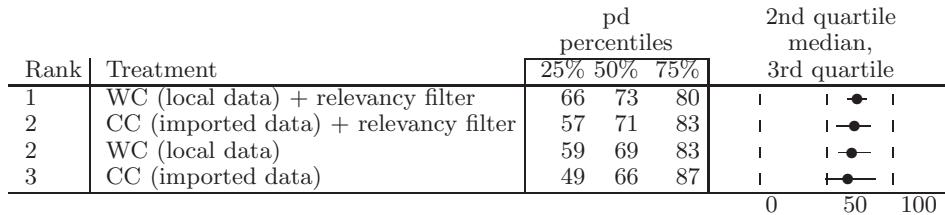


Figure 15. Experiment #2 (WC vs. CC). Probability of Detection (PD) results, sorted by rank then median values.

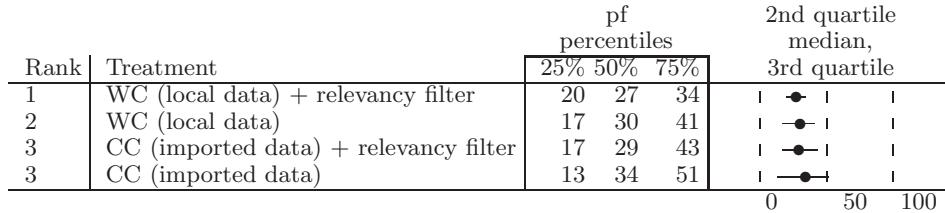


Figure 16. Experiment #2 (WC vs. CC). Probability of False Alarm (PF) results, sorted by rank then median values.

- However, CC data leads to defect predictors nearly as effective as WC data when using relevancy filtering.

Thus, this study also makes the same conclusions as Turhan et al. A company should use local data to develop defect predictors if that local development data is available. However, if local data is not available, relevancy-filtered cross-company data provides a feasible means to build defect predictors.

3.3. Experiment III: Scaling Up Text Miners

As stated above, the purpose of this experiment conducted for this paper is to verify if lightweight data mining methods perform worse than more thorough and rigorous ones.

The data sets used in this experiment are:

- EXPRESS schemas: AP-203, AP-214
- Text mining datasets: BBC, Reuters, The Guardian (multi-view text datasets), 20 Newsgroup subsets: sb-3-2, sb-8-2, ss-3-2, sl-8-2

3.3.1. Classes of Methods

This experiment compares different *row* and *column* reduction methods. Given a table of data where each row is one example and each columns counts different features, then:

- Row reduction methods *cluster* related rows into the same group;
- Column reduction methods remove columns with little information.

Reduction methods are essential in text mining. For example:

- A standard text mining corpus may store information in tens of thousands of columns. For such data sets, column reduction is an essential first step before any other algorithm can execute
- The process of clustering data into similar groups can be used in a wide variety of applications, such as:
 - Marketing: finding groups of customers with similar behaviors given a large database of customer data
 - Biology: classification of plants and animals given their features
 - WWW: document classification and clustering weblog data to discover groups of similar access patterns.

3.3.2. The Algorithms

While there are many clustering algorithms used today, this experiment focused on three: a naive K-Means implementation, GenIc [6], and clustering using canopies [11]:

1. K-means, a special case of a class of EM algorithms, works as follows:
 - (a) Select initial K centroids at random;
 - (b) Assign each incoming point to its nearest centroid;
 - (c) Adjusts each cluster's centroid to the mean of each cluster;
 - (d) Repeat steps 2 and 3 until the centroids in all clusters stop moving by a noteworthy amount

Here we use a naive implementation of K-means, requiring $K * N * I$ comparisons, where N and I represent the total number of points and maximum iterations respectively.

2. GenIc is a single-pass, stochastic clustering algorithm. It begins by initially selecting K centroids at random from all instances in the data. At the beginning of each generation, set the centroid weight to one. When new instances arrive, nudge the nearest centroid to that instance and increase the score for that centroid. In this process, centroids become “fatter” and slow down the rate at which they move toward newer examples. When a generation ends, replace the centroids with less than X percent of the max weight with N more random centroids. Genic repeats for many generations, then returns the highest scoring centroids.

[†]http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/

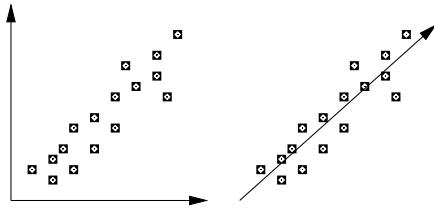


Figure 17. A PCA dimension feature.

3. Canopy clustering, developed by Google, reduces the need for comparing all items in the data using an expensive distance measure, by first partitioning the data into overlapping subsets called *canopies*. Canopies are first built using a cheap, approximate distance measure. Then, more expensive distance measures are used inside of each canopy to cluster the data.

As to column reduction, we will focus on two methods:

1. PCA, or Principal Components Analysis, is a reduction method that treats every instance in a dataset as a point in N-dimensional space. PCA looks for new dimensions that better fit these points—by mapping data points to these new dimensions where the variance is found to be maximized. Mathematically, this is conducted by utilizing eigenvalue decompositions of a data covariance matrix or singular value decomposition of a data matrix. Figure 17 shows an example of PCA. Before, on the left-hand-side, the data exists in a two-dimensional space, neither of which captures the distribution of the data. Afterwards, on the right-hand-side, a new dimension has been synthesized that is more relevant to the data distribution.
2. TF-IDF, or term frequency times inverse document frequency, reduces the number of terms (dimensions) by describing how important a term is in a document (or collection of documents) by incrementing its importance according to how many times the term appears in a document. However, this importance is also offset by the frequency of the term in the entire corpus. Thus, we are concerned with only terms that occur frequently in a small set of documents, and very infrequently everywhere else. To calculate the Tf*IDF value for each term in a document, we use the following equation:

$$Tf * Idf(t, D_j) = \frac{tf(t_i, D_j)}{|D_j|} \log\left(\frac{|D|}{df(t_i)}\right) \quad (1)$$

where $tf(t_i, D_j)$ denotes the frequency of term i in document j , and $df(t_i)$ represents the number of documents containing term i . Here, $|D|$ denotes the number of documents in the corpus, and $|D_j|$ is the total number of terms in document j . To reduce all terms (and thus, dimensions), we must find the sum of the above in order to assign values to terms across *all* documents

$$Tf * Idf_{sum}(t) = \sum_{D_j \in D} Tf * Idf(t, D_j) \quad (2)$$

In theory, TF*IDF and GenIc should perform worse than K-Means, canopy clustering and PCA:

- Any single-pass algorithm like GenIc can be confused by “order effects”; i.e. if the data arrives in some confusing order then the single-pass algorithm can perform worse than other algorithms that are allowed to examine all the data.
- TF*IDF is a heuristic method while PCA is a well-founded mathematical technique

On the other hand, the more rigorous methods are slower to compute:

- Computing the correlation matrix used by PCA requires at least a $O(N^2)$ calculation.
- As shown below, K-means is much slower than the other methods studied here.

3.3.3. Building the Experiment

This experiment was conducted entirely with OURMINE using a collection of BASH scripts, as well as custom Java code. The framework was built as follows:

1. A command-line API was developed in Java for parsing the data, reducing/clustering the data, and outputting the data. Java was chosen due to its preferred speed for the execution of computationally expensive instructions.
2. The data was then iteratively loaded into this Java code via shell scripting. This provides many freedoms, such as allowing parameters to be altered as desired, as well as outputting any experimental results in any manner seen fit.

Figure 18 shows the OURMINE code for clustering data using the K-means algorithm. Shell scripting provides us with much leverage in this example. For instance, by looking at Lines 2-5, we can see that by passing the function four parameters, we can cluster data in the range from $minK$ to $maxK$ on all data in $dataDir$. This was a powerful feature used in this experiment, because it provides the opportunity to run the clusterer across multiple machines simultaneously. As a small example, suppose we wish to run K-means across three different machines with a minimum K of 2 and a maximum K of 256. Since larger values of K generally yield longer runtimes, we may wish to distribute the execution as follows:

```
Machine 1: clusterKmeansWorker 256 256 0 dataDir
Machine 2: clusterKmeansWorker 64 128 2 dataDir
Machine 3: clusterKmeansWorker 2 32 2 dataDir
```

Lines 9-13 of Figure 18 load the data from $dataDir$ for every k , and formats the name of the output file. Then, lines 15-19 begin the timer, cluster the data, and output statistical information such as k , the dataset, and runtime of the clusterer on that data set. This file will then be used later in the analysis of these clusters.

Similarly, the flags in line 16 can be changed to perform a different action, such as clustering using GenIc or Canopy, by changing $-k$ to $-g$ or $-c$ respectively, as well as finding cluster similarities (as described below) and purities, by using $-sim$ and $-purity$ as inputs.

Since any number of variables can be set to represent different libraries elsewhere in OURMINE, the variable

```
$Reducers
```

```

1 clusterKmeansWorker(){
2     local minK=$1
3     local maxK=$2
4     local incVal=$3
5     local dataDir=$4
6     local stats="clusterer,k,dataset,time(seconds)"
7     local statsfile=$Save/kmeans_runtimes
8     echo $stats >> $statsfile
9     for((k=$minK;k<=$maxK;k*=$incVal)); do
10         for file in $dataDir/*.arff; do
11             filename=`basename $file`
12             filename=${filename%.*}
13             out=kmeans_k="$k"_$filename.arff
14             echo $out
15             start=$(date +%s.%N)
16             $Clusterers -k $k $file $Save/$out
17             end=$(date +%s.%N)
18             time=$(echo "$end - $start" | bc)
19             echo "kmeans,$k,$filename,$time" >> $statsfile
20         done
21     done
22 }

```

Figure 18. An OURMINE worker function to cluster data using the K-means algorithm. Note that experiments using other clustering methods (such as GenIC and Canopy), could be conducted by calling line 16 above in much the same way, but with varying flags to represent the clusterer.

is used for the dimensionality reduction of the raw dataset, as seen in Figure 19, whose overall structure is very similar to Figure 18.

3.4. Results

To determine the overall benefits of each clustering method, this experiment used both cluster similarities, as well as the runtimes of each method.

3.4.1. Similarities

Cluster similarities tell us how similar points are, either within a cluster (*Intra*-similarity), or with members of other clusters (*Inter*-similarity). The idea here is simple: gauge how well a clustering algorithm groups similar documents, and how well it separates different documents. Therefore, intra-cluster similarity values should be maximized, while minimizing inter-cluster similarities.

Similarities are obtained by using the cosine similarity between two documents. The cosine similarity measure defines the cosine of the angle between two documents, each containing vectors of terms. The similarity measure is represented as

```

1 reduceWorkerTfidf(){
2     local datadir=$1
3     local minN=$2
4     local maxN=$3
5     local incVal=$4
6     local outdir=$5
7     local runtimes=$outdir/tfidf_runtimes
8
9     for((n=$minN;n<=$maxN;n+=$incVal)); do
10        for file in $datadir/*.*; do
11            out=`basename $file`
12            out=${out%.*}
13            dataset=$out
14            out=tfidf_n="$n"_$out.arff
15            echo $out
16            start=$(date +%s)
17            $Reducers -tfidf $file $n $outdir/$out
18            end=$(date +%s)
19            time=$((end - start))
20            echo "tfidf,$n,$dataset,$time" >> $runtimes
21        done
22    done
23 }

```

Figure 19. An OURMINE worker function to reduce the data using TF-IDF.

$$sim(D_i, D_j) = \frac{D_i \cdot D_j}{\|D_i\| \|D_j\|} = \cos(\theta) \quad (3)$$

where D_i and D_j denote two term frequency vectors for documents i and j , and where the denominator contains magnitudes of these vectors.

Cluster similarities are determined as follows:

- Cluster intra-similarity: For each document d in cluster C_i , find the cosine similarity between d and all documents belonging to C_i
- Cluster inter-similarity: For each document d in cluster C_i , find the cosine similarity between d and all documents belonging to all other clusters

Thus the resulting sum of these values represents the overall similarities of a clustering solution. Figure 20 shows the results from the similarity tests conducted in this experiment. The slowest clustering and reduction methods were set as a baseline, because it was assumed that these methods would perform the best. With intra-similarity and inter-similarity values normalized to 100 and 0 respectively, we can see that surprisingly, faster heuristic clustering and reduction methods perform just as well or better than more rigorous methods. *Gain* represents the overall score used in the assessment of each method, and is computed as a method's cluster *intra*-similarity value minus its *inter*-similarity value. Thus, the conclusions

Reducer and Clusterer	Time	InterSim	IntraSim	Gain
TF-IDF*K-means	17.52	-0.085	141.73	141.82
TF-IDF*GenIc	3.75	-0.14	141.22	141.36
PCA*K-means	100.0	0.0	100.0	100.0
PCA*Canopy	117.49	0.00	99.87	99.87
PCA*GenIc	11.71	-0.07	99.74	99.81
TF-IDF*Canopy	6.58	5.02	93.42	88.4

Figure 20. Experiment #3 (Text mining). Similarity values normalized according to the combination of most rigorous reducer and clusterer. Note that *Gain* is a value representing the difference in cluster intrasimilarity and intersimilarity.

from this experiment shows that fast heuristic methods are sufficient for large data sets due to their scalability and performance.

4. Discussion

Productivity of a development environment depends on many factors. In our opinion, among the most important of these are simplicity, power and motivation provided by the environment. There exists an intricate balance between simplicity and power. If a tool is too simple, a user's actions are confined to a smaller space, thus decreasing the overall power and intricacy of the system. Also, exceedingly simple environments can restrict our learning process because we are not forced to learn or implement new ideas. However, even though a tool yields a great number of features, the options presented to a user can be overly complex and thus discourage further experimentation. We find that quick, yet careful experiment building promotes more motivation and interest to work on and finish the task.

Practitioners and researchers can both benefit from using OURMINE's syntax based on widely recognized languages. While practitioners are offered extensive freedom in modifying and extending the environment to be used as necessary, researchers can duplicate previously published experiments whose results could change the way we view data mining and software engineering.

5. Conclusions

The next challenge in the empirical SE community will be to not only share data, but to share experiments. We look forward to the day when it is routine for conference and journal submissions to come not just with supporting data but also with a fully executable version of the experimental rig used in the paper. Ideally, when reviewing papers, program committee members could run the rig and check if the results recorded by the authors are reproducible.

This paper has reviewed a UNIX scripting tool called OURMINE as a method of documenting, executing, and sharing data mining experiments. We have used OURMINE to reproduce and check several important result. In our first experiment, we learned that:

- First, “tune” our predictors to best fit a data set or corpus, and to select the winning method based on results.
- Secondly, understand how *early* we can apply these winning methods to our data by determining the fewest number of examples required in order to learn an adequate theory.

In our second experiment, we concluded that:

- When local data is available, that data should be used to build defect predictors
- If local data is not available, however, imported data can be used to build defect predictors when using *relevancy filtering*
- Imported data that uses *relevancy filtering* performs nearly as well as using local data to build defect predictors

In our third experiment, we learned:

- When examining cluster inter/intra similarities resulting from each clustering/reduction solution, we found that faster heuristic methods can outperform more rigorous ones when observing decreases in runtimes.
- This means that faster solutions are suitable on large datasets due to *scalability*, as well as performance.

We prefer OURMINE to other tools. Four features are worthy of mention:

1. OURMINE is very succinct. As seen above, a few lines can describe even complex experiments.
2. OURMINE’s experimental descriptions are complete. There is nothing hidden in Figure 14; it is not the pseudocode of an experiment, it is the experiment.
3. OURMINE code like in Figure 14, Figure 18 and Figure 19 is executable and can be executed by other researchers directly.
4. Lastly, the execution environment of OURMINE is readily available. Unlike RAPID-I, WEKA, “R”, etc, there is nothing to debug or install. Many machines already have the support tools required for OURMINE. For example, we have run OURMINE on Linux, Mac, and Windows machines (with Cygwin installed).

Like Ritthol et al., we doubt that the standard interfaces of tools like WEKA, etc, are adequate for representing the space of possible experiments. Impressive visual programming environments are not the answer: their sophistication can either distract or discourage novice data miners from extensive modification and experimentation. Also, we find that the functionality of the visual environments can be achieved with a few BASH and GAWK scripts, with a fraction of the development effort and a greatly increased chance that novices will modify the environment.

OURMINE is hence a candidate format for sharing descriptions of experiments. The data mining community might find this format unacceptable but discussions about the drawbacks (or strengths) of OURMINE would help evolve not just OURMINE, but also the discussion on how to represent data mining experiments for software engineering.

REFERENCES

1. Brian W. Kernighan Alfred V. Aho and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
2. Zhihao Chen, Tim Menzies, Dan Port, and Barry Boehm. Finding the right data for software cost modeling. *IEEE Software*, Nov 2005.
3. Jacob Eisenstein and Randall Davis. Visual and linguistic information in gesture classification. In *ICMI*, pages 113–120, 2004. Available from <http://iccle.googlecode.com/svn/trunk/share/pdf/eisenstein04.pdf>.
4. Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *In Machine Learning: Proceedings of the Sixteenth International Conference*, pages 124–133. Morgan Kaufmann, 1999.
5. Greg Gay, Tim Menzies, and Bojan Cukic. How to build repeatable experiments. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.
6. Chetan Gupta and Robert Grossman. Genic: A single pass generalized incremental algorithm for clustering. In *In SIAM Int. Conf. on Data Mining*. SIAM, 2004.
7. B. A. Kitchenham, E. Mendes, and G. H. Travassos. Cross- vs. within-company cost estimation studies: A systematic review. *IEEE Transactions on Software Engineering*, pages 316–329, May 2007.
8. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, May 2008. Available from <http://iccle.googlecode.com/svn/trunk/share/pdf/lessmann08.pdf>.
9. R. Loui. Gawk for ai. *Class Lecture*. Available from <http://menzies.us/cs591o/?lecture=gawk>.
10. A. Matheny. Scaling up text mining, 2009. Masters thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University.
11. Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, New York, NY, USA, 2000. ACM.
12. T. Menzies. Evaluation issues for visual programming languages, 2002. Available from <http://menzies.us/pdf/00vp.pdf>.
13. T. Menzies, D. Port, Z. Chen, J. Hihn, and S. Stukes. Specialization and extrapolation of induced domain models: Case studies in software effort estimation. 2005. IEEE ASE, 2005, Available from <http://menzies.us/pdf/05learncost.pdf>.
14. T. Menzies, B. Turhan, A. Bener, and J. Distefano. Cross- vs within-company defect prediction studies. 2007. Available from <http://menzies.us/pdf/07ccwc.pdf>.
15. Tim Menzies, Zhihao Chen, Jairus Hihi, and Karen Lum. Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, November 2006. Available from <http://menzies.us/pdf/06coseekmo.pdf>.
16. Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from <http://menzies.us/pdf/06learnPredict.pdf>.
17. I. Mierswa, M. Wurst, and R. Klinkenberg. Yale: Rapid prototyping for complex data mining tasks. In *KDD'06*, 1996.
18. A.S. Orrego. Sawtooth: Learning from huge amounts of data, 2004.
19. Chet Ramey. Bash, the bourne-again shell. 1994. Available from <http://tiswww.case.edu/php/chet/bash/rose94.pdf>.
20. Juan Ramos. Using tf-idf to determine word relevance in document queries. In *Proceedings of the First Instructional Conference on Machine Learning*, 2003. Available from <http://www.cs.rutgers.edu/~mlittman/courses/ml03/iCML03/papers/ramos.pdf>.
21. O. Ritthoff, R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske. Yale: Yet another learning environment. In *LLWA 01 - Tagungsband der GI-Workshop-Woche, Dortmund, Germany*, pages 84–92, October 2001. Available from <http://ls2-www.cs.uni-dortmund.de/~fischer/publications/YaleLLWA01.pdf>.
22. Burak Turhan, Tim Menzies, Ayse B. Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 2009. Available from <http://menzies.us/pdf/08ccwc.pdf>.

Installing OURMINE

OURMINE is an open source toolkit licensed under GPL 3.0. It can be downloaded and installed from <http://code.google.com/p/ourmine>.

OURMINE is a command-line environment, and as such, system requirements are minimal. However, in order to use OURMINE three things must be in place:

- A Unix-based environment. This does not include Windows. Any machine with OSX or Linux installed will do.
- The Java Runtime Environment. This is required in order to use the WEKA, as well as any other Java code written for OURMINE.
- The GAWK Programming Language. GAWK will already be installed with up-to-date Linux versions. However, OSX users will need to install this.

To install and run OURMINE, navigate to <http://code.google.com/p/ourmine> and follow the instructions.

Built-in OURMINE Functions

Utility Functions I

Function Name	Description	Usage
abcd	Performs confusion matrix computations on any classifier output. This includes statistics such as \$pd, \$pf, \$accuracy, \$balance and \$f-measure	<code>— abcd —prefix —goal</code> , where <i>prefix</i> refers to a string to be inserted before the result of the <i>abcd</i> function, and <i>goal</i> is the desired class of a specific instance.
arffToLisp	Converts a single .arff file into an equivalent .lisp file	<code>arffToLisp \$dataset.arff</code>
blab	Prints to the screen using a separate environment. This provides the ability to print to the screen without the output interfering with the results of an experiment	<code>blab \$message</code>
blabln	The same as blab, except this will print a new line after the given output	<code>blabln \$message</code>
docsToSparff	Converts a directory of document files into a sparse .arff file. Prior to building the file, however, the text is cleaned	<code>docsToSparff \$docDirectory \$output.sparff</code>
docsToTfidfSparff	Builds a sparse .arff file from a directory of documents, as above, but instead constructs the file based on TF-IDF values for each term in the entire corpus.	<code>docsToTfidfSparff \$docDirectory \$numberOfAttributes \$output.sparff</code>
formatGotWant	Formats an association list returned from any custom LISP classifier containing actual and predicted class values in order to work properly with existing OURMINE functions	<code>formatGotWant</code>
funs	Prints a sorted list of all available OURMINE functions	<code>funs</code>
getClasses	Obtains a list of all class values from a specific data set	<code>getClasses</code>
getDataDefun	Returns the name of a .arff relation to be used to construct a LISP function that acts as a data set	<code>getDataDefun</code>
gotwant	Returns a comma separated list of actual and predicted class values from the output of a WEKA classifier	<code>gotwant</code>
help	When given with an OURMINE function, prints helpful information about the function, such as a description of the function, how to use it, etc.	<code>help \$function</code> , where \$function is the name of the function

Utility Functions II

Function Name	Description	Usage
makeQuartiles	Builds quartile charts using any key and performance value from the abcd results (see above)	<i>makeQuartiles \$csv \$keyField \$performanceField</i> , where \$keyField can be a learner/treatment, etc., and \$performanceField can be any value desired, such as <i>pd</i> , <i>accuract</i> , etc.
makeTrainAndTest	Constructs a training set and a test set given an input data set. The outputs of the function are train.arff, test.arff and also train.lisp and test.lisp	<i>makeTrainAndTest \$dataset \$bins \$bin</i> , where \$dataset refers to any data set in correct .arff format, \$bins refers to the number of bins desired in the construction of the sets, and \$bin is the bin to select as the test set. For instance, if 10 is chosen as the number of bins, and 1 is chosen as the test set bin, then the resulting training set would consist of 90% of the data, and the test set would consist of 10%.
malign	Neatly aligns any comma-separated format into an easily readable format	<i>malign</i>
medians	Computes median values given a list of numbers	<i>medians</i>
quartile	Generates a quartile chart along with min/max/median values, as well as second and third quartile values given a specific column	<i>quartile</i>
show	Prints an entire OURMINE function so that the script can be seen in its entirety	<i>show \$functionName</i>
winLossTie	Generates win-loss-tie tables given a data set. Win-loss-tie tables, in this case, depict results after a statistical analysis test on treatments. These tests include the Mann-Whitney-U test, as well as the Ranked Wilcoxon test	<i>winLossTie -input \$input.csv -fields \$numOfFields -perform \$performanceField -key \$keyField -\$confidence</i> , where \$input.csv refers to the saved output from the <i>abcd</i> function described above, \$numOfFields represents the number of fields in the input file, \$performanceField is the field on which to determine performance, such as <i>pd</i> , <i>pf</i> , <i>acc</i> , \$keyField is the field of the key, which could be a learner/treatment, etc., and \$confidence is the percentage of confidence when running the test. The default confidence value is 95%

Learners

Function Name	Description	Usage
adtree	Calls WEKA's Alternating Decision Tree	<i>adtree \$strain \$test</i>
bnet	Calls WEKA's Bayes Net	<i>bnet \$strain \$test</i>
j48	Calls WEKA's J48	<i>j48 \$strain \$test</i>
nb	Calls WEKA's Naive Bayes	<i>nb \$strain \$test</i>
oner	Calls WEKA's One-R	<i>oner \$strain \$test</i>
rbfnet	Calls WEKA's RBFNet	<i>rbfnet \$strain \$test</i>
ridor	Calls WEKA's RIDOR	<i>ridor \$strain \$test</i>
zeror	Calls WEKA's Zero-R	<i>zeror \$strain \$test</i>

Preprocessors

Function Name	Description	Usage
caps	Reduces capitalization to lowercase from an input text	<i>caps</i>
clean	Cleans text data by removing capitals, words in a stop list, special tokens, and performing Porter's stemming algorithm	<i>clean</i>
discretize	Discretizes the incoming data via WEKA's discretizer	<i>discretize \$input.darff \$output.arff</i>
logArff	Logs numeric data in incoming data	<i>logArff \$minVal \$fields</i> , where \$minVal denotes the minimum value to be passed to the log function, and \$fields is the specific fields on which to perform log calculations
stems	Performs Porter's stemming algorithm on incoming text data	<i>stems \$inputFile</i>
stops	Removes any terms from incoming text data that are in a stored stop list	<i>stops</i>
tfidf	Computes TF*IDF values for terms in a document	<i>tfidf \$file</i>
tokes	Removes unimportant tokens or whitespace from incoming textual data	<i>tokes</i>

Feature Subset Selectors

Function Name	Description	Usage
cfs	Calls WEKA's Correlation-based Feature Selector	<i>cfs \$input.arff \$numAttributes \$out.arff</i>
chisquared	Calls WEKA's Chi-Squared Feature Selector	<i>chisquared \$input.arff \$numAttributes \$out.arff</i>
infogain	Calls WEKA's Infogain Feature Selector	<i>infogain \$input.arff \$numAttributes \$out.arff</i>
oneR	Calls WEKA's One-R Feature Selector	<i>oneR \$input.arff \$numAttributes \$out.arff</i>
pca	Calls WEKA's Principal Components Analysis Feature Selector	<i>pca \$input.arff \$numAttributes \$out.arff</i>
relief	Calls WEKA's RELIEF Feature Selector	<i>relief \$input.arff \$numAttributes \$out.arff</i>

Clusterers

Function Name	Description	Usage
K-means	Calls custom Java K-means	<i>\$Clusterers -k \$k \$input.arff \$out.arff, where \$k is the initial number of centroids</i>
Genic	Calls custom Java GeNic	<i>\$Clusterers -g \$k \$n \$input.arff \$out.arff, where \$k is the initial number of centroids, and \$n is the size of a generation</i>
Canopy	Calls custom Java Canopy Clustering	<i>\$Clusterers -c \$k \$p1 \$p2 \$input.arff \$out.arff, where k is the initial number of centroids, \$p1 is a similarity percentage value for the outer threshold, and \$p2 is a similarity percentage value for the inner threshold. If these percentages are not desired, a value of 1 should be provided for both</i>
EM	Calls WEKA's Expectation-Maximization Clusterer	<i>em \$input.arff \$k, where \$k is the initial number of centroids</i>

Measuring the Heterogeneity of Cross-company Dataset

Jia Chen, Ye Yang, Wen Zhang

Institute of Software,
Chinese Academy of Sciences,
Beijing, China

{chenjia, ye, zhangwen}@itechs.icas.ac.cn

Gregory Gay

Lane Department of Computer Science and
Electrical Engineering,
West Virginia University,
Morgantown, WV, USA

gregoryg@csee.wvu.edu

ABSTRACT

As a standard practice, general effort estimate models are calibrated from large cross-company datasets. However, many of the records within such datasets are taken from companies that have calibrated the model to match their own local practices. Locally calibrated models are a double-edged sword; they often improve estimate accuracy for that particular organization, but they also encourage the growth of local biases. Such biases remain present when projects from that firm are used in a new cross-company dataset. Over time, such biases compound, and the reliability and accuracy of a general model derived from the data will be affected by the increased level of heterogeneity. In this paper, we propose a statistical measure of the exact level of heterogeneity of a cross-company dataset. In experimental tests, we measure the heterogeneity of two COCOMO-based datasets and demonstrate that one is more homogeneous than the other. Such a measure has potentially important implications for both model maintainers and model users. Furthermore, a heterogeneity measure can be used to inform users of the appropriate data handling techniques.

Categories and Subject Descriptors

D.2.9 [Software Engineering]: Management – cost estimation, time estimation

General Terms

Economics, Experimentation, Management

Keywords

Heterogeneous datasets, software effort estimation, parameter comparison, estimation model calibration.

1. INTRODUCTION

Obtaining an accurate estimate of software effort is of great value for project managers as well as other stakeholders. Such an estimate can be used to appropriately arrange various activities of software development, as well as to allocate the suitable amount of resources to these activities. However, obtaining an estimate that is both accurate and reliable is a difficult task. To address this problem, many different software effort estimation methods have been proposed. These methods tend to fall into three categories: expert-based, analogy-based, and model-based estimation.

Model-based estimation, as the name implies, makes use of a mathematical model, such as COCOMO [3, 4], to produce effort estimates. This approach is also known as parametric estimation, as there are several variable parameters within the model that must be determined before that model is used. Initially, the values of

these parameters are often supplied by experts or the model's designers. The software engineering industry changes rapidly, and regardless of the model's accuracy, the parameters used to estimate projects from the previous decade are unlikely to remain relevant to modern projects. The regular re-calibration of model parameters, either through a series of new expert judgments or through the use of an Ordinary Least Squares (OLS)-based method, ensures the reliability of a model's estimates.

The calibration of the general COCOMO model has met with several problems, the most major of which is the existence of counter-intuitive – that is, negative – regression coefficients [4, 5, 6, 7]. Such coefficients make little logical or practical sense; a higher level of programmer capability (PCAP) leads to a decrease in the calculated project effort, but a negative coefficient indicates that higher PCAP, unreasonably, increases the effort level.

To adapt a general COCOMO model calibrated from a cross-company dataset to the local environment of a particular firm, it is advised to locally-calibrate the general model, which involves tuning the two constants representing the overall productivity of the firm [4, 5]. While the local model often provides more accurate estimates for that company in the short term, its long-term use can be harmful. If local calibration is highly effective for a company, it logically follows that this company's practices differ from the mainstream (that is, the average practices that can be summarized from the general model) by some significant amount. In other words, firms that benefit from local calibration demonstrate a local bias. These firms may grow content with the accurate estimates provided by their local models, and will be unlikely to change their practices. Over time, even if their models still provide accurate estimates, they will grow more unreliable.

We propose an approach to measuring the heterogeneity of COCOMO datasets by comparing the calibrated parameters of a cross-company dataset with a derived version of that dataset where the effort values have been replaced with those given by locally-calibrated models. This is because we believe that the heterogeneity is aggregated into the estimate given by each local model. By this comparison, we can calculate a measure of the heterogeneity of the original dataset.

2. Related Work

There is a large body of work on comparing software effort estimation models derived from within-company datasets with those derived from cross-company datasets. Kitchenham et al. [1] systematically reviewed 10 such papers with the aim of determining under what circumstances estimation models derived from cross-company datasets are as good as those derived from within-company datasets. It was only certain that models derived from

within-company datasets were significantly better (that is, more accurate) than models derived from cross-company datasets when the within-company datasets were small (less than 20 projects) and leave-one-out cross validation was used.

Jeffery et al. [2] compared the accuracy of estimation models derived from the ISBSG repository (a cross-company dataset) with those derived from the dataset of an Australian company called Megatec (a within-company dataset). They found that the model derived from the within-company dataset was significantly more accurate than the model derived from the cross-company dataset. Several papers have supported this conclusion [8, 9], while others have rejected it (see [1] for a complete list).

Rather than pure model comparison, some authors have focused on the preliminary analysis of datasets used to build an estimation model. Kitchenham [11] proposed a procedure for analyzing unbalanced datasets, and helped to explain the difficult situation that happened when COCOMO II was initially calibrated [5]. Based on forward pass residual analysis, the procedure identifies really significant factors, and then produces a better model. Another paper by Liu et al. [10] proposed a rather generic framework for preliminary analysis of cost estimation dataset. Using this framework, analyst can systematically remove outliers and identify dominant variables.

In order to improve the accuracy of estimation models derived from datasets with heterogeneous sources (such as the ISBSG database), Cuadrado-Gallego et al. [12] proposed an automated segmentation process that splits a single parametric model into a number of sub-models. However, the segmentation of a general model will dramatically decrease the maintainability of the general model over time, and lead to the lack of a common basis for comparing estimates produced by different model variants.

We focus on the task of measuring the pure level of heterogeneity in a cross-company dataset and what implications such heterogeneity has for the life cycle management of cost models. Instead of comparing estimation accuracy, we compare calibrated parameters and come to a specific measurement.

3. RESEARCH METHOD

3.1 Overview

We first take the cross-company dataset, called the original dataset, and filter out any unusable within-company subsets (those with less than three projects are too small to make use of). Using the slightly smaller original dataset, we build a second, derived, dataset as follows:

- (1) The original dataset consists of several disjoint subsets, each consisting of records from a single organization. We derive a local model for each of those subsets through calibrating the A and B constants by the OLS method.
- (2) For each project contained in a within-company dataset, we calculate its effort estimate using the local model for the organization that contributes the project.
- (3) Copy the original dataset, replacing the recorded actual effort with the model's estimate. The derived dataset differs from the original dataset only in effort values.

These two datasets (the original and derived datasets) form the basis of the subsequent heterogeneity comparison. We then repeat the following three steps in sequence for both datasets.

- (1) Randomly select about 90% of projects from the dataset.
- (2) Calibrate the parameters of a general model from selected projects by the OLS method.
- (3) Save all of the calibrated parameters as a new element of a predefined array. Each element is a set of values for all calibrated parameters.

After a specified number of trials, we fill two arrays of value-sets for all of the calibrated parameters (one array per dataset). From another perspective, there are two arrays of values for each calibrated parameter. We regard each array as a random sample of the same calibrated parameter. In other words, we actually take a pair of random samples for each calibrated parameter. By means of the statistical test defined in Section 3.4, we determine whether or not the difference of two means of a calibrated parameter is equal to zero. The p-value of the statistical test is used as an indicator of the difference.

Finally, we count the number of calibrated parameters whose p-values are relatively large, compared with others. We use the proportion of calibrated parameters with small p-values as a positive indicator of the degree of heterogeneity. This provides a quantitative measure of the heterogeneity of a cross-company dataset. The foregoing process is applied to both NASA and USC datasets. Since it has a parameter that specifies the times of repetition, this parameter's value is varied in order to test its effect on the results of our measurement.

3.2 Datasets

We examine two datasets in this paper, and explore their varying degrees of heterogeneity. The first is the NASA93 dataset from the National Aeronautics and Space Administration (NASA). The other is a subset of the COCOMO II dataset from the University of Southern California (USC). Both datasets use a variant of the COCOMO software effort estimation model. Both contain effort multipliers and the two COCOMO constants as variable parameters, but the NASA dataset lacks scale factors. Readers are referred to Boehm [3] and Boehm et al [4] for detailed definitions of these effort multipliers and scale factors.

The software size is measured in KSLOC (thousand of logical line of code), and the development effort is measured in PM (Person Month). Table 1 compares some statistics for the software size and development effort of these two datasets. In this table, we see that the NASA dataset has less variety than the USC dataset.

Table 1 Software size and development effort of NASA and USC datasets

	Software Size		Development Effort	
	NASA	USC	NASA	USC
Mean	94.02	130.9	624.41	711.03
S.D.	133.6	236.23	1135.93	1519.3
Min	0.9	2.6	8.4	6
Max	980	1292.8	8211	11400

Projects of the NASA dataset are contributed by several centers that are geographically distributed across the United States, and we treat each center as an individual company. Similarly, projects of the USC dataset are contributed by many organizations, and we treat each organization as an individual company. Thus, we divide

each dataset into several disjoint subsets whose projects are contributed by the same company. Each whole dataset is, by definition, a cross-company dataset, and each such subset of it forms a within-company dataset.

3.3 Data Preparation

In the local COCOMO model, there are only two calibrated parameters: A and B. In order to calibrate them by the OLS method from a within-company dataset, there must be no less than three projects in the dataset. As we need to build a local model for each within-company dataset, we have to exclude those within-company datasets whose sizes are less than three. Applying this filter to NASA and USC datasets, we excluded one within-company dataset from the former, and two within-company datasets from the latter. As a result, 91 projects of the NASA dataset are distributed among 4 within-company datasets, and 158 projects of the USC dataset are distributed among 14 within-company datasets. Within-company datasets range in size from 3 to 39 for the NASA dataset, and from 3 to 48 for the USC dataset.

3.4 Statistical Test

As a result of repeatedly calibrating parameters, each of these parameters has two arrays of values (see Section 3.1). One array is from the original dataset, and the other is from the derived dataset. Each array can be regarded as a random sample of the same calibrated parameter and each of its elements as an observation of the sample. Based on these observations, we can calculate the sample mean and sample variance for hypothesis testing.

We do not directly test the null hypothesis that the difference between two means of a calibrated parameter is equal to zero. Instead, we calculate the p-value of such a test, which is defined by the following equations.

$$p = 2 \times (1 - \text{PDF}_Z(z))$$

$$z = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_1^2 + s_2^2}{n}}}, \quad \text{PDF}_Z(z) = P\{\mathbf{Z} \leq z\}$$

$$\mathbf{Z} \sim \mathbf{N}(0,1)$$

Where \bar{x} and \bar{y} are the two sample means of a calibrated parameter, s_1^2 and s_2^2 are the two sample variances of a calibrated parameter, n is the number of observations in a sample, \mathbf{Z} is a random variable of standard normal distribution, and PDF_Z denotes the Probability Distribution Function of \mathbf{Z} .

Note that these two samples are of equal size, because we specify the same number of trials for both datasets. That is, the sample size equals the times of repetition. We do not use the common t-test, because it assumes that two distributions have the same variance, but we observe that the two sample variances are quite different. In the equation for z , we replace the variances of the two distributions with the sample variances respectively, simply because the former is not available and the latter is a good approximation. For our hypothesis, a lower p-value implies that

the two means of a calibrated parameter are more probably different from a statistical perspective.

3.5 Measure of Heterogeneity

For each of the calibrated parameters, we calculate the p-values of the statistical test defined in the previous subsection. There are some calibrated parameters whose p-values are relatively smaller than others. We count the number of these small p-values and calculate their proportion with regard to total number of calibrated parameters. This proportion is proposed as a positive indicator for the degree of heterogeneity of a cross-company dataset. The larger the proportion is, the greater the heterogeneity.

$$\text{Heterogeneity} = \frac{s}{n}$$

Where, s denotes the number of calibrated parameters that has small p-values, and n denotes the total number of calibrated parameters.

Currently, we define a p-value as being small if it is less than 0.025. The value is chosen because it is a common choice of significance level for hypothesis testing, and it acts as a clear boundary when the heterogeneity is calculated for the two cross-company datasets used in this paper.

4. RESULTS AND DISCUSSION

Two cross-company datasets are used in this paper. One is the NASA dataset, and the other is the USC dataset. We apply the process summarized in Section 3.1 to each of these datasets three times, each time with a different value for the parameter that specifies how many calibration trials are conducted. Doing this, we can see how the p-value derived from statistical test varies with this parameter.

Table 2 Calibrated parameters with large p-values

Trials	NASA			USC	
	B	VIRT	MODP	TEAM	RUSE
36	0.5642	0.534	0.1477	0.1174	0.08853
48	0.5322	0.3397	0.1532	0.06147	0.1527
60	0.5247	0.4184	0.04999	0.02535	0.05405

Table 2 lists the exact number of p-value for the calibrated parameters with large p-values. For most of these calibrated parameters, their statistical tests reject the null hypothesis (that the difference between two means of a calibrated parameter is zero) at a significance level of 0.05.

The USC dataset has a *smaller* proportion (2/25) of calibrated parameters with *large* p-values, compared with the NASA dataset (3/17). Therefore, the former has *greater* degree of heterogeneity than the latter. This result agrees with two characteristics of these two datasets. And these two characteristics help to explain why the USC dataset has a greater degree of heterogeneity than the NASA dataset.

- (1) The USC dataset has a greater degree of variety in the software size and development effort than the NASA dataset (see Table 1). These two attributes can substantially influence the values of calibrated parameters.

- (2) There are merely 4 within-company datasets in the NASA dataset, but 14 within-company datasets in the USC dataset. It is usually true that a cross-company dataset consisting of more within-company datasets has a greater degree of heterogeneity.

However, these two intuitive characteristics cannot determine the degree of heterogeneity. Suppose a large cross-company dataset consists of many homogeneous within-company datasets, it can demonstrate great variety as long as all the within-company datasets demonstrate it, but little heterogeneity will be measured.

Before calibrating a general estimation model such as COCOMO model, our method may be used to analyze the heterogeneity of the source dataset. Based on the results of analysis, a user can determine whether a cross-company dataset is appropriate for calibrating a general estimation model. In many cases, however, model users and maintainers alike are restricted to what data they have in their possession. If there is a high degree of heterogeneity in the dataset, it is inadvisable to use the data “as is.” Instead, one approach suggested by Cuadrado-Gallego et al. [12] could be used to form a segmented (composite) general model that consists of several sub-models, instead of using an overall (singleton) general model that consists of a single equation. However, this approach cannot be applied to COCOMO, because its definition strictly stipulates that a single equation should be used.

An open research question is the exact definition of small p-value. We suggest that a relative approach should be adopted for determining the appropriate level for small p-value, because the appropriate level largely depends on the method used to calibrate the general estimation model. In this paper, we used the Ordinary Least Squares method and assigned a level of 0.025 to the small p-value. However, this is not universally appropriate - other methods may require different levels for small p-value.

5. CONCLUSIONS

In this paper, we propose a method that statistically compares the calibrated parameters of a general estimation model on the basis of a cross-company dataset. The results of this comparison can be used to calculate the heterogeneity of this cross-company dataset. We propose that the proportion of calibrated parameters with small p-values should be used as a positive indicator of the degree of heterogeneity. The larger the proportion is, the greater the heterogeneity.

The ability to measure the heterogeneity of a dataset has important implications for both the maintainers of the general model and for the organizations that typically employ locally-biased models. By using more homogenous datasets, general models can be frequently re-calibrated with some expectation of reliability. This would ease the difficulty of maintaining such models and would help ensure that general models remain relevant to the frequently changing state of the software engineering field. A more homogenous general model, or even a method of determining the heterogeneity of the data that was used to calibrate the general model, could then create a feedback loop where an organization uses the more homogeneous general model to evolve their practices into proximity to the mainstream, which in turn would help further homogenize the general model. Furthermore, such a measure can be used to steer model users away from overtly heterogeneous datasets. Model users restricted to heterogeneous

datasets can make more informed decisions about how to use their data. Rather than using it “as is,” they may elect to use a data preprocessing technique to filter out some of the heterogeneity.

6. ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 90718042, 60873072, and 60803023; the National Hi-Tech R&D Plan of China under Grant No. 2007AA010303; the National Basic Research Program under Grant No. 2007CB310802.

7. REFERENCES

- [1] B. A. Kitchenham, E. Mendes, and G. H. Travassos, “Cross versus within-company cost estimation studies: A systematic review,” *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 316-329, May, 2007.
- [2] R. Jeffery, M. Ruhe, and I. Wieczorek, “A comparative study of two software development cost modeling techniques using multi-organizational and company-specific data,” *Information and Software Technology*, vol. 42, no. 14, pp. 1009-1016, Nov, 2000.
- [3] B. W. Boehm, *Software Engineering Economics*: Prentice Hall PTR, 1981.
- [4] B. W. Boehm, Clark, Horowitz et al., *Software Cost Estimation with Cocomo II with Cdrom*: Prentice Hall PTR, 2000.
- [5] B. Clark, S. Devnani-Chulani, B. Boehm et al., "Calibrating the COCOMO II Post-Architecture model," *Proceedings of the 1998 International Conference on Software Engineering*, International Conference on Software Engineering, pp. 477-480, Los Alamitos: IEEE Computer Soc, 1998.
- [6] V. Nguyen, B. Steele, B. Boehm et al., *A Constrained Regression Technique for COCOMO Calibration*, New York: Assoc Computing Machinery, 2008.
- [7] S. Chulani, B. Boehm, and B. Steele, “Bayesian analysis of empirical software engineering cost models,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 573-583, Jul-Aug, 1999.
- [8] E. Mendes, B. Kitchenham, and s. IEEE computer, *Further comparison of cross-company and within-company effort estimation models for web applications*, Los Alamitos: IEEE Computer Soc, 2004.
- [9] K. Maxwell, L. Van Wassenhove, and S. Dutta, “Performance evaluation of general and company specific models in software development effort estimation,” *Management Science*, vol. 45, no. 6, pp. 787-803, Jun, 1999.
- [10] Q. Liu, and R. Mintram, “Preliminary data analysis methods in software estimation,” *Software Quality Journal*, vol. 13, no. 1, pp. 91-115, Mar, 2005.
- [11] B. Kitchenham, “A procedure for analyzing unbalanced datasets,” *IEEE Transactions on Software Engineering*, vol. 24, no. 4, pp. 278-301, Apr, 1998.
- [12] J. J. Cuadrado-Gallego, and M. A. Sicilia, “An algorithm for the generation of segmented parametric software estimation models and its empirical evaluation,” *Computing and Informatics*, vol. 26, no. 1, pp. 1-15, 2007.

Automatically Finding the Control Variables for Complex System Behavior

Gregory Gay¹, Tim Menzies¹, Misty Davies², and Karen Gundy-Burlet²

¹ West Virginia University, Morgantown, WV, USA

² NASA Ames Research Center, Moffett Field, CA, USA

greg@greggay.com, tim@menzies.us,

misty.d.davies@nasa.gov, karen.gundy-burlet@nasa.gov

Abstract. Testing large-scale systems is expensive in terms of both time and money. Running simulations early in the process is a proven method of finding the design faults likely to lead to critical system failures, but determining the exact cause of those errors is still time-consuming and requires access to a limited number of domain experts. It is desirable to find an automated method that explores the large number of combinations and is able to isolate likely fault points.

Treatment learning is a subset of minimal contrast-set learning that, rather than classifying data into distinct categories, focuses on finding the unique factors that lead to a particular classification. That is, they find the smallest change to the data that causes the largest change in the class distribution. These treatments, when imposed, are able to identify the factors most likely to cause a mission-critical failure. The goal of this research is to comparatively assess treatment learning against state-of-the-art numerical optimization techniques. To achieve this, this paper benchmarks the TAR3 and TAR4.1 treatment learners against optimization techniques across three complex systems, including two projects from the Robust Software Engineering (RSE) group within the National Aeronautics and Space Administration (NASA) Ames Research Center. The results clearly show that treatment learning is both faster and more accurate than traditional optimization methods.

1 Introduction

Large-scale industrial systems, often containing both hardware and software components, are expensive to design and expensive to build. The cost of design failures decreases exponentially the earlier that these faults are discovered in the design process [8]. When a system is expensive (especially when safety is part of the system cost), it is critical to use the best tools available at each stage of the design life cycle. Early in the design process, lower-level requirements (e.g. the center of gravity shall be placed exactly *here*) are built from higher-level requirements (e.g. the craft shall not spin about any axis above some rate) using simplified models and a factor of safety. As the complexity of

the system grows, the compounded errors in the simplified models can erode the factor of safety until the overall system is not as robust as originally intended. As a last line of defense before a complete prototype is constructed, high-fidelity simulations composed of models for the entire system—hardware, software and environment—are used to eliminate potentially costly fault points and estimate the overall system margins to failure.

However, simply simulating the behavior of a system is not enough. Knowing that something fails in simulation is not the same as knowing *why* it failed or whether there are control variables that can be used to eliminate that failure. Once the simulations become high-fidelity enough and complex enough to represent reality, the relationships between the control variables and the eventual system outcomes become obscured. No matter what domains a system falls into, there are a limited number of experts and their time is even further limited. It is cost and risk-prohibitive to ask those experts to sift through gigabytes of simulation settings and outcomes. Therefore, it is desirable to find methods of eliminating that bottleneck: methods that either reduce the amount of data that experts need to examine or methods that can identify the most obvious faults automatically.

For example, consider *Monte Carlo Filtering* as applied at NASA's Robust Software Engineering (RSE) group. The goal of this filtering is to determine which inputs are most likely to determine some portion of the output distributions. The output space is divided into 'good' or 'bad' partitions using some mathematical function of the outputs—for example, a NASA scientist may be most interested in data where the allowable dynamic pressure on the parachutes is exceeded. For this kind of sensitivity analysis, the first step is to run a Monte Carlo experiment sampling the input space, and the second step is to filter the data into two partitions based on the output. In the first step, model inputs are selected at random. In the second step, some sensitivity analysis is then used to predict the variables and ranges in the input space most likely to lead to one of the partitions of the data. A detailed description of Monte Carlo Filtering including a variety of examples and techniques for this type of sensitivity analysis is located in [44].

The field of data mining uses techniques from statistics and artificial intelligence to find small, yet relevant, patterns in large sets of data. The standard practice in this field is to *classify*, to look at an object and make a guess at what category it belongs to. As new evidence is examined, these guesses are refined and improved. When testing a complex hardware system, a scientist might try to classify by assessing whether that particular simulation *succeeded* or *failed*.

Treatment learning [36] focuses on a different goal. It does not try to determine *what is*, it tries to determine *what could be* (and thus, enables the practice of Monte Carlo Filtering). Classifiers read a collection of data and collect statistics that are used to place unseen data into a series of discrete categories (called classes). Treatment learners work in reverse. They take the classification of a piece of evidence (that is, the category that it belongs to) and try to reverse-engineer the statistical evidence that led a classifier to assign the data to a

particular class. If a scientist already knows whether a simulation succeeded or failed, the scientist can use treatment learning to determine why it failed. Treatment learners produce a treatment—a small set of rules that, if imposed, will change the expected classification distribution. By filtering the data for entries that follow the rules set in the treatment, you should be able to identify *why* a particular classification was reached.

Ultimately, classifiers will strive to increase the representational accuracy. They will assess the data and grow a collection of statistical rules with the goal of making more and more accurate categorizations. As a result, if the data is complex, the decision tree output by the classifier will also be complex. Treatment learning instead focuses on minimality: what is the *smallest* rule that can be imposed to cause the *largest* change. Often, these rules may involve only one control variable (e.g. don't launch if the wind speed is greater than 45 knots).

To give a simplified example, consider the case of a rocket intended to place a satellite into orbit. Any number of events could cause this rocket to fail—too long or too short of a burning time, a faulty timing mechanism, or a crack in the outer fairing. A common scenario would be to run a series of simulations prior to any actual launch. Measurements are taken at regular intervals throughout the simulation, noting factors such as the trajectory, external pressure, temperature, etc. Along with these readings comes a classification, whether or not any of the system requirements for the rocket have been violated. After a number of simulation trials, a treatment learner can be focused upon those trials with a “rocket failed” classification. The treatment learner will produce a rule set that states definitively the control variables (temperature too high, fuel level too low) that most often caused a mission-critical failure.

Stated formally, treatment learning is a form of minimal contrast-set association rule learning. The treatments contrast undesirable situations with the desirable ones (represented by weighted classes). Treatment learning, however, is different from other contrast-set methods like STUCCO [6] because of its focus on minimal theories. Conceptually, a treatment learner explores all possible subsets of the attribute ranges looking for good treatments. Such a search is infeasible in practice, so the art of treatment learning lies in quickly pruning unpromising attribute ranges (i.e. ignoring those that, when applied, lead to a class distribution where the target class is in the minority).

In an industrial setting like NASA, critical mission failures will cost thousands, if not millions, of dollars. Therefore, it is absolutely crucial to identify the conditions under which a design will fail as early as possible in the design process, so that design changes may be made before any physical hardware is constructed. Standard optimization techniques can be used to relate the control variables to the outcomes, but many such algorithms rely on continuous variables (all of which must be controllable by the simulator). They are ill-equipped to handle many real-world situations where factors are either discrete or stochastic. Treatment learning has no such restrictions.

While treatment learning (specifically the TAR3 algorithm [24, 25, 28, 46]) has been discussed in prior publications, these algorithms have never been bench-

marked against standard or state-of-the-art optimization algorithms in an industrial setting. This study utilizes the TAR3 and TAR4.1 treatment learners, which operate similarly but score treatments in radically different manners, and compares the quality of the produced treatments against the Simulated Annealing [30,38] and Quasi-Newton [21] optimization methods. Data used in this case study comes from actual simulation trials of projects from NASA’s Robust Software Engineering (RSE) group, as well as real-world data from a bicycle ride. Our goal is to show that, in this real-world industrial setting, treatment learning offers a faster, higher-quality identification of the factors likely to cause a failure in a complex system than traditional optimization techniques.

The results of this study clearly demonstrate that:

- Treatment learners are orders of magnitude faster than standard methods.
- TAR3’s results are more precise than those from standard techniques.
- TAR4.1’s results demonstrate a higher recall, while maintaining a lower false positive rate, than the standard techniques (the Quasi-Newton algorithm also demonstrates a high recall, but at the notable cost of a higher rate of false positives).

2 Data Background

NASA often uses high-fidelity physics simulations early in the design process to verify that flight software will meet the mission requirements. The possible inputs to the simulation can be design parameters like lift coefficients and center of gravity positions; they may also be environmental parameters like the average magnitudes and the standard deviations of wind gusts; they may be flags that indicate the failure of a hardware component at some critical time; or they may be any of a plethora of other parameters that specify the bounds on the acceptable flight envelope. Errors in software design are much less expensive to fix early in the design process, but the design space early in the process is very large. To explore all of the parameter combinations exhaustively is infeasible. However, a very large sampling of the configurations increases the chance that a design error will be caught early, and it allows for the possibility of identifying trends or anomalies within the data.

Manual inspection of these large datasets requires domain expertise, and is likely to focus only on absolute compliance with requirements. For systems this complex many different kinds of failures are possible. For aero-braking scenarios, for instance, representative failures include skipping out of the Earth’s atmosphere instead of re-entering and parachutes failing to open. One common heuristic for these analyses is to push the bounds of the flight envelope until between 10 and 30 percent of all of the attempted cases have failed in one way or another—this kind of analysis allows you to find the margins of failure within the system. In the probable event that failures are identified this early in the process, it is a time-consuming task to determine the cause of those failures; the failures may be associated with environmental factors (e.g. strong sustained wind gusts overwhelm the control system), they may be associated with software errors in

the unit under test (e.g. the gains on the control system are set incorrectly for the nominal case), or they may be associated with software errors in the simulation used to perform the test (that is, a legacy environmental model in the simulation uses different units than expected). A likely first step towards determining the cause of any of the failures is to find the input parameters that the failures are associated with. Even this first step is non-trivial—there are usually hundreds of input parameters associated with each dataset, and those parameters were chosen from thousands of input parameters to the actual simulation.

Two of the datasets used for this paper were generated using the Advanced NASA Technology ARchitecture for Exploration Studies (ANTARES) [1] simulation tool. ANTARES is composed of high-fidelity physics models that are used to study Constellation and the International Space Station. The version of ANTARES used for this work contains over 1 million source lines of code in 15 different programming languages. Each individual simulation, however, touches a relatively small subset of this code. The two ANTARES datasets used for this paper represent the Monte Carlo simulation trials done for a re-entry study and for a launch abort study. The collected variables include environmental parameters, internal simulation values (like random seeds), and spacecraft state specifications—including both continuous and modal information. Both of these datasets had many different possible failure types. The third dataset referenced here was collected during a bicycle ride. The data comes from a bicycle power meter that calculates the power output and collects the following data at 60 Hz: distance traveled, heart rate, speed, wind speed, cadence, elevation, hill slope, and temperature. The power calculation for this particular meter could be very noisy and there was an open question about what measured parameters were most associated with this noise. Note that, because the bicycle data is real-world data, the relationships between the measured variables are not explicit.

Failures for the NASA datasets were defined for several phases of flight corresponding to reentry and launch abort scenarios. Some of the specific failure types included missed landings, aerodynamic angles or body rates that exceeded thresholds, excessively high velocity at impact, and dynamic pressures exceeding tolerances for the parachutes. Any of these failures can lead to loss of life or mission, and are considered unacceptable. In general use at NASA, this tool is used to find the margins to failure from nominal launch conditions over all of the mission-critical failures. In essence, the question is—which of the controllable input parameters need to be most-closely monitored in order to prevent any of these unacceptable failures? Since each individual failure type has its own complicated, usually non-smooth, function of the inputs, the composite of all of the failures creates a non-trivial, almost certainly non-convex, hypersurface that must be searched.

To decrease the amount of time necessary to isolate suspicious inputs, the Robust Software Engineering (RSE) group at Ames utilizes a multi-step process [24, 25, 46, 47] that includes targeting tests with n-factor combinatorial test vectors and sorting the data into clusters with an unsupervised EM algorithm [18] in order to find anomalies and to aid visualization. This tool is

known as Margins Analysis. A key component in the Margins Analysis is the use of a treatment learner to tie behaviors in the dataset to their associated variables and ranges. The treatment learner is used many times throughout the analysis process. It first is used to find variables associated with the overall performance; a penalty is built for each dataset that accounts for all failures and often includes some continuous metric like a target miss distance. The treatment learner is then used to find the parameters associated with each individual type of failure. In practice there are 10's to 100's of different possible failure types identified for each dataset. Finally, the treatment learner is used within a loop to discover the variables associated with each unsupervised cluster. This analysis aids the researcher in understanding the underlying structure of the dataset and can uncover details in the dataset that lead to new requirements.

3 Data Mining and Treatment Learning

3.1 Data Mining

Data mining is a summarization technique that reduces large sets of examples into small understandable patterns using a range of techniques taken from the statistics and artificial intelligence fields [9,22,57]. One way to learn such patterns is to *split* the whole example set into subsets based on some attribute value test. The process then repeats recursively on the subsets. Each splitter value becomes the root of a sub-tree. Splitting stops when either a subset gets so small that further splitting is superfluous, or a subset contains examples with only one classification.

A *good split* decreases the percentage of different classifications in a subset. Such a good split ensures that smaller subtrees will be generated since less further splitting is required to sort out the subsets. Various schemes have been described in the literature for finding good splits. For example, the C4.5 [41] and J4.8 [57] decision tree algorithms uses an information theoretic measure (entropy) to find its splits while the CART [10] decision tree learner uses another measure called the GINA index.

Decision trees can be large and complex. The problem of explaining the performance of these learners to end-users has been explored extensively in the literature (see the review in [51]). Often, some *post-processor* is used to convert an opaque model into a more understandable form:

- Towell and Shavlik generate refined rules from the internal data structures of a neural network [53].
- Quinlan implemented a post-processor to C4.5 called C45rules that generates succinct rules from cumbersome decision tree branches via (a) a greedy pruning algorithm followed by (b) duplicate removal then (c) exploring subsets of the rules relating to the same class [41].
- The first version of our treatment learner (TAR1) was another post-processor to C4.5 that searched for the smallest number of decisions in decision tree branches that (a) pruned the most branches to undesired outcomes while (b) retaining branches leading to desired outcomes [37].

Association rule learners such as APRIORI [42] find attributes that commonly occur together in a training set. In the association $LHS \rightarrow RHS$, no attribute can appear on both sides of the association; i.e. $LHS \cap RHS = \emptyset$. The rule $LHS \rightarrow RHS$ holds in the example set with *confidence* c if $c\%$ of the examples that contain LHS also contain RHS ; i.e. $c = \frac{|LHS \cup RHS| * 100}{|LHS|}$. The rule $LHS \rightarrow RHS$ has *support* s in the example set if $s\%$ of the examples contain $LHS \cup RHS$; i.e. $s = \frac{|LHS \cup RHS| * 100}{|D|}$ where $|D|$ is the number of examples. Association rule learners return rules with high confidence (e.g. $c > 90\%$). The search for associations is often culled via first rejecting associations with low support. Association rule learners can be viewed as generalizations of decision tree learning since the latter restrict the RHS of rules to just one special class attribute while the former can add any number of attributes to the RHS .

An interesting variant of association rule learning is *contrast set learning*. Instead of finding rules that describe the current situation, contrast set learners like STUCCO [6] find rules that differ meaningfully in their distribution across groups. For example, in STUCCO, an analyst could ask “What are the differences between people with Ph.D. and bachelor degrees?”.

Another interesting variant is *weighted class learning*. Standard classifier algorithms such as C4.5 or CART have no concept of a *good* or *bad* class. Such learners therefore can't filter their learned theories to emphasize the location of the *good* classes or *bad* classes. Association rule learners such as MINWAL [11] use weights assigned to each class to focus the learning onto issues that are of particular interest to some audience.

3.2 Treatment Learning

In terms of the above, treatment learning is a weighted contrast set learner that finds rules that associate attribute values with changes to the class distributions. Menzies [37] elaborated the concept of treatment learning while trying to explain the output of data miners to business users. In one domain, he found that users never understood the large theories being generated using any of the above techniques. In an extreme example of this, C4.5 was generating trees with 6000 nodes. The TAR1 prototype (discussed above) achieved some remarkable reductions in that space; specifically, it found constraints on just four variables that pruned away all branches except those leading to the most preferred outcome.

The lesson of TAR1 was that, sometimes, a small minority of constraints can control a much larger space of variables. TAR2 was an experiment in generating tiny theories using this *simplicity assumption*; a small number of factors most influence the outcome. This assumption has two consequences: (1) it implies that the search for an effective model need not be too elaborate; (2) more importantly (in terms of explanation) the generated theory is very small.

The details of treatment learning are discussed below. Before that, it is insightful to ask just how general is this *simplicity assumption* of treatment learning? Empirically, we can state TAR2, TAR3 and TAR4 have been applied to dozens of data sets and in all cases, the small rules generated by this method

have been sufficient to select for a large percentage of the preferred outcomes. Other machine learning researchers have also discovered that simple schemes, using only a subset of the available attributes, can generate effective theories. For example, Holte [26] wrote a machine learner called 1R that was deliberately restricted to learning theories using a single attribute. Surprisingly, he found that learners that use many attributes perform only moderately better than the simpler 1R solution. It should be noted that we don't use the 1R technique - our results show that many of the best treatments will require more than one attribute (though, generally less than four).

In their work on learning simple theories, Kohavi and John [31] *wrapped* their learners in a pre-processor that used a heuristic search to grow subsets of the available attributes from a size of one. At each step in this growth, a learner was used to assess the accuracy of the model learned from the current subset. Subset growth was stopped when the addition of new attributes failed to improve the accuracy. In their experiments, 83% (on average) of the attributes in a domain could be ignored with only a minimal loss of accuracy. Again, our learners do not use this technique—relevant feature selection with wrappers can be prohibitively slow since *each step* of the heuristic search requires a call to the learning algorithm. Under treatment learning's *simplicity assumption*, such an exhaustive search is needlessly complex.

3.3 Example Output

One reason to recommend treatment learning is that its theories are succinct and easy to understand. Figure 1 shows the output of a classifier (the j48 tree learner from the WEKA³ toolkit) as contrasted with the output of a treatment learner (TAR3, as defined in a following section) in Figure 2 on housing data from the city of Boston (a nontrivial dataset with over five hundred examples). The tree output by the j48 classifier is detailed, but difficult to understand. A user must follow the branches of the tree in order to derive conclusions. The treatment given by TAR3 is much easier to understand. It only presents three important pieces of information. First, it tells us the baseline class distribution. It then shows the best treatment—the smallest constraint that most changes the class distribution. In this case, the best houses had 6.7 to 9.78 rooms, and the optimal parent-teacher ratio was 12.6 to 16. Finally, it shows the class distribution if that treatment is applied.

These same treatments can be mapped graphically in a way that is even easier to understand. Figure 3 shows a two-variable treatment for a dataset that represents the time series operation of a bicycle. Each variable or combination of variables within the treatment is given a one or two-dimensional plot. All of the possible values for the noted attributes are plotted, with different shapes to indicate the class. The area inside of the rectangle is the data that the treatment learner is trying to isolate. Lines are plotted around the minimum and maximum values of the ranges given by the treatment. A single glance informs the user that

³ <http://www.cs.waikato.ac.nz/ml/weka/>

```

LSTAT <= 14.98
| RM <= 6.54
| | DIS <= 1.6102
| | | DIS <= 1.358: high (4.0/1.0)
| | | DIS > 1.358
| | | | LSTAT <= 12.67: low (2.0)
| | | | LSTAT > 12.67: medlow (2.0)
| | DIS > 1.6102
| | | TAX <= 222
| | | | CRIM <= 0.06888: medhigh (3.0)
| | | | CRIM > 0.06888: medlow (4.0)
| | | TAX > 222: medlow (199.0/9.0)
RM > 6.54
| RM <= 7.42
| | DIS <= 1.8773: high (4.0/1.0)
| | DIS > 1.8773
| | | PTRATIO <= 19.2
| | | | RM <= 7.007
| | | | | LSTAT <= 5.39
| | | | | | INDUS <= 6.41: medhigh (25.0/1.0)
| | | | | | INDUS > 6.41: medlow (2.0)
| | | | | LSTAT > 5.39
| | | | | | DIS <= 3.9454
| | | | | | | RM <= 6.861
| | | | | | | | INDUS <= 7.87: medhigh (9.0)
| | | | | | | | INDUS > 7.87: medlow (3.0/1.0)
| | | | | | | | RM > 6.861: medlow (3.0)
| | | | | | | | DIS > 3.9454: medlow (14.0/1.0)
| | | | | | | RM > 7.007: medhigh (29.0)
| | | | | | PTRATIO > 19.2: medlow (11.0/1.0)
RM > 7.42
| PTRATIO <= 17.9: high (25.0/1.0)
| PTRATIO > 17.9
| | | AGE <= 43.7: high (2.0)
| | | AGE > 43.7: medhigh (3.0/1.0)
LSTAT > 14.98
| CRIM <= 0.63796
| | INDUS <= 25.65
| | | DIS <= 1.7984: low (5.0/1.0)
| | | DIS > 1.7984: medlow (37.0/2.0)
| | INDUS > 25.65: low (4.0)
CRIM > 0.63796
| RAD <= 4: low (13.0)
| RAD > 4
| | NOX <= 0.655
| | | AGE <= 97.5
| | | | DIS <= 2.2222: low (8.0)
| | | | DIS > 2.2222: medlow (6.0/1.0)
| | | AGE > 97.5: medlow (5.0)
| | NOX > 0.655
| | | CHAS = 0: low (80.0/8.0)
| | | CHAS = 1
| | | | DIS <= 1.7455: low (2.0)
| | | | DIS > 1.7455: medlow (2.0)

```

Fig.1. A decision tree generated by the WEKA's j48 classifier

```

Baseline class distribution:
low:~~~~~ [ 133 - 29%]
medlow:~~~~~ [ 131 - 29%]
medhigh:~~~~~ [ 97 - 21%]
high:~~~~~ [ 94 - 21%]

Treatment:[PTRATIO=[12.6..16)
           RM=[6.7..9.78)]

New class distribution:
low: [ 0 - 0%]
medlow: [ 0 - 0%]
medhigh: [ 1 - 3%]
high:~~~~~ [ 38 - 97%]

```

Fig. 2. A textual treatment generated by the TAR3 learner.

Treatment 1 With Lifting Operator 1.6275

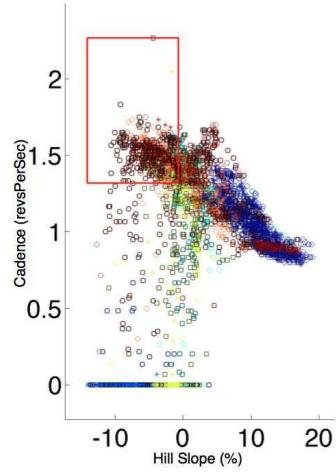


Fig. 3. A graphical representation of a treatment generated by the TAR3 learner.

the area contained within these bold lines is the area of interest. For example, the plot in Figure 3 shows the values for the attributes “Hill Slope” and “Cadence” The treatment suggests limiting the values for these variables to the area enclosed by these lines (about -14% to -1% for Hill Slope and 1.3 to 2.2 for Cadence). This output is easy to read and understand, especially when compared to the complex tree output by a classifier (see Figure 1).

Cognitive scientists have demonstrated that humans are far more likely to use simple models over more complex ones when making decisions [20]. The simpler the output, the easier it is to implement and the more likely that designers will make use of it. This is a key point. For treatment learning to make an effective difference during the design phase of a project, it must produce output that can be understood in a single glance.

3.4 BORE Classification

The raw datasets commonly produced by NASA simulations are not stamped with a basic classification (such as “failed” or “succeeded”). Instead, each simulation trial is assigned a score from a continuous distribution (as assigned by a penalty function unique to each system being simulated). Therefore, before they can be used by the treatment learner, these scores must be sorted into discrete classes. We assign these classes using a process called *BORE*, short for *best or rest*. BORE is a general classification scheme that—given data and a mathematical function involving one or more attributes—categorizes a piece of data as “best” or “rest” according to this function. Commonly, this is not a strict binary split. For example, the scores might be sorted into four quartiles. The top quartile ($0.75 * MAX$ to MAX) will be classified as “best,” while the other three quartiles will be classified as *rest1, rest2, rest3*.

BORE maps the individual factors into a hypercube, which has one dimension for each scored utility. It then normalizes instances scored on the N dimensions from 0 for “worst” to 1 for “best.” The corner of the hypercube at 1,1,... is the *apex* of the cube and represents the desired goal for the system. All of the examples are scored by their normalized Euclidean distance to the apex.

For the purposes of this study, outputs were scored on only one dimension—the scores assigned to each data instance by that system’s penalty function. For each run i of the simulator, the n outputs X_i are normalized to the range 0...1 as follows:

$$N_i = \frac{X_i - \min(X)}{\max(X) - \min(X)}. \quad (1)$$

The Euclidean distance of N_1, N_2, \dots to the ideal position of $N_1=1, N_2=2, \dots$ is then computed and normalized to the range 0..1 as

$$W_i = 1 - \frac{\sqrt{N_1^2 + N_2^2 + \dots}}{\sqrt{n}}, \quad (2)$$

where higher W_i ($0 \leq W_i \leq 1$) correspond to better runs. This means that the W_i can only be improved by increasing all of the utilities. To determine the “best

and “rest values, all of the W_i scores were sorted according to a given threshold. Those that fall above this threshold are classified as “best” and the remainder as “rest” (or, in some cases, multiple divisions of “rest”).

3.5 TAR3

TAR3 (and its predecessor TAR2 [36]) are based on two fundamental concepts—lift and support. The *lift* of a treatment is the change that some decision makes to a set of examples after imposing that decision. TAR3 is given a set of training examples E . Each example $e \in E$ contains a set of attributes, each with a specific value (which have commonly been discretized into a series of ranges). These attributes (and the range their values fall within) are directly mapped to a specific classification (stated formally - $R_i, R_j, \dots \rightarrow C$). The individual class symbols C_1, C_2, \dots are ranked and sorted based on a utility score ($U_1 < U_2 < \dots < U_C$, where U_C is the target class). Within dataset E , these classes occur at certain frequencies (F_1, F_2, \dots, F_C) where $\sum F_i = 1$ (that is, each class occupies a fraction of the overall dataset). A treatment T of size M is a conjunction of attribute value ranges $R_1 \wedge R_2 \dots \wedge R_M$ (these ranges are obtained by discretizing and combining several of the original continuous attribute values). Some subset of the dataset ($e \subseteq E$) is contained within the treatment; that is, if the treatment is used to filter E , $e \subseteq E$ is what will remain. In that subset, the classes occur at frequencies f_1, f_2, \dots, f_C . TAR3 seeks the smallest treatment T which induces the biggest changes in the weighted sum of the utilities multiplied by the frequencies of the classes. This score, the score of $e \subseteq E$ where T has been imposed, is divided by the score of the baseline (dataset E when no treatment has been applied). Formally, the lift is defined as

$$\text{lift} = \frac{\sum_c U_c f_c}{\sum_c U_c F_c}. \quad (3)$$

The classes used for treatment learning are assigned a score $U_1 < U_2 < \dots < U_C$ and the learner uses this to assess the class frequencies resulting from applying a treatment by finding the subset of the inputs that falls within the reduced treatment space. In normal operation, a treatment learner conducts *controller learning*; that is, it finds a treatment which selects for better classes and rejects worse classes. By reversing the scoring function, treatment learning can also select for the worst classes and reject the better classes. This mode is called *monitor learning* because it locates the one thing we should most watch for.

Real-world datasets, especially those from hardware systems, contain some *noise*—incorrect or misleading data caused by accidents and miscalculations. If these noisy examples are perfectly correlated with failing examples, the treatment may become overfitted. An overfitted model may come with a massive lift score, but it does not accurately reflect the general conditions of the search space. To avoid overfitting, learners need to adopt a threshold and reject all treatments that fall on the wrong side of this threshold. We define this threshold as the *minimum best support*.

Given the desired class, the best support is the ratio of the frequency of that class within the treatment subset to the frequency of that class in the overall dataset. To avoid overfitting, TAR3 rejects all treatments with best support lower than a user-defined minimum (usually 0.2). As a result, the only treatments returned by TAR3 will have both a high *lift* and a high *best support*. This is also the reason that TAR3 prefers smaller treatments. The fewer rules adopted, the more evidence that will exist supporting those rules.

TAR3's lift and support calculations can assess the effectiveness of a treatment, but they are not what generates the treatments themselves. A naive treatment learner might attempt to test all subsets of all ranges of all of the attributes. Because a dataset of size N has 2^N possible subsets, this type of brute force attempt is inefficient. The art of a good treatment learner is in finding good heuristics for generating candidate treatments.

The algorithm begins by discretizing every continuous attribute into smaller ranges by sorting their values and dividing them into a set of equally-sized bins. It then assumes the small-treatment effect; that is, it only builds treatments up to a user-defined size. Past research [24, 25] has shown that this threshold should be no higher than four attributes. Note that this is not hard scientific fact, more of a *rule of thumb* —larger treatments are harder for humans to quickly comprehend.

TAR3 will only build treatments from the discretized ranges with a high heuristic value. It determines which ranges to use by first determining the lift score of each attribute's value ranges (that is, the score of the class distribution obtained by filtering for the data instances that contain a value in that particular range for that particular attribute). These individual scores are then sorted and converted into a cumulative probability distribution, as seen in Figure 4. TAR3 randomly selects values from this distribution, meaning that low-scoring ranges are unlikely to be selected. To build a treatment, n (random from 1...max treatment size) ranges are selected and combined. These treatments are then scored and sorted. If no improvement is seen after a certain number of rounds, TAR3 terminates and returns the top treatments.

3.6 TAR4.1

TAR3, while effective at generating informative treatments, is not a very efficient algorithm. It stores all examples from the dataset in RAM and requires three

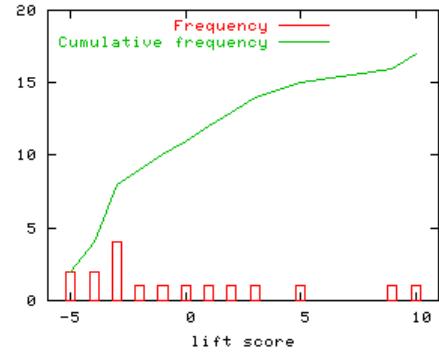


Fig. 4. Probability distribution of individual attribute scores.

scans of the data in order to discretize, build theories, and rank the generated treatments. The TAR4.1 treatment learner was designed to address these inefficiencies. Modeled after the SAWTOOTH [40] incremental Naive Bayes classifier, TAR4.1’s scoring heuristic allows for an improved runtime, lower memory usage, and a better ability to scale to large datasets.

Naive Bayes classifiers offer a relationship between fragments of evidence E_i , a prior probability for a class $P(H)$, and a posteriori probability $P(H|E)$ defined by

$$P(H|E) = \prod_i P(E_i|H) \frac{P(H)}{P(E)}. \quad (4)$$

For numeric features, a features mean μ and standard deviation σ are used in a Gaussian probability function [58]:

$$f(x) = 1/(\sqrt{2\pi}\sigma) e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \quad (5)$$

TAR4.1 still requires two passes through the data, for discretization and for building treatments. These two steps function in exactly the same manner as the corresponding steps in the TAR3 learner. TAR4.1, however, eliminates the final pass by building a scoring cache during the BORE classification stage. As explained previously, examples are placed in a U -dimensional hypercube during classification, with one dimension for each utility. Each example $e \in E$ has a normalized distance $0 \leq D_i \leq 1$ from an *apex*, an area where the best examples reside. When BORE classifies examples into *best* and *rest*, that normalized distance is added as a score, called D_i (the Euclidean distance from 0), to the *down* table and a separate score, $1 - D_i$ (or, the distance from the best), is entered into the *up* table.

When treatments are scored by TAR4.1, the algorithm does a linear-time table lookup instead of scanning the entire dataset. Each range $R_j \in \text{example}_i$ adds scores $down_i$ and up_i to counters $F(R_j|\text{rest})$ and $F(R_j|\text{best})$. These counters are a summation of scores for a range R_j across the dataset, and represent how often data examples containing that range appear in the *best* and *rest*. These summations are then used to compute the following probability and likelihood equations:

$$P(\text{best}) = \frac{\sum_i up_i}{\sum_i up_i + \sum_i down_i}, \quad (6)$$

$$P(\text{rest}) = \frac{\sum_i down_i}{\sum_i up_i + \sum_i down_i}, \quad (7)$$

$$P(R_j|\text{best}) = \frac{F(R_j|\text{best})}{\sum_i up_i}, \quad (8)$$

$$P(R_j|\text{rest}) = \frac{F(R_j|\text{rest})}{\sum_i down_i}, \quad (9)$$

$$L(\text{best}|R_k \wedge R_l \wedge \dots) = \prod_x P(R_x|\text{best}) * P(\text{best}), \quad (10)$$

$$L(rest|R_k \wedge R_l \wedge \dots) = \prod_x P(R_x|rest) * P(rest). \quad (11)$$

TAR4.1 finds the *smallest* treatment T that *maximizes*

$$P(best|T) = \frac{L(best|T)^2}{L(best|T) + L(rest|T)}. \quad (12)$$

Note the squared term in the top of the equation, $L(best|T)^2$. The standard Naive Bayes design assumes independence between all attributes and keeps singleton counts. By not squaring that term, TAR4.1 adds redundant information, which alters the generated probabilities. In effect, it produced treatments with high scores, but without the *support* required by the TAR3 algorithm. By squaring that term, the likelihood that a range appears in an area of top scores, those treatments that lack support are pruned in favor of those that have both a good score and support.

4 Optimization Techniques

Treatment learning is a relatively unexplored field, limiting the number of algorithms that TAR3 and TAR4.1 can be benchmarked against. However, the treatment problem is fundamentally an *optimization* [14] problem. The scoring methods used are simply mathematical objective functions. Therefore, it becomes possible to compare the treatment learning tools against the state-of-the-art techniques used to address optimization problems.

When presented as an optimization problem, the objective function F for TAR3 looks like:

$$\text{maximize } F(x) = \frac{\sum_c U_c f_c(x)}{\sum_c U_c F_c}, \quad (13)$$

where U_c , f_c and F_c are defined as for Equation 3, and x is the attributes and ranges for a suggested treatment. Similarly, the objective function for TAR4.1 is defined as

$$\text{maximize } F(x) = \frac{L(apex|x)^2}{L(apex|x) + L(base|x)}, \quad (14)$$

where the likelihood functions L are the same as those given in Equation 12. For both Equation 13 and Equation 14, x is the treatment—the attributes (discrete values) and their ranges (both continuous and discrete values)—and the size of x will vary based on the number of attributes that the algorithms choose for that particular treatment. Note that since the algorithms used by TAR3 and TAR4.1 do not use gradients there is no requirement for either of the above $F(x)$ to be smooth, and, in practice, both objective functions are highly non-smooth.

Numerous researchers have warned of the difficulties associated with comparing radically different algorithms [23, 27, 56]. Both of these optimization techniques—Simulated Annealing and a Quasi-Newton method—were chosen because they are well-studied, powerful, and ubiquitous approaches that could easily use the

same objective functions as the TAR3 and TAR4.1 treatment learners (thus rendering the results comparable). Furthermore, the algorithms that we use are *unconstrained* (constrained algorithms work towards a pre-determined number of possible solutions while unconstrained methods are allowed to adjust to the goal space). Simulated Annealing has been used to optimize similar design models in our own previous work [19].

Technically, any gradient-based approach (including the Quasi-Newton method used in our experiments) is at a disadvantage when addressing these problems; both the problem and the search space are both a mixture of discrete and continuous variables and the solution space is often locally discontinuous or highly non-linear. Still, researchers often choose to use gradient-based optimization for these problems because, while there is no expectation that they should work, they often (against expectation) do work. When gradient-based methods perform well, they usually do so at a lower computational cost than standard sampling methods (which require heuristics in order to avoid becoming stuck within local minima). Quasi-Newton methods make local approximations to the function, and as a result, they don't require that the function is locally smooth in order to make their next guess. Furthermore, as they are constantly rebuilding the Hessian matrix, they do not have the same tendency towards searching a subspace of a high-dimensional space that Conjugate Gradient methods tend to have. The particular Quasi-Newton method implemented here utilizes heuristics for jumping away from discontinuities in the solution when those discontinuities are discovered. Thus, of all gradient-based methods, this is the most applicable for solving the optimization problem presented by these design simulations.

4.1 Simulated Annealing

Simulated Annealing (SA) is a classic stochastic search algorithm. It was first described in 1953 [38] and refined in 1983 [30]. Fundamentally, SA is a hill climber—it starts in a random location and travels to higher-scoring locations in the immediate neighborhood. Standard hill climbers are prone to becoming stuck in local maxima. To avoid this, Simulated Annealing borrows a heuristic from its namesake, the metallurgy technique “annealing.” In real-world annealing, a material is rapidly heated, then cooled. The heat causes the atoms in the material to rapidly jump around. However, as it cools, the atoms stabilize and solidify—they transition from large jumps to small wiggles. Similarly, Simulated Annealing will jump to sub-optimal solutions at a probability determined by the current state of the temperature function. At first, it will rapidly jump around the search space before finding stability.

The Simulated Annealing algorithm used in this experiment begins by making an initial guess. This guess is a twelve number vector that approximates a four variable treatment. The exact structure of this vector is {ATTRIBUTE, MIN, MAX} repeated four times. The algorithm then tries to solve objective functions corresponding to Equation 13 and Equation 14, except that, in this case x is limited by the algorithm to the structure of the initial guess. Note that Simulated Annealing only requests that the objective functions be smooth

in a limited region near the final solution. We have no such guarantees for our problem, but in practice this is a workable approximation.

The algorithm will continue to operate until a.) the number of tries is exhausted, b.) improvement has not been seen for several rounds, or c.) a certain temperature threshold (a function of the time) is met. The current worth will then be compared to a minimum worth threshold and, if that value is not met, the algorithm will reset. The number of possible resets can be limited, but was not for this experiment.

4.2 Quasi-Newton Optimization

The data mining problem posed in this study requests some subset of the variables and then requests a range for those variables. The best solution to this problem necessarily consists of a combination of integer and (likely nonlinear) continuous values. What's more, the search space is large: there are on the order of hundreds of attributes and on the order of thousands of individual runs. To complicate the problem further, the possible values for each attribute may themselves be continuous or discrete. As a final barrier to traditional optimization techniques, the input variables may not be directly correlated with the output class that is being chosen—either because the appropriate input variables were not selected out of the thousands or (sometimes) tens of thousands of possible input variables or because of some non-determinism in the solution. These factors cause the objective function to be highly non-smooth—there are likely to be many discontinuities and there are no guarantees that the neighborhood of the global solution will be discontinuity-free. Classically, this is the sort of problem that must be solved by direct search optimization methods. However, on a practical basis, optimization techniques derived by assuming that the optimization function is smooth tend to be much more efficient, and often work better-than-expected by utilizing a wide array of numerical “tricks” that work to make the objective function act as if it were more smooth.

The particular optimization technique implemented for this work is a Quasi-Newton method with a BFGS update [49]. This $O(n^2)$ method builds up Hessian information as the iterations proceed, and the approximate Hessian is updated with a rank-one matrix. This constant building of the curvature information tends to avoid the subspace-searching problem that Conjugate Gradient methods can fall into for large problems like those solved within the RSE group. It also avoids the uncertainty that comes with parameter tuning for trust-region methods like Levenberg-Marquardt. However, like all descent methods, the algorithm assumes that the function it is optimizing is essentially continuous.

Mixed integer-nonlinear problems can be solved in several ways [21]. The first such way is a combinatorial approach—solving all possible combinations and choosing the combination that gives the best answer to the objective function. For the types of problems solved in practice within the RSE group at NASA Ames, the combinatorial approach is computationally intractable. One recent example looked for the best 4 attribute treatment out of 128 attributes; the solution of this problem would have required almost 11 million separate optimizations.

Even limiting the problem to choose the one best treatment would still require 128 different optimizations. Instead, as suggested by Gill et al [21], we introduced a new set of variables n_i into the TAR3 objective function where each variable was the percentage likelihood that the attribute i should be included in the treatment. We then introduced the term $\sum_i n_i^{2^{-1}}$ into the objective function to further increase the likelihood of a single discrete choice being made by the optimizer. The final form of this objective function is

$$\text{minimize } F(x) = \sum_i \frac{1}{n_i^2} \frac{-\sum_c U_c f_c(x, n_i, \delta_x(n_i))}{\sum_c U_c F_c}, \quad (15)$$

where U_c and F_c are defined as for Equation 3. The vector x now takes the form $\{n_i, \text{MIN}, \text{MAX}, \dots\}$ where all of the components are continuous and is 3 times the number of attributes N in size. The variable f_c is still the frequency of the class within each subset, but each sum in f_c is now modified by n_i , the percentage likelihood that the attribute i should be included in the treatment. The threshold function δ_x determines whether a particular attribute has enough of a percentage likelihood of being included in the treatment by comparing the value of n_i to a predetermined threshold. The initial values of n_i are set to the inverse of the number of attributes, $1/N$. As the optimizer shrinks some of the values of n_i to maximize the $\sum_i n_i^{2^{-1}}$ term, some of the values of n_i become less than $1/(N + 1)$. When n_i crosses this threshold, the δ_x function removes the attribute from the rule. The optimizer must then choose between increasing the $\sum_i n_i^{2^{-1}}$ term by choosing discrete values (this will also increase the support term) at the cost of reducing the overall worth when attributes are dropped from the rule. Note that Quasi-Newton BFGS methods have some small advantage over some other gradient-based methods in this case because they are making local smooth approximations to the curvature of the function. However, the problem itself, as mentioned before, is inherently non-smooth. In fact, in this case, the objective function can have cliffs even with respect to the continuous ranges because the input to the objective function, the data, is also inherently discrete. As a result, it is very likely that the optimizer will become stuck in local equilibria and that the final solution will be highly dependent on the initial conditions.

While it is possible that we could have improved the performance of any gradient-based method by recasting the objective function to one that was more smooth or (perhaps) by recasting the problem as a constrained minimization problem, no such solution presented itself after a good-faith effort to discover it. This is a common limitation of gradient-based methods. While this limitation exists, it does not prevent researchers from trying to use gradient-based optimization methods for non-smooth problems—when the problem is smooth enough and the initial guess is close enough to the global minimum there can be a significant performance increase over direct search methods like polytope or simulated annealing. Treatment learning is, in essence, a direct search method.

5 Related Work

The work being performed here—choosing the inputs and ranges most likely to lead to some output—can be thought of as a type of sensitivity analysis known as Monte Carlo Filtering [43]. The heavy lifting in most sensitivity analyses of this type is currently being done either with purely linear correlation between the output class and the inputs, a method known as regional sensitivity analysis (RSA) which relies heavily on standard statistical tests such as the Smirnov two-sample test, or it is being done using some sort of regression analysis in which the relationships between the inputs and the outputs are derived from the data [4, 39, 45]. Linear correlations fail in models which are not smooth. For large models, RSA tends to have a very low success rate [50]. Regression analysis builds a polynomial relationship by finding the correlation coefficients between the inputs and outputs. These correlation coefficients are then used to solve the original question—which inputs and their ranges most affect the output—by looking at the magnitudes of the correlation coefficients across the entire range. Regression analysis is computationally expensive and tends to be limited to relatively small numbers of theoretically independent inputs. RSA also tends to assume that relationships between the inputs and outputs are smooth [39, 45].

The types of problems being solved in this paper are non-smooth and of high dimensionality. To overcome the complications involved in finding the correlation coefficients for this sort of problem, we choose in practice to ignore the correlation coefficients altogether and use machine learning techniques that sample the space and solve the original question directly. One example of another existing sensitivity analysis that uses machine learning is the identification of tool faults in the semiconductor industry. Intel uses a technique similar in spirit to the analysis used by NASA’s Robust Software Engineering (RSE) group to find spatial fault patterns on silicon wafers [29]. While the overall goal and appearance of Intel’s method is comparable to RSE’s, the details for every step of the analysis are very different and they do not use treatment learning for their analysis [15, 52, 55]. The fact that parametric testing is being used across widespread applications demonstrates its promise; the massive divergence in the individual components of the technique is evidence that there is still significant research to be done to streamline its use for real-world data.

Gay and Menzies recently conducted a similar treatment learning exercise on NASA Jet Propulsion Lab projects [19]. These projects were encoded in the Defect Detection & Prevention format [13, 17], which is a compiled model representing the requirements of a module, the risks that could compromise those requirements, and mitigations that can allay these risks. Their candidate solution, KEYS2, is based on the theory that a small number of important (“key”) variables control the overall search space. The algorithm generates a large population of treatments and uses a Bayesian ranking mechanism similar to that of the TAR4.1 algorithm (presented later in this paper) to score these treatments. Each round, the top-scoring treatments are used to fix model attributes to specific values. They benchmarked KEYS2 against Simulated Annealing, MaxWalkSat, and an A* search and found that their treatment learner proposed solutions that

completed a higher number of requirements on a lower budget than the other optimization techniques. Additionally, KEYS2 executed the largest models in a fraction of the time that it took for other algorithms.

6 Experiment

Ten Monte Carlo Filtering analyses were run for three different datasets, using five different methods: TAR3, TAR4.1, Simulated Annealing with the TAR3 objective function, Simulated Annealing with the TAR4.1 objective function, and a Quasi-Newton BFGS method with a modified version of TAR3’s scoring function, as shown in Equation 15. As discussed in §2, two of the datasets come from actual analyses performed within the RSE group at NASA Ames. The data in these two sets were gathered during Monte Carlo runs using a high-fidelity physics simulation. One of these datasets represents reentry simulations while the other dataset represents launch abort simulations. The first dataset contains 191 runs worth of 52 different attributes. The second dataset contains 1000 runs worth of 249 attributes. The data from these two projects had complicated penalty functions based on all of the flight requirements—these requirements include metrics like bounds on miss distances, fuel consumption, and the stress on the parachutes. Data with the highest penalty function values are said to have ‘failed’ and data with the lowest penalty function values are considered to be the ‘best’ data. Note that, because of the complicated penalty function, the ‘failed’ data items are likely to have exceeded the allowed values on more than one requirement. An analysis like this gives an overall view of the safest flight conditions given all of the different possible individual mission-critical failures. While the RSE group will often go on to look at individual failure types, the purpose of this experiment was to search for the factors leading to any type of mission-critical failure.

To demonstrate the broad applicability of the technique, we also ran a Monte Carlo Filtering analysis on some data obtained during a bicycle ride. The software that generated the data gave a particularly noisy power measurement. The penalty function used in this dataset evaluated each point in real-time as an individual trial and penalized each run by the noise in the power measurement. The goal was to see which of the other measured parameters was most likely to correspond with the noisy power measurement. This dataset contained 4435 runs over 11 attributes.

7 Results

During each trial, several statistics were collected in order to assess the treatments output by that algorithm. Let $\{A, B, C, \text{ and } D\}$ denote the true negatives, false negatives, false positives, and true positives. From these measures, we can compute certain standard formulas.

$$\text{recall} = \text{probability of detection} = \frac{D}{B + D} \quad (16)$$

$$\text{probability of false alarm} = \frac{C}{A + C} \quad (17)$$

$$\text{precision} = \frac{D}{D + C} \quad (18)$$

For recall and precision, higher values are better. For the probability of false alarm, lower values are desired. Those performance measures, along with the runtime (which should be minimized) were collected for each individual run of each algorithm, then the averages, medians, and standard deviations were saved for each trial. After ten repeats, those statistics were combined and used to create quartile charts. The results for each of our algorithms were combined and ranked using the Mann-Whitney rank-sum test [32].

These quartile charts, sorted by the Mann-Whitney ranks, can be seen in Figure 5, Figure 6, and Figure 7. Note that SA-T3 and SA-T4 refer to the two variants of Simulated Annealing, using the TAR3 and TAR4.1 objective functions respectively. Where used, QN refers to the Quasi-Newton gradient-based method. Also note that only the median values from each individual run were used in the combined results. In each quartile chart, the horizontal lines show the 25 to 75 percentile range, and the black dot represents the median point. The ranks come from the Mann-Whitney rank-sum test at a 95% confidence level. Each rank, from one to five, is statistically different and better than the following rank. If two algorithms have the same rank, their results are not statistically different.

After looking at the results from all three datasets, a clear ranking emerges for each of the collected performance statistics (assessed by the Mann-Whitney test). These rankings are (from best to worst, with parentheses denoting a tie):

- Runtimes: TAR4.1, TAR3, QN, (SA-T4, SA-T3)
- Recall: (TAR4.1, QN), SA-T4, TAR3, SA-T3
- Probability of False Alarm: TAR3, SA-T3, TAR4.1, SA-T4, QN
- Precision: TAR3, SA-T3, TAR4.1, (SA-T4, QN)

While these rankings do not show a single “winner,” they do present a clear victory for the two treatment learning techniques. Either TAR3 or TAR4.1 is ranked at the top in every category, and neither of them are ranked worst in any category. TAR4.1 ties with the Quasi-Newton method in the recall category when one considers the statistical ranking; however, TAR4.1 does tend to show a higher median value.

The Simulated Annealer acted in accordance with its objective function. When using the TAR3 objective function, it tends towards high precision and low recall. Likewise, when using the TAR4.1 objective function, Simulated Annealing returns treatments with higher recall and low precision. In both cases, it performed more weakly than its treatment learner counterpart. Both the weaker

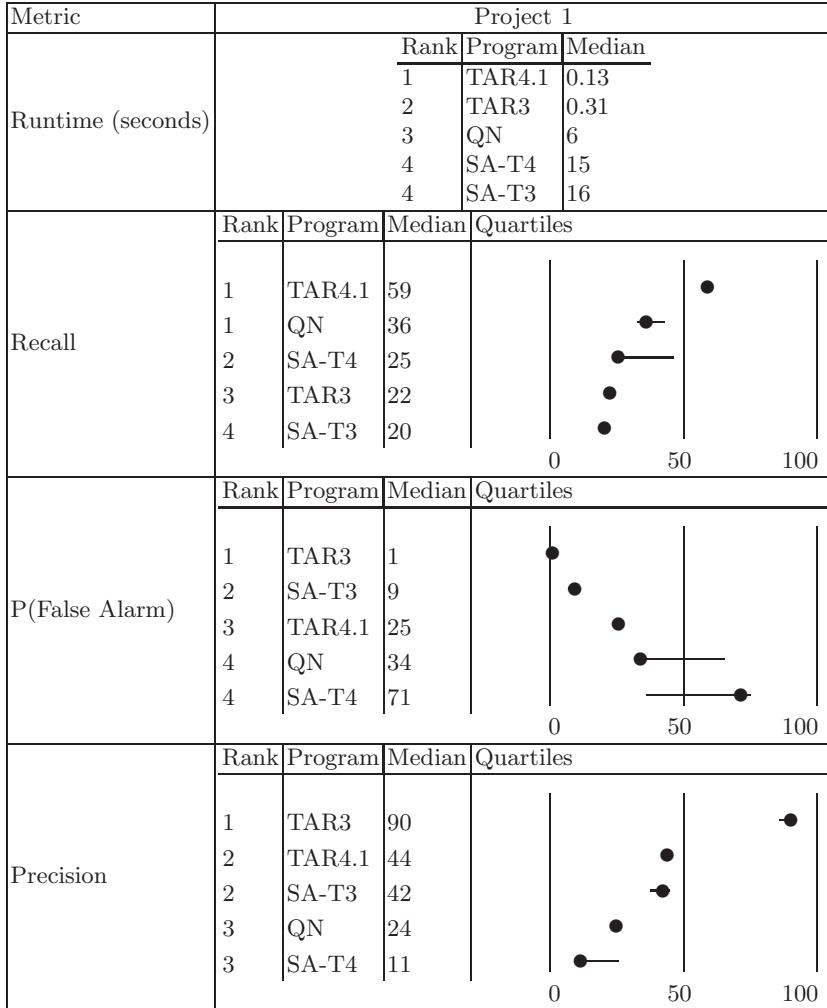


Fig. 5. Results on several criterion, sorted by Mann-Whitney rank, for the RSE Project 1. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles are obtained from summarizing data over ten repeats. Row i is *ranked* higher than row $i - 1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i + 1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired.

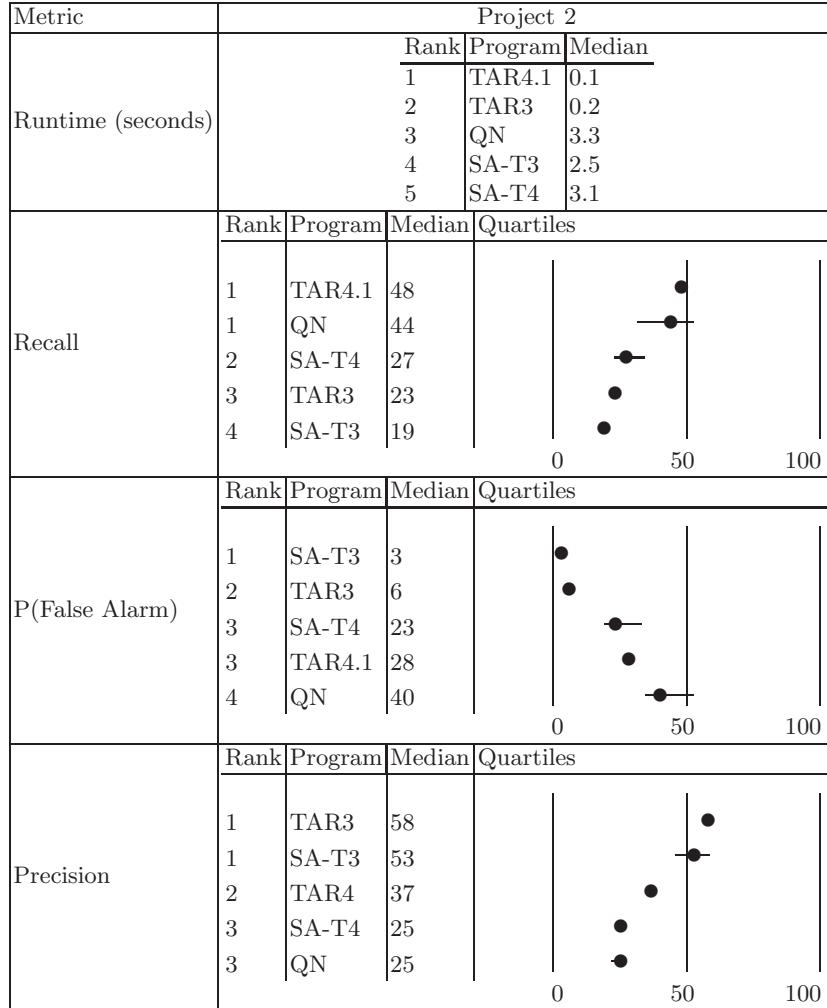


Fig. 6. Results on several criterion, sorted by Mann-Whitney rank, for RSE Project 2. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles are obtained by summarizing data over ten repeats. Row i is *ranked* higher than row $i - 1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i + 1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired.

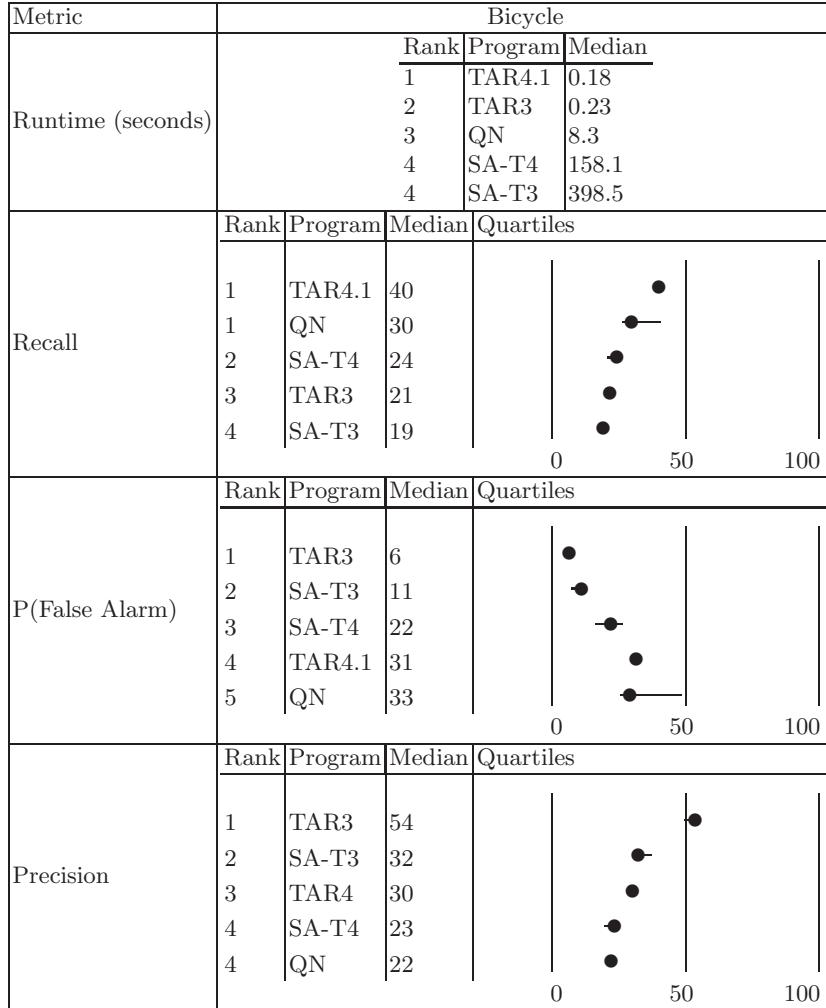


Fig. 7. Results on several criterion, sorted by Mann-Whitney rank, for the bicycle dataset. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles represent data summarized over ten repeats. Row i is *ranked* higher than row $i - 1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i + 1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired.

results and slower runtime can be explained by the very design of Simulated Annealing: because it makes a single initial guess and mutates it, it is unable to try as many combinations as TAR3 or TAR4.1. It must keep trying to make its guess better, and only resets after certain timers expire. It keeps resetting until a certain score threshold is met, which is why it is slower than the other algorithms. If its initial guess is particularly poor, it will never be able to mutate it into something that scores highly. Thus, it will reset until it is able to find a good mutation.

Quasi-Newton performs very well on recall, even tying with TAR4.1 in the rank-sum test. However, it also has the worst probability of selecting false positives. In fact, its false positive rate exceeds its true positive rate on the bicycle dataset. Quasi-Newton is extremely imprecise, it tries to suggest treatments that contain most of the data rather than making any attempt to fit the treatment to the data. As with Simulated Annealing, Quasi-Newton returns results that are highly dependent on the initial conditions. This is because the data has a tendency to be discrete, while Quasi-Newton assumes continuous conditions. In these situations, the algorithm is likely to become stuck in local minima.

Both Simulated Annealing and Quasi-Newton require favorable initial conditions. This weakness is not shared by the treatment learners because of their highly randomized nature. They do not make any single initial guess, and they do not try to manipulate their findings. The use of stochastic search algorithms has been criticized because their results may not be optimal; they may miss potentially powerful treatments because they randomly skip around the space of possible solutions. However, the problem that we are trying to solve is inherently not smooth (much less not convex), which means that gradient-based optimization techniques are also likely to miss the optimal solution. This effect is somewhat mitigated by TAR3 and TAR4.1 because they form treatments from a cumulative probability distribution that favors high-scoring ranges.

8 Discussion

While the results show the advantage for using treatment learning algorithms for these kinds of problems, they do not answer *which* one to use. TAR3 and TAR4.1 excel in different areas, and there is a notable tradeoff between the two. This makes it difficult to clearly recommend one over the other.

TAR3 is extremely specific in its recommendations. It tends to produce treatments that maximize the *lift* calculation while just meeting the support requirement. The result of this are treatments with a low recall value and a very high level of precision. While the recall values are weak, reflecting the lower support, the false alarm rate is nonexistent. This alone may be a reason to favor TAR3's treatments. TAR3 will not give you all of the sources of failure, but it will suggest very few false positives.

TAR4.1, on the other hand, is prone to suggesting treatments with a very high level of support, leading to a higher probability of detection. The problem with TAR4.1's results is that its treatments are not well-fitted to the data, they

do a poor job of filtering out noisy factors or unnecessary information. This results in a much higher false alarm rate than that seen in TAR3's predictions.

The results for both treatment learners when asked to solve the problems as designed by the RSE group, regardless of their respective strengths, tend to be low when compared to the results of standard data mining problems in the literature. Note that, in most cases, every single algorithm used in this experiment returned performance values below 50%. This effect is largely due to the type of problem being solved within the RSE group. In the experiments run in this paper, these treatment learners were being asked to look for *any* critical failure (not just *specific* types of critical failures). This has a tendency to blur the results, as the learners must correlate back to a wide range of inputs, perhaps with disjoint ranges, and there is no guarantee that key inputs for an individual type of failure are in the dataset.

To gain a clearer look at their potential performance, we ran one additional experiment, asking the treatment learning and optimization algorithms to look at a specific failure type in isolation for the second RSE project. In this case, we chose to look at the failures in which a critical slideslip angle limit was exceeded. Those results can be seen in Figure 8. Both TAR3 and TAR4.1 are better able to fit their treatments to the specific problem, resulting in a much lower false alarm rate and almost 100% precision. Interestingly, TAR3 and TAR4.1 performed almost identically, with TAR4.1 maintaining a slightly higher recall and TAR3 a slightly higher precision. TAR3's recall rose significantly, from a 23% median to 36%, while TAR4.1's dropped roughly the same amount, from 48% to 39%.

This experiment in looking at a specific failure type is an even clearer example of why treatment learning techniques should be used. While there was an increase in precision across the board due to the more precise nature of the problem, the performance of the optimization methods was far below the treatment learners. When the TAR3 objective function was used by the simulated annealer, it performed similarly to the actual TAR3. However, its results were poorer and its runtime was slower. Simulated annealing with the TAR4.1 objective function and the Quasi-Newton method showed particularly poor results. For both of these algorithms, the false alarm rate was higher than the median detection rate.

Both the data mining and information retrieval fields have weighed in on the tradeoff between precision and recall on numerous occasions [2,3,12,33,34], never definitively preferring one over the other. For NASA use, both values are highly important. The RSE simulators allow for stochastic parameters, with the wind values being a classic example. Even on a day in which there is no wind, there is a chance for a gust. The model tries to mimic measured parameters for the time of year and day in the launch location. The learners used in these experiments only suggest treatments for parameters we can control or measure, but these uncontrollable stochastic parameters still exist (note, however, that these parameters are likely to be at least loosely correlated with parameters we can control and measure). As a result, the failure boundaries are not well-defined.

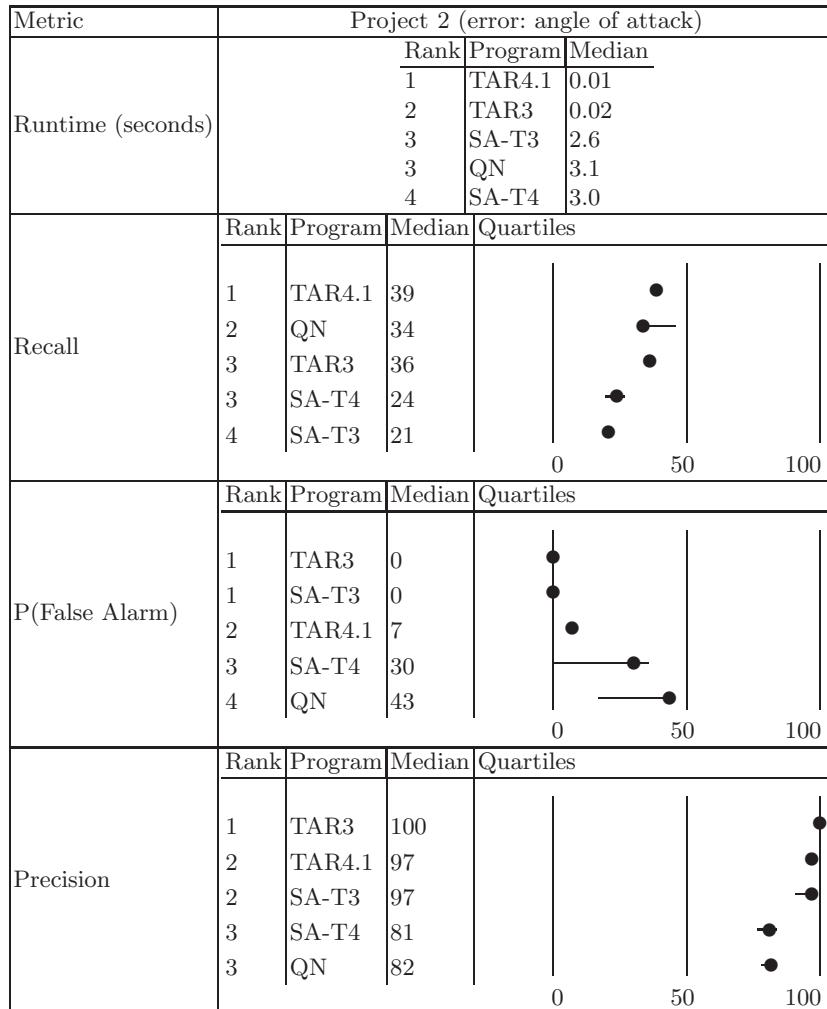


Fig. 8. Results on several criterion, sorted by Mann-Whitney rank, for a specific error type in the second RSE project. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles are obtained by summarizing data over ten repeats. Row i is *ranked* higher than row $i - 1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i + 1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired.

In the particular cases that the RSE group is trying to solve, recall equates to the percentage of failures contained within the predicted rule. Obviously, a user would like recall to be high—you want the produced rules to actually predict the failure. For example, if 95% of all parachute failures happen when the easterly winds exceed some parameter in combination with a center of gravity (cg) within some given range, then you would want to know that restricting the allowable wind velocities and cg on launch day will greatly decrease the odds of that kind of parachute failure (the next goal, at this point, would be to find a rule that eliminates the odds of the other 5% of the failures). However, most treatment-finding algorithms can trivially prevent 100% of launch failures simply by specifying that the launch should never happen at all. If the learner decides that the failures occur when the wind velocities are greater than zero, the produced treatment is essentially stating that no launch is safe.

This is why precision is important in addition to recall—high precision values imply that the treatment doesn't trivially satisfy the constraints. Furthermore, precision does more than just prevent trivial solutions; it gives the engineers definitive trade spaces in which to work. In our previous example, we prevented 95% of parachute failures simply by restricting wind velocities in combination with the cg . The rule could have prevented the same 95% of failures by just restricting the cg placement without considering the wind velocities, but would have done so with worse precision. If restricting the cg of the vehicle becomes too expensive, it may be easier to move the launch date and time to make sure that the the wind velocities are particularly low on the day of launch.

Given the high importance of both precision and recall on NASA simulations, our recommendation would be to favor neither TAR3 or TAR4.1, but to run both and compare the treatments delivered. The runtime advantage that both algorithms have over standard optimization techniques allows for the use of both to quickly explore the treatment space.

9 External Validity

These experiments were conducted at NASA Ames Research Center with assistance from NASA contractors and civil servants. Additionally, two of the primary sources of data were from large-scale NASA project simulations. Therefore, a possible threat to validity exists from the data and environment used for this experiment. The external validity of NASA-based research has been debated by other authors [35, 54]. Basili et al. [5] have argued that conclusions derived from NASA data are relevant to the overall software industry because NASA contractors are obliged to demonstrate an understanding and adherence to modern industrial best practices. These same contractors service numerous industries. For example, Rockwell-Collins builds systems for both defense contractors and civilian aerospace corporations.

However, the work of other authors is not enough to completely dispel the issue of external validity. This is one of the reasons that the bicycle dataset was included in this experiment. The data, recorded during the operation of a

bicycle, was not collected using NASA hardware or at a NASA facility. Despite this separation, the same trends occurred and each of the algorithms performed at a similar efficiency. This replication of trends shows that treatment learning is not a task that has been tuned to NASA data; it, in fact, has applications for both large-scale and small-scale industrial testing.

10 Conclusions and Future Work

Building a large-scale industrial system is a difficult task. It can cost millions of dollars and require months to years of testing. Early simulation makes a large difference, cutting both the cost and time to market [48]. However, this early simulation is of limited value if there is not a way to tell which specific factors led to system failures. Experts are expensive and their time is limited, they cannot waste hours sorting through gigabytes of simulation logs. They need a way to limit the number of possible combinations that need to be examined.

Treatment learning, formally a subset of minimal contrast-set learning, is one method of accomplishing this task. A treatment learner gathers evidence from labeled simulation instances and determines the smallest rule that, when imposed, makes it most likely that a specific outcome will occur.

Although the TAR3 learner has been used in prior publications, it has never been benchmarked against optimization techniques on real-world applications. The goal of this research is to comparatively assess two different treatment learning techniques (TAR3 and TAR4.1) against two standard optimization algorithms (a Quasi-Newton method and Simulated Annealing) on real-world industrial projects. Three sets of data were used, two from large NASA projects and one from the operation of a bicycle. Each algorithm was executed multiple times over each dataset and performance statistics were collected. The results show that treatment learning shows better performance when compared to standard optimization algorithms for these sorts of problems. Both TAR3 and TAR4.1 are orders of magnitude faster than standard techniques. TAR3 demonstrates the lowest false positive rate and highest precision, while TAR4.1 produces the highest recall. Thus demonstrating the superiority of treatment learning over standard optimization algorithms for such design improvement. As both precision and recall are important for such NASA simulations, we favor neither treatment learner; rather, we advocate the use of both TAR3 and TAR4.1 to provide a pool of design suggestions.

The immediate research direction for the treatment learners will center around improvements to the internal heuristics. One idea proposed has been to change the discretization technique. Currently, both TAR3 and TAR4.1 use a simple equal-bin scheme. This naive approach is likely to miss important curves in the data space. Experiments are being conducted with various alternative schemes, including recursive cliff-based methods [16]. Other planned improvements center around optimization of the source code. There are still numerous memory issues that should be addressed and the code should be re-engineered to follow the highest industrial programming standards.

Both of our treatment learners ignore the time-dependency of the recorded data. This is a potential weakness when looking at the failure of a complex system, where the exact cause of a failure may not always be present at the moment where the effect of that failure causes the system to cease functioning. A potential avenue for future research could include incorporating a Markov Model or a Linear Dynamical System into the data processing steps [7] and modifying TAR3 and TAR4.1 with the ability to use these models in their analysis. The treatment learners should consider extreme or mean values over some period of time, whether a particular system mode was ever entered into, and other key events. A goal for this analysis would be to find a way to use sequential data within the machine learning techniques in order to automatically identify interesting time-dependent factors.

11 Acknowledgments

This research was conducted at West Virginia University and the Ames Research Center under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government

References

1. A. Acevedo, J. Arnold, W. Othon, and J. Berndt. ANTARES: Spacecraft simulation for multiple user communities and facilities. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, pages AIAA 2007-6888, 2007.
2. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.
3. G. Antoniol and Y. Gueheneuc. Feature identification: A novel approach and a case study. In *ICSM 2005*, pages 357–366, 2005.
4. P. Austin, P. Grootendorst, and G. Anderson. A comparison of the ability of different propensity score models to balance measured variables between treated and untreated subjects: a monte carlo study. *Statistics in Medicine*, 26:734–753, 2007.
5. V. Basili, F. McGarry, R. Pajerski, and M. Zelkowitz. Lessons learned from 25 years of process improvement: The rise and fall of the NASA software engineering laboratory. In *Proceedings of the 24th International Conference on Software Engineering (ICSE) 2002, Orlando, Florida*, 2002. Available from <http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/83.88.pdf>.
6. S.B. Bay and M.J. Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, 1999. Available from <http://www.ics.uci.edu/~pazzani/Publications/stucco.pdf>.
7. C. Bishop. *Pattern Recognition and Machine Learning*. Springer New York, 2007.
8. B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Trans. on Software Engineering*, 14(10):1462–1477, October 1988.

9. G. Boetticher. An assessment of metric contribution in the construction of a neural network-based effort estimator. In *Second International Workshop on Soft Computing Applied to Software Engineering, Enschede, NL*, 2001. Available from: <http://nas.csl.uh.edu/boetticher/publications.html>.
10. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.
11. C.H. Cai, A.W.C. Fu, C.H. Cheng, and W.W. Kwong. Mining association rules with weighted items. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS 98)*, August 1998. Available from http://www.cse.cuhk.edu.hk/~kdd/assoc_rule/paper.pdf.
12. J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc. The detection and classification of non-functional requirements with application to early aspects. In *RE 2006*, pages 36–45, 2006.
13. S.L. Cornford, M.S. Feather, and K.A. Hicks. DDP a tool for life-cycle risk management. In *IEEE Aerospace Conference, Big Sky, Montana*, pages 441–451, March 2001.
14. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
15. V. Eruhimov, V. Martyanov, and E. Tuv. *Constructing High Dimensional Feature Space for Time Series Classification*, chapter Knowledge Discovery in Databases: PKDD 2007, pages 414–421. Springer Berlin / Heidelberg, 2007.
16. U. Fayyad and I. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.
17. M. Feather, S. Cornford, K. Hicks, J. Kiper, and T. Menzies. Application of a broad-spectrum quantitative requirements model to early-lifecycle decision making. *IEEE Software*, 2008. Available from <http://menzies.us/pdf/08ddp.pdf>.
18. B. Fischer and J. Schumann. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13:483–508, 2003.
19. G. Gay, T. Menzies, O. Jalali, G. Mundy, B. Gilkerson, M. Feather, and J. Kiper. Finding robust solutions in requirements models. *Automated Software Engg.*, 17(1):87–116, 2010.
20. G. Gigerenzer and D.G. Goldstein. Reasoning the fast and frugal way: Models of bounded rationality. *Psychological Review*, pages 650–669, 1996.
21. P.E. Gill, W. Murray, and M.H. Wright. *Practical Optimization*. Academic Press, 1981.
22. D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
23. J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.
24. K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of ANTARES re-entry guidance algorithms using advanced test generation and data analysis. In *9th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2007.
25. K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In *AIAA Aerospace*, 2009.
26. R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

27. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
28. Y. Hu. Treatment learning: Implementation and application. Master’s thesis, Department of Electrical Engineering, University of British Columbia, 2003.
29. H. Jing, R. George, and E. Tuv. *Informatics in Control, Automation and Robotics II*, chapter Contributors to a Signal from an Artificial Contrast, pages 71–78. Springer Berlin / Heidelberg, 2007.
30. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
31. R. Kohavi and G. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.
32. H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 1947. Available on-line at <http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177730491>.
33. A. Marcus and J. Maletic. Recovering documentation-to-source code traceability links using latent semantic indexing. In *Proceedings of the Twenty-Fifth International Conference on Software Engineering*, 2003.
34. T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision. *IEEE Transactions on Software Engineering*, September 2007. <http://menzies.us/pdf/07precision.pdf>.
35. T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from <http://menzies.us/pdf/06learnPredict.pdf>.
36. T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from <http://menzies.us/pdf/03tar2.pdf>.
37. T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In *Proceedings ASE 2000*, 2000. Available from <http://menzies.us/pdf/00ase.pdf>.
38. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys*, 21:1087–1092, 1953.
39. J. Oakley and A. O’Hagan. Probabalistic sensitivity analysis of complex models: a Bayesian approach. *Journal of the Royal Statistical Society B*, 66(3):751–769, 2004.
40. A.S. Orrego. Sawtooth: Learning from huge amounts of data. Master’s thesis, Computer Science, West Virginia University, 2004.
41. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.
42. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD Conference, Washington DC, USA*, 1993. Available from <http://citeseer.nj.nec.com/agrawal93mining.html>.
43. K. Rose, E. Smith, R. Gardner, A. Brenkert, and S. Bartell. Parameter sensitivities, monte carlo filtering, and model forecasting under uncertainty. *Journal of Forecasting*, 10:117–133, 1991.
44. A. Saltelli, K. Chan, and E.M. Scott. *Sensitivity Analysis*. Wiley, 2000.
45. A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global Sensitivity Analysis: The Primer*. Wiley, 2008.

46. J. Schumann, K. Gundy-Burlet, C. Pasareanu, T. Menzies, and A. Barrett. Software V&V support by parametric analysis of large software simulation systems. In *2009 IEEE Aerospace Conference*, 2009.
47. J. Schumann, K. Gundy-Burlet, C. Pasareanu, T. Menzies, and T. Barrett. Tool support for parametric analysis of large software systems. In *Proc. Automated Software Engineering, 23rd IEEE/ACM International Conference*, 2008.
48. S. Sendall and W. Kozacaynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, Sept.-Oct. 2003.
49. C. Sims. Matlab optimization software. QM&RBC Codes, Quantitative Macroeconomics & Real Business Cycles, March 1999.
50. R. Spear, T. Grieb, and N. Shang. Parameter uncertainty and interaction in complex environmental models. *Water Resources Research*, 30(11):3159–3169, 1994.
51. B.J. Taylor and M.A. Darrah. Rule extraction as a formal method for the verification and validation of neural networks. In *IJCNN '05: Proceedings. 2005 IEEE International Joint Conference on Neural Networks*, volume 5, pages 2915–2920, 2005.
52. K. Torkkola and E. Tuv. *Feature Extraction*, chapter Ensembles of Regularized Least Squares Classifiers for High-Dimensional Problems, pages 297–313. Springer Berlin / Heidelberg, 2006.
53. G. Towell and J. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993.
54. B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 2009. Available from <http://menzies.us/pdf/08ccwc.pdf>.
55. E. Tuv, A. Borisov, and K. Torkkola. *Intelligent Data Engineering and Automated Learning – IDEAL 2006*, chapter Best Subset Feature Selection for Massive Mixed-Type Problems, pages 1048–1056. Springer Berlin / Heidelberg, 2006.
56. T. Uribe and M. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *In Proc. of the 1st International Conference on Constraints in Computational Logics*, pages 34–49. Springer-Verlag, 1994.
57. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.
58. I.H. Witten and E. Frank. *Data mining: Practical Machine Learning Tools and Techniques 2nd edition*. Morgan Kaufmann, 2005.

On the Danger of Coverage Directed Test Case Generation

Matt Staats¹, Gregory Gay², Michael Whalen², Mats Heimdahl²

¹ Korea Advanced Institute of Science & Technology, Daejeon, Republic of Korea

² University of Minnesota, Minneapolis MN, USA

staatasm@kaist.ac.kr, greg@greggay.com, {whalen,heimdahl}@cs.umn.edu

Abstract. In the avionics domain, the use of structural coverage criteria is legally required in determining test suite adequacy. With the success of automated test generation tools, it is tempting to use these criteria as the basis for test generation. To more firmly establish the effectiveness of such approaches, we have generated and evaluated test suites to satisfy two coverage criteria using counterexample-based test generation and a random generation approach, contrasted against purely random test suites of equal size.

Our results yield two key conclusions. First, coverage criteria satisfaction alone is a poor indication of test suite effectiveness. Second, the use of structural coverage as a supplement—not a target—for test generation can have a positive impact. These observations points to the dangers inherent in the increase in test automation in critical systems and the need for more research in how coverage criteria, generation approach, and system structure jointly influence test effectiveness.

1 Introduction

In software testing, the need to determine the adequacy of test suites has motivated the development of several test coverage criteria [1]. One such class of criteria are *structural coverage criteria*, which measure test suite adequacy in terms of coverage over the structural elements of the system under test. In the domain of critical systems—particularly in avionics—demonstrating structural coverage is required for certification [2]. In recent years, there has been rapid progress in the creation of tools for automatic directed test generation for structural coverage criteria [3–5]; tools promising to improve coverage and reduce the cost associated with test creation.

In principle, this represents a success for software engineering research: a mandatory—and potentially arduous—engineering task has been automated. However, while there is some evidence that using structural coverage to guide random test generation provides better tests than purely random tests, the effectiveness of test suites automatically generated to satisfy various structural coverage criteria has not been firmly established. In pilot studies, we found that test inputs generated specifically to satisfy three structural coverage criteria via

counterexample-based test generation were *less effective* than random test inputs [6]. Further, we found that reducing larger test suites providing a certain coverage—in our case MC/DC—while maintaining coverage reduced their fault finding significantly, hinting that it is not always wise to build test suites solely to satisfy a coverage criterion [7].

These results are concerning. Given the strong incentives and the ability to automate test generation, it is essential to ask: “*Are test suites generated using automated test generation techniques effective?*” In earlier studies, we used a single system to explore this question. In this paper, we report the results of a study conducted using four production avionics systems from Rockwell Collins Inc. and one example system from NASA. Our study measures the fault finding effectiveness of automatically generated test suites satisfying two structural coverage criteria, branch coverage and Modified Condition Decision Coverage (MC/DC coverage) as compared to randomly generated test suites of the same size. We generate tests using both counterexample-based test generation and a random generation approach. In our study we use mutation analysis [8] to compare the effectiveness of the generated test suites as compared to purely randomly generated test suites of equal size.

Our results show that for both coverage criteria, in our industrial systems, the automatically generated test suites perform *significantly worse* than random test suites of equal size when coupled with an output-only oracle (5.2% to 58.8% fewer faults found). On the other hand, for the NASA example—which was selected specifically because its structure is significantly different from the Rockwell Collins systems—test suites generated to satisfy structural coverage perform dramatically better, finding 16 times as many faults as random test suites of equal size. Furthermore, we found that for most combinations of coverage criteria and case examples, randomly generated test suites reduced while maintaining structural coverage find *more* faults than pure randomly generated test suites of equal size, finding up to 7% more faults.

We draw two key conclusions from these results. First, automatic test generation to satisfy branch or MC/DC coverage does not, for the systems investigated, yield effective tests relative to their size. This in turn indicates that satisfying even a highly rigorous coverage criterion such as MC/DC is a poor indication of test suite effectiveness. Second, the use of branch or MC/DC as a supplement—not a target—for test generation (as Chilensky and Miller recommend in their seminal work on MC/DC [9]) does appear effective.

These results in this paper highlight the need for more research in how the coverage criterion, test generation approach, and the structure of the system under test jointly influence the effectiveness of testing. The increasing availability and use of advanced test-generation tools coupled with our lack of knowledge of their effectiveness is worrisome and careful attention must be paid to their use and acceptance.

2 Related Work

There exist a number of empirical studies comparing structural coverage criteria with random testing, with mixed results. Juristo et al. provide a survey of much of the existing work [10]. With respect to branch coverage, they note that some authors (such as Hutchins et al. [11]) find that it outperforms random testing, while others (such as Frankl and Weiss [12]) discover the opposite. Namin and Andrews have found coverage levels are positively correlated with fault finding effectiveness [13]. Theoretical work comparing the effectiveness of partition testing against random testing yields similarly mixed results. Weyuker and Jeng, and Chen and Yu, indicated that partition testing is not necessarily more effective than random testing [14, 15]. Later theoretical work by Gutjahr [16], however, provides a stronger case for partition testing. Arcuri et al. [17] recently demonstrated that in many scenarios, random testing is more predictable and cost-effective at reaching high levels of structural coverage than previously thought. The authors have also demonstrated that, when cost is taken into account, random testing is often more effective at detecting failures than a popular alternative—adaptive random testing [18].

Most studies concerning automatic test generation for structural coverage criteria are focused on how to generate tests quickly and/or improve coverage [19, 3]. Comparisons of the fault-finding effectiveness of the resulting test suites against other methods of test generation are few. Those that exist apart from our own limited previous work are, to the best of our knowledge, studies in concolic execution [4, 5]. One concolic approach by Majumdar and Sen [20] has even merged random testing with symbolic execution, though their evaluation only focused on two case examples, and did not explore fault finding effectiveness.

Despite the importance of the MC/DC criterion [9, 2], studies of its effectiveness are few. Yu and Lau study several structural coverage criteria, including MC/DC, and find MC/DC is cost effective relative to other criteria [21]. Kandl and Kirner evaluate MC/DC using an example from the automotive domain, and note less than perfect fault finding [22]. Dupuy and Leveson evaluate the MC/DC as a compliment to functional testing, finding that the use of MC/DC improves the quality of tests [23]. None of these studies, however, compare the effectiveness of MC/DC to that of random testing. They therefore do not indicate if test suites satisfying MC/DC are truly effective, or if they are effective merely because MC/DC test suites are generally quite large.

3 Study

Of interest in this paper are two broad classes of approaches: random test generation and directed test generation. In random test generation, tests are randomly generated and then later reduced with respect to the coverage criterion. This approach is useful as a gauge of value of a coverage criterion: if tests randomly generated and reduced with respect to a coverage criterion are more effective than pure randomly generated tests, we can safely conclude the use of the coverage criterion led to the improvement. Unfortunately, evidence demonstrating

this is, at best, mixed for branch coverage [10], and non-existent for MC/DC coverage.

Directed test generation is specifically targeted at satisfying a coverage criterion. Examples include heuristic search methods and approaches based on reachability [19, 3, 4]. Such techniques have advanced to the point where they can be effectively applied to real-world avionics systems. Such approaches are usually slower than random testing, but offer the potential to improve the coverage of the resulting test suites. We aim to determine if using existing directed generation techniques with these criteria results in test suites more effective than randomly generated test suites. Evidence addressing this is sparse and, for branch and MC/DC coverage, absent from the critical systems domain.³

We expect that a test suite satisfying the coverage criterion to be, at a minimum, at least as effective as randomly generated test suites of equal size. Given the central—and mandated—role the coverage criteria play within certain domains (e.g., DO-178B for airborne software [2]), and the resources required to satisfy them, this area requires additional study. We thus seek answers to the following research questions:

RQ1: *Are random test suites reduced to satisfy branch and MC/DC coverage more effective than purely randomly generated test suites of equal size?*

RQ2: *Are test suites directly generated to satisfy branch and MC/DC coverage more effective than randomly generated test suites of equal size?*

We explore two structural coverage criteria: branch coverage, and MC/DC coverage [10, 9]. Branch coverage is commonly used in software testing research and improving branch coverage is a common goal in automatic test generation. MC/DC coverage is a more rigorous coverage criterion based on exercising complex Boolean conditions (such as the ones present in many avionics systems), and is required when testing critical avionics systems. Accordingly, we view it as likely to be an effective criterion—particularly for the class of systems studied in this report.

3.1 Experimental Setup Overview

In this study, we have used four industrial systems developed by Rockwell Collins, and a fifth system created as a case example at NASA. The Rockwell Collins systems were modeled using the Simulink notation and the NASA system using Stateflow [25, 26], and were translated to the Lustre synchronous programming language [27] to take advantage of existing automation. Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window

³ It has been suggested that structural coverage criteria should *only* be used to determine if a test suite has failed to cover functionality in the source code [1, 13]. Nevertheless, test suite adequacy measurement can always be transformed into test suite generation. In mandating that a coverage criterion be used for measurement, it seems inevitable that some testers will opt to perform generation to speed the testing process, and such tools already exist [24].

Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). The NASA system, *Docking_Approach*, describes the behavior of a space shuttle as it docks with the International Space Station.

Information related to these systems is provided in Table 1. We list the number of Simulink subsystems, which are analogous to functions, and the number of blocks, which are analogous to operators. For the NASA example developed in Stateflow, we list the number of states, transitions, and variables.

	# Simulink Subsystems	# Blocks
DWM_1	3,109	11,439
DWM_2	128	429
Vertmax_Batch	396	1,453
Latctl_Batch	120	718
# Stateflow States	# Transitions	# Vars
Docking_Approach	64	104

Table 1. Case Example Information

For each case example, we performed the following steps: (1) mutant generation (described in Section 3.2), (2) random and structural test generation (Section 3.3 and 3.4), and (3) computation of fault finding (Section 3.5).

3.2 Mutant Generation

We have created 250 *mutants* (faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. The mutation operators used in this study are fairly typical and are discussed at length in [28]. They are similar to the operators used by Andrews et al. where they conclude that mutation testing is an adequate proxy for real faults [29].

One risk of mutation testing is *functionally equivalent* mutants—the scenario in which faults exist, but these faults cannot cause a *failure* (an externally visible deviation from correct behavior). This presents a problem when using oracles that consider internal state: we may detect failures that can never propagate to the output. For our study, we used NuSMV to detect and remove functionally equivalent mutants for the four Rockwell Collins systems⁴. This is made possible thanks to our use of synchronous reactive systems—each system is finite, and thus determining equivalence is decidable⁵.

The complexity of determining non-equivalence for the *Docking_Approach* system is, unfortunately, prohibitive, and we only report results using the output-only oracle. Therefore, for every mutant reported as killed in our study, there

⁴ The percentage of mutants removed is very small, 2.8% on average

⁵ Equivalence checking is fairly routine in the hardware side of the synchronous reactive system community; a good introduction can be found in [30].

exists at least one trace that can lead to a user-visible failure, and all fault finding measurements indeed measure faults detected.

3.3 Test Data Generation

We generated a single set of 1,000 random tests for each case example. The tests in this set are between 2 and 10 steps (evenly distributed in the set). For each test step, we randomly selected a valid value for all inputs. As all inputs are scalar, this is trivial. We refer to this as a *random test suite*.

We have directly generated test suites satisfying the branch and MC/DC [10, 31] criteria. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [31].

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying branch and MC/DC coverage [19, 3]. In this approach each coverage obligation is encoded as a temporal logic formula and the model checker can be used to detect a counterexample (test case) illustrating how the coverage obligation can be covered. This approach guarantees that we achieve the maximum possible coverage of the system under test. This guarantee is why we have elected to use counterexample-based test generation, as other directed approaches (such as concolic/SAT-based approaches) do not offer such a straightforward guarantee. We have used the NuSMV model checker in our experiments [32] because we have found that it is efficient and produces tests that are both simple and short [6].

Note that as all of our case examples are modules of larger systems, the tests generated are effectively *unit tests*.

3.4 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [13], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naïvely generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different test suites for each case example using this process.

Per *RQ1*, we also create tests suites satisfying branch and MC/DC coverage by reducing the random test suite with respect to the coverage criteria (that is, the suite is reduced while maintaining the coverage level of the unreduced suite). Again, we produce 50 tests suites satisfying each coverage criterion.

For both counterexample-based test generation and random testing reduced with respect to a criterion, reduction is done using a simple greedy algorithm.

We first determine the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test input from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been removed from the full set of tests.

For each of our existing reduced test suites, we also produce a purely random test suite of equal size using the set of random test data. We measure suite size in terms of the number of total test steps, rather than the number of tests, as random tests are on average longer than tests generated using counterexample-based test generation. These random suites are used as a baseline when evaluating the effectiveness of test suites reduced with respect to coverage criteria. We also generate random test suites of sizes varying from 1 to 1,000. These tests are not part of our analysis, but provide context in our illustrations.

When generating tests suites to satisfy a structural coverage criterion, the suite size can vary from the minimum required to satisfy the coverage criterion (generally unknown) to infinity. Previous work has demonstrated that test suite reduction can have a negative impact on test suite effectiveness [7]. Despite this, we believe the test suite size most likely to be used in practice is one designed to be small—reduced with respect to coverage—rather than large (every test generated in the case of counterexample-based generation or, even more arbitrarily, 1,000 random tests)⁶.

3.5 Computing Fault Finding

In our study, we use *expected value oracles*, which define concrete expected values for each test input. We explore the use of two oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice. Both oracles have been used in previous work, and thus we use both to allow for comparison. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or “killed”).

4 Results and Analysis

We present the fault finding results in Tables 2 and 3, listing for each case example, coverage criterion, test generation method, and oracle: the average fault finding for test suites reduced to satisfy a coverage criterion, next to the average fault finding for random test suites of equal size; the relative change in average fault finding when using the test suite satisfying the coverage criteria versus the random test suite of equal size; and the p-value for the statistical

⁶ One could build a counterexample-based test suite generation tool that, upon generating a test, removes from consideration *all* newly covered obligations, and randomly selects a new uncovered obligation to try to satisfy, repeating until finished. Such a tool would produce test suites equivalent to our reduced test suites, and thus require no reduction; alternatively, we could view such test suites as pre-reduced.

Case Example	Oracle	Counterexample Generation				Random Generation			
		Satisfying Branch	Random of Same Size	% Change	p-val	Satisfying Branch	Random of Same Size	% Change	p-val
Latctl_Batch	MX	217.0	215.8	-0.6%	0.24	238.7	234.4	-1.8%	< 0.01
	OO	82.2	140.3	-41.4%		196.4	189.2	-3.8%	
Vertmax_Batch	MX	211.2	175.3	-20.5%		219.5	209.7	-4.6%	< 0.01
	OO	77.1	101.7	-24.2%		153.5	143.4	-7.0%	
DWM_1	MX	195.1	227.9	-14.4%	< 0.01	230.2	227.1	-1.4%	
	OO	32.1	77.9	-58.8%		79.8	76.9	-3.77%	0.04
DWM_2	MX	202.1	215.5	-6.2%		232.0	225.8	-2.7%	< 0.01
	OO	131.9	174.7	-24.5%		200.3	192.3	-4.2%	
Docking_Approach	OO	38.1	2.0	1805%		2.0	2.0	0.0%	1.0

Table 2. Average number of faults identified, branch coverage criterion. OO = Output-Only, MX = Maximum

Case Example	Oracle	Counterexample Generation				Random Generation			
		Satisfying MCDC	Random of Same Size	% Change	p-val	Satisfying MCDC	Random of Same Size	% Change	p-val
Latctl_Batch	MX	235.0	241.8	-2.8%	< 0.01	241.5	240.3	-0.3%	< 0.01
	OO	194.2	226.7	-14.4%		218.6	214.6	-1.9%	
Vertmax_Batch	MX	248.0	239.3	3.6%	< 0.01	248.0	237.1	-4.6%	< 0.01
	OO	147.0	195.5	-24.8%		204.2	191.4	-6.7%	
DWM_1	MX	210.0	230.4	-12.8%		230.6	229.5	-0.4%	0.048
	OO	44.6	86.6	-48.5%		85.4	86.4	-1.2%	0.6
DWM_2	MX	233.7	232.2	0.7%	0.08	241.9	235.5	-2.7%	< 0.01
	OO	196.2	207.0	-5.2%	< 0.01	222.6	213.5	-4.3%	
Docking_Approach	OO	37.34	2.0	1750%		2.0	2.0	0.0%	1.0

Table 3. Average number of faults identified, MCDC criterion. OO = Output-Only, MX = Maximum

analysis below. Note that negative values for *% Change* indicate the test suites satisfying the coverage criterion are less effective on average than random test suites of equal size.

	Branch Coverage	MCDC Coverage
DWM_1	100.0%	100.0%
DWM_2	100.0%	97.76%
Vertmax_Batch	100.0%	99.4%
Latctl_Batch	100.0%	100.0%
Docking_Approach	58.1%	37.76%

Table 4. Coverage Achieved (of Maximum Coverage) by Randomly Generated Test Suites Reduced to Satisfy Coverage Criteria

The test suites generated using counterexample-based test generation are guaranteed to achieve the maximum achievable coverage, but the randomly generated test suites reduced to satisfy structural coverage criteria are not. We therefore present the coverage achieved by these test suites (with 100% representing the maximum *achievable* coverage) in Table 4.

In Figure 1, we plot, for MC/DC coverage and four case examples, the test suites size and fault finding effectiveness of every test suite generated when using the output-only oracle.⁷ Test suites generated via counterexample-based test generation are shown as pluses, random test suites reduced to satisfy structural coverage criteria are shown as circles, and random test suites of increasing size (including those paired with test suites satisfying coverage criteria) are shown in the line. The line has been smoothed with LOESS smoothing (with a factor

⁷ For reasons of space, plots for branch coverage and the maximum oracle are omitted. Figures for the *Docking_Approach* case example are not very illustrative.

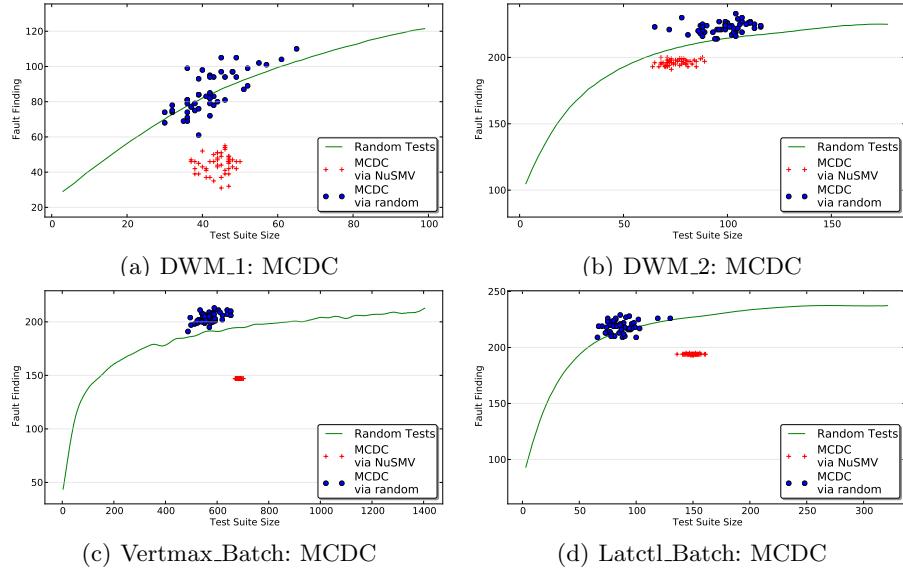


Fig. 1. Faults identified compared to test suite size using NuSMV-generated test suites ('+') , randomly generated test suites reduced to satisfy a coverage criterion ('o') , and pure random test generation (line). Output-only oracles.

of 0.3) to improve the readability of the figure. Note that, while 1,000 random test inputs have been generated, we have only plotted random test suites (i.e., the line) of sizes slightly larger than the test suites satisfying coverage criteria to maintain readability.

4.1 Statistical Analysis

For both *RQ1* and *RQ2*, we are interested in determining if test suites satisfying structural coverage criteria outperform purely random test suites of equal size. We begin by formulating statistical hypotheses H_1 and H_2 :

H_1 : A test suite generated using random test generation to provide structural coverage will find more faults than a pure random test suite of similar size.

H_2 : A test suite generated using counterexample-based test generation to provide structural coverage will find more faults than a random test suite of similar size.
We then formulate the appropriate null hypotheses:

H_{01} : A test suite generated using random test generation to provide structural coverage will find the same number of faults as a pure random test suite of similar size.

H_{02} : A test suite generated using counterexample-based test generation to provide structural coverage will find the same number of faults as a random test suite of similar size.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H_{01} and H_{02} without any assumptions on the distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution assumptions [33]) with 250,000 samples. We pair each test suite

reduced to satisfy a coverage criteria with a purely random test suite of equal size. We then apply this statistical test for each case example, structural coverage criteria, and test oracle with $\alpha = 0.05$.⁸

4.2 Evaluation of RQ1 and RQ2

Based on the p-values less than 0.05 in Tables 2 and 3, we *reject* H_0_1 for nearly all case examples and coverage criteria when using either oracle.⁹ For cases with differences that are statistically significant, test suites reduced to satisfy coverage criteria are more effective than purely randomly generated test suites of equal size; for these combinations, we accept H_1 . Our results confirm that branch and MC/DC coverage can be effective metrics for test suite adequacy within the domain of critical avionics systems: reducing test suites generated via a non-directed approach to satisfy structural coverage criteria is at least not harmful, and in some instances improves test suite effectiveness relative to their size by up to 7.0%. Thus, considering branch and MC/DC coverage when using random test generation generally leads to a positive, albeit slight, improvement in test suite effectiveness.

Based on the p-values less than 0.05 in Tables 2 and 3, we *reject* H_0_2 for all case examples and coverage criteria when using the output-only oracle. However, for all but one case example, test suites generated via counterexample-based test generation are *less* effective than pure random test suites by 5.2% to 58.8%; we therefore conclude that our initial hypothesis H_2 is false¹⁰. Nevertheless, the converse of H_2 —randomly generated test suites are more effective than equally large test suites generated via counterexample-based test generation—is also false, as the *Docking_Approach* example illustrates. For this case example, random testing is effectively useless, finding a mere 2 faults, while tests generated using counterexample-based test generation find 37-38 faults. We discuss the reasons behind, and implications of, this strong dichotomy in Section 5.

When using the maximum oracle, we see that the test suites generated via counterexample-based test generation fare better. In select instances, counterexample-based test suites outperform random test suites of equal size (notably *Vertmax_Batch*), and otherwise close the gap, being less effective than pure random test suites by at most 14.4%. Nevertheless, we note that for most case examples and coverage criteria, random test suites of equal size are still more effective.

⁸ Note that we do not generalize across case examples, oracles or coverage criteria, as the needed statistical assumption, random selection from the population of case examples, oracles, or coverage criteria, is not met. The statistical tests are used only to demonstrate that observed differences are unlikely to have occurred by chance.

⁹ We do not reject H_0_1 for the *DWM_1* case example when using MC/DC coverage and the output-only oracle, nor do we reject H_0_1 for the *Docking_Approach* case example.

¹⁰ In our previous work we found the opposite effect [34].

5 Discussion

Our results indicate that for our systems (1) the use of branch and MC/DC coverage as a *supplement* to random testing generally results in more effective tests suites than random testing alone, and (2) the use of branch and MC/DC coverage as a *target* for directed, automatic test case generation (specifically counterexample-based test generation) results in *less* effective test suites than random testing alone, with decreases of up to 58.8%. This indicates that branch and MC/DC coverage are—by themselves—not good indicators of test suite effectiveness. Given the role of structural coverage criteria in software validation in our domain of interest, we find these results quite troublesome.

The lack of effectiveness for test suites generated via counterexample-based test generation is a result of the formulation of these structural coverage criteria, properties of the case examples, and the behavior of NuSMV. We have previously shown that varying the structure of the program can significantly impact the number of tests required to satisfy the MC/DC coverage criterion [35]. These results were linked partly to *masking* present in the systems—some expressions in the systems can easily be prevented from influencing the outputs. This can reduce the effectiveness of a testing process based on structural coverage criteria, as we can satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

This masking can be a problem; we have found that test inputs generated using counterexample-based generation (including those in this study) tend to be short, and manipulate only a handful of input values, leaving other inputs at default values (`false` or 0) [6]. Such tests tend to exercise the program just enough to satisfy the coverage obligations for which they were generated and do not consider the propagation of values to the outputs. In contrast, random test inputs can vary arbitrarily in length (up to 10 in this study) and vary all input values; such test inputs may be more likely to overcome any masking present in the system.

As highlighted by the *Docking_Approach* example, however, tests generated to satisfy structural coverage criteria can sometimes dramatically outperform random test generation. This example differs from the Rockwell Collins systems chiefly in its structure: large portions of the system’s behavior are activated only when very specific conditions are met. The state space is both deep and contains bottlenecks; exploration of states requires relatively long tests with specific combinations of input values. Thus, random testing is highly unlikely to reach much of the state space. The impact of structure on the effectiveness of random testing can be seen in the coverage of the *Docking_Approach* (only 37.7% of obligations were covered) and is in contrast to the Rockwell Collins systems which—while stateful—have a state space that is shallow and highly interconnected and is, therefore, easier to cover with random testing.

We see two key implications in our results. First, per *RQ1*, using branch and MC/DC coverage as an addition to another non-structure-based testing method—in this case, random testing—can yield improvements (albeit small) in the testing process. These results are similar to those of other authors, for

example, results indicating MC/DC is an effective coverage criterion when used to check the adequacy of manual, requirement-driven test generation [23] and results indicating that reducing randomly generated tests with respect to branch coverage yields improvements over pure random test generation [13]. These results, in conjunction with the results for *RQ2*, reinforce the advice that coverage criteria are best applied after test generation to find areas of the source code that have not been tested. In the case of MC/DC this advice is explicitly stated in regulatory requirements and by experts on the use of the criterion [2, 9].

Second, the dichotomy between the *Docking_Approach* example and the Rockwell Collins systems highlights that while the current methods of determining test suite adequacy in avionics systems are themselves inadequate, some method of determining testing adequacy is needed. While current practice stipulates that coverage criteria should be applied after test generation, in practice, this relies on the honesty of the tester (it is not required in the standard). Therefore, it seems inevitable that at least some practitioners will use automatic test generation to reduce the cost of achieving the required coverage.

Assuming our results generalize, we believe this represents a serious problem. The tools are not at fault: we have asked these tools to produce test inputs satisfying branch and MC/DC coverage, and they have done so admirably; for example, satisfying MC/DC for the *Docking_Approach* example, for which random testing achieves a mere 37.7% of the possible coverage. The problem is that the coverage criteria are simply too weak, which allows for the construction of ineffective tests. We see two possible solutions. First, automatic test generation tools could be improved to avoid pitfalls in using structural coverage criteria. For example, such tools could be encouraged to generate longer test cases increasing the chances that a corrupted internal state would propagate to an observable output (or other monitored variable). Nevertheless, this is a somewhat ad-hoc solution to weak coverage criteria and various tool vendors would provide diverse solutions rendering the coverage criteria themselves useless as certification or quality control tools.

Second, we could improve—or replace—existing structural coverage criteria. Automatic test generation has improved greatly in the last decade, thanks to improvements in search heuristics, SAT solving tools, etc. However, the targets of such tools have not been updated to account for this increase in power. Instead, we continue to use coverage criteria that were originally formulated when manual test generation was the only practical method of ensuring 100% coverage. New and improved coverage metrics are required in order to take full advantage of the improvements in automatic test generation without allowing the generation of inefficient test suites (such as some generated in our study).

6 Threats to Validity

External Validity: Our study has focused on a relatively small number of systems but, nevertheless, we believe the systems are representative of the class of systems in which we are interested, and our results are generalizable to other systems in the avionics domain.

We have used two methods for test generation (random generation and counterexample-based). There are many methods of generating tests and these methods may yield different results. Nevertheless, we have selected methods that we believe are likely to be used in our domain of interest.

For all coverage criteria, we have examined 50 test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [35].

Construct Validity: In our study, we measure fault finding over seeded faults, rather than real faults encountered during development. It is possible real faults would lead to different results. However, Andrews et al. showed that seeded faults leads to conclusions similar to those obtained using real faults [29].

We measure the cost of test suites in terms of the number of steps. Other measurements exist, e.g., the time required to generate and/or execute tests [34]. We chose size to be favorable towards directed test generation. Thus, conclusions concerning the inefficacy of directed test generation are reasonable.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. (Notably, we have avoided statistical inference *across* case examples.)

7 Conclusion and Future Work

The results presented in this paper indicate that coverage directed test generation may not be an effective means of creating tests within the domain of avionics systems, even when using metrics which can improve random test generation. Simple random test generation can yield equivalently sized, but more effective test suites (up to twice as effective in our study). This indicates that adequacy criteria are, for the domain explored, potentially unreliable, and thus, unsuitable, for determining test suite adequacy.

The observations in this paper indicate a need for methods of determining test adequacy that (1) provide a reliable measure of test quality and (2) are better suited as targets for automated techniques. At a minimum, such coverage criteria must, when satisfied, indicate that our test suites are better than simple random test suites of equal size. Such criteria must address the problem *holistically* to account for all factors influencing testing, including the program structure, the nature of the state space of the system under test, the test oracle used, and the method of test generation.

8 Acknowledgements

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, NSF grants CCF-0916583, CNS-0931931, CNS-1035715, and an NSF graduate

research fellowship. Matt Staats was supported by the WCU (World Class University) program under the National Research Foundation of Korea and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007) We would also like to thank our collaborators at Rockwell Collins: Matthew Wilding, Steven Miller, and Darren Cofer. Without their continuing support, this investigation would not have been possible. Thank you!

References

1. H. Zhu and P. Hall, "Test data adequacy measurement," *Software Engineering Journal*, vol. 8, no. 1, pp. 21–29, 1993.
2. RTCA, *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
3. S. Rayadurgam and M. P. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*. IEEE Computer Society, April 2001, pp. 83–91.
4. K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. of the 10th European Software Engineering Conf. / 13th ACM SIGSOFT Int'l. Symp. on Foundations of Software Engineering*. ACM New York, NY, USA, 2005.
5. P. Godefroid, N. Karlund, and K. Sen, "DART: Directed automated random testing," *PLDI05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005.
6. M. P. Heimdahl, G. Devaraj, and R. J. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?" in *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.
7. M. P. Heimdahl and G. Devaraj, "Test-suite reduction for model based tests: Effects on test quality and implications for testing," in *Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.
8. Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, no. 99, p. 1, 2010.
9. J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Engineering Journal*, pp. 193–200, September 1994.
10. N. Juristo, A. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Empirical Software Engineering*, vol. 9, no. 1, pp. 7–44, 2004.
11. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *Proc. of the 16th Int'l Conf. on Software Engineering*. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.
12. P. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Proc. of the Symposium on Testing, Analysis, and Verification*, 1991.
13. A. Namin and J. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proc. of the 18th Int'l Symp. on Software Testing and Analysis*. ACM, 2009.
14. E. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 703–711, 1991.

15. T. Chen and Y. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, 1996.
16. W. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, 1999.
17. A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *ISSTA*, 2010, pp. 219–230.
18. A. Arcuri and L. C. Briand, "Adaptive random testing: An illusion of effectiveness?" in *ISSTA*, 2011.
19. A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *Software Engineering Notes*, vol. 24, no. 6, pp. 146–162, November 1999.
20. R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*, 2007, pp. 416–426.
21. Y. Yu and M. Lau, "A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions," *Journal of Systems and Software*, vol. 79, no. 5, pp. 577–590, 2006.
22. S. Kandl and R. Kirner, "Error detection rate of MC/DC for a case study from the automotive domain," *Software Technologies for Embedded and Ubiquitous Systems*, pp. 131–142, 2011.
23. A. Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the hete-2 satellite software," in *Proc. of the Digital Aviation Systems Conference (DASC)*, Philadelphia, USA, October 2000.
24. "Reactive systems inc. Reactis Product Description," <http://www.reactive-systems.com/index.msp>.
25. "Mathworks Inc. Simulink product web site," <http://www.mathworks.com/products/simulink>.
26. "Mathworks Inc. Stateflow product web site," <http://www.mathworks.com>.
27. N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
28. A. Rajan, M. Whalen, M. Staats, and M. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," in *Proc. of the 10th Int'l Conf. on Formal Methods and Software Engineering*. Springer, 2008, pp. 86–104.
29. J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pp. 402–411, 2005.
30. C. Van Eijk, "Sequential equivalence checking based on structural similarities," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, vol. 19, no. 7, pp. 814–819, 2002.
31. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Office of Aviation Research, Washington, D.C., Tech. Rep. DOT/FAA/AR-01/18, April 2001.
32. "The NuSMV Toolset," 2005, available at <http://nusmv.irst.itc.it/>.
33. R. Fisher, *The Design of Experiment*. New York: Hafner, 1935.
34. G. Devaraj, M. Heimdahl, and D. Liang, "Coverage-directed test generation with model checkers: Challenges and opportunities," *Computer Software and Applications Conference, Annual International*, vol. 1, pp. 455–462, 2005.
35. A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proc. of the 30th Int'l Conference on Software engineering*. ACM New York, NY, USA, 2008, pp. 161–170.

Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing

Matt Staats

Division of Web Science Technology

Korea Advanced Institute of Science & Technology

staatsm@gmail.com

Gregory Gay, Mats P.E. Heimdahl

Department of Computer Science and Engineering

University of Minnesota

greg@greggay.com, heimdahl@cs.umn.edu

Abstract—In testing, the test oracle is the artifact that determines whether an application under test executes correctly. The choice of test oracle can significantly impact the effectiveness of the testing process. However, despite the prevalence of tools that support the selection of test inputs, little work exists for supporting oracle creation.

In this work, we propose a method of supporting test oracle creation. This method automatically selects the *oracle data*—the set of variables monitored during testing—for expected value test oracles. This approach is based on the use of mutation analysis to rank variables in terms of fault-finding effectiveness, thus automating the selection of the oracle data. Experiments over four industrial examples demonstrate that our method may be a cost-effective approach for producing small, effective oracle data, with fault finding improvements over current industrial best practice of up to 145.8% observed.

Keywords-testing, test oracles, oracle data, oracle selection, verification

I. INTRODUCTION

There are two key artifacts to consider when testing software: the *test data*—the inputs given to the application under test—and the *test oracle*, which determines if the application executes correctly. Substantial research has focused on supporting the creation of effective test inputs but relatively little attention has been paid to the creation of oracles. We are interested in the development of automated tools that support the creation of part or all of a test oracle.

Of particular interest in our work are *expected value test oracles* [27]. Consider the following testing process for a software system: (1) the tester selects test inputs using some criterion (structural coverage, random testing, engineering judgement, etc.), potentially employing automated test generation techniques; (2) the tester then defines concrete, expected values for these inputs for one or more variables (internal state variables or output variables) in the program, thus creating an expected value test oracle. The set of variables for which expected values are defined is termed the *oracle data set* [28]. Our focus is on critical systems; past experience with industrial practitioners indicates that expected value test oracles are commonly used in testing such systems.

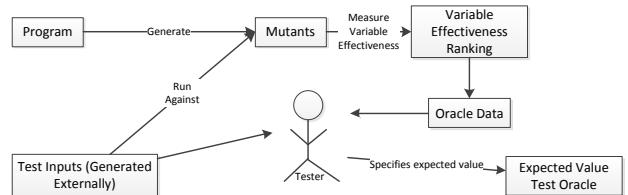


Figure 1. Supporting Expected Value Test Oracle Creation

We present an approach to automatically select the oracle data for use by expected value oracles. Our motivation is twofold. First, existing research indicates that the choice of oracle has a significant impact on the effectiveness of the testing process [32], [3], [28], [27]. However, current widespread practice is to define expected values for only the outputs, a practice that can be suboptimal. Second, manually defining expected values is a potentially time-consuming and consequently, expensive process. Thus in practice the naive solution of “monitor everything” is not feasible because of the high cost of creating the oracle. Even in situations where an executable software specification can be used as an oracle, e.g., in some model-based development scenarios, limited visibility into embedded systems or the high cost of logging often make it highly desirable to have the oracle observe only a small subset of all variables.

Our goal is to support the creation of test oracles by selecting oracle data such that the fault finding potential of the testing process is maximized with respect to the cost. We illustrate our proposed approach in Figure 1. First, we generate a collection of mutants from the system under test. Second, the test suite (generated externally) is run against the mutants using the original system as the oracle (fully automated back-to-back testing). Third, we measure how often each variable in the system reveals a fault in a mutant and—based on this information—we rank variable effectiveness in terms of fault finding. Finally, we estimate—based on this ranking—which variables to include in the oracle data for an expected value oracle. The underlying hypothesis is that, as with mutation-based test data selection, oracle data that is likely to reveal faults in the mutants will also be likely to reveal faults in the actual system under test. This oracle data selection process is completely automated and requires no manual intervention. Once this

oracle data is selected, the tester defines expected values for each element of the oracle data. Testing then commences with a—hopefully—small and highly effective oracle.

We hypothesize that this oracle creation support process will produce an oracle data set that is more effective at fault finding than an oracle data set selected with a standard approach, such as the monitoring of all output variables. To evaluate our hypothesis, we have evaluated our approach using four commercial sub-systems from the civil avionics domain. To our knowledge, there are no alternative approaches to oracle data selection discussed in the literature. To provide an evaluation in the absence of related work, we perform a comparison against two common *baseline approaches*: (1) current practice, favoring the outputs of the system under test as oracle data, and (2) simple random selection of the oracle data set. In addition, we also compare to an idealized scenario where the seeded faults in the mutants are identical to the faults in the actual system under test; thus, providing an estimate of the maximum fault finding effectiveness we could hope to achieve with any oracle data selection support method.

Our results indicate that our approach is generally successful with respect to the baseline approaches, performing as well or better in almost all scenarios, with best-case improvement of 145.8%, and consistent improvements in the 5-30% range. Furthermore, we have also found that our approach often performs almost as well as the estimated maximum. We therefore conclude that our approach may be a cost effective method of supporting the creation of an oracle data set.

II. BACKGROUND & RELATED WORK

In software testing, a *test oracle* is the artifact used to determine whether the software is working correctly [25]. We are interested in test oracles defined as expected values; test oracles that, for each test input, specify concrete values the system is expected to produce for one or more variables (internal state and/or output). During testing, the oracle compares the actual values against the expected values¹. We term such oracles *expected value oracles*. In our experience with industrial partners such test oracles are commonly used when testing critical software systems.

Our goal is to support the selection of the *oracle data* [27] or *oracle data set*². The oracle data is the subset of internal state variables and outputs for which expected values are specified. For example, an oracle may specify expected values for all of the outputs; we term this an *output-only* oracle. This type of oracle appears to be the most common expected value oracle used in testing critical systems. Other types of test oracles include *output-base* oracles, whose

¹For our case examples, all variables are *scalar* and cannot be a heap object or pointers. Thus, comparison is straightforward.

²Oracle data roughly corresponds to Richardson et al.’s concept of *oracle information* [25].

oracle data contain all the outputs, followed by some number of internal states, and *maximum* oracles, whose oracle data contain *all* of the outputs and internal state variables. In the remainder of this paper, we will refer to the *size* of an oracle, where *size* refers to the number of variables used in the oracle data set.

Larger oracle data sets are generally more powerful than smaller oracle data sets [27], [32], [3]. This phenomenon is due to *fault propagation*—faults leading to incorrect states do not always propagate and manifest themselves as failures in a variable in the oracle data set. By using larger oracle data, we can improve the likelihood we will detect faults.

While the maximum oracle is always (provably) the most effective expected value test oracle, it is often prohibitively expensive to use. This is the case when (1) expected values must be manually specified (consulting requirements documents as needed), a highly labor intensive process, or (2) when the cost of monitoring a large oracle data set is prohibitive, e.g., when testing embedded software on the target platform. In current practice, testers must manually select the oracle data set without the aid of automation; this process is naturally dependent on the skill of the tester. We therefore wish to automatically construct small, but effective oracle data sets for expected value oracles.

Work on test oracles often consists of methods of constructing test oracles from other software engineering artifacts [2]. In our work, we are not constructing the entire test oracle; rather, we are identifying an effective oracle data set for an expected value oracle. We are not aware of any work proposing or evaluating alternative methods of selecting the oracle data set.

Voas and Miller suggest that the PIE approach, which—like our work—relies on a form of mutation analysis, could be used to select internal variables for monitoring [31], though evaluation of this idea is lacking. More recent work has demonstrated how test oracle selection can impact the effectiveness of testing, indicating a need for effective oracle selection techniques [28], [27]. Xie and Memon explore methods of constructing test oracles specifically for GUI systems, yielding several recommendations [32]. Briand et al. demonstrate for object-oriented systems that expected value oracles outperform state-based invariants, with the former detecting faults missed by the latter [3]. Several tools exist for automatically generating invariant-based test oracles for use in regression testing, including Eclat [20], DiffGen [29], and work by Evans and Savoy [8]. Note that these tools do not quantify the potential effectiveness of invariants, and thus—unlike our approach—no prioritization can be performed.

Fraser and Zeller use mutation testing to generate both test inputs and test oracles [10] for Java programs. The test inputs are generated first, followed by generation of post-conditions capable of distinguishing the mutants from the program with respect to the test inputs. Unlike our work,

the end result of their approach is a *complete* test case, with inputs paired with expected results (in this case assertions). Such tests, being generated from the program under test, are guaranteed to pass (excepting program crashes). Accordingly, the role of the user in their approach differs: the user must decide, for each input and assertion pair, if the program is working correctly. Thus, in some sense their approach is more akin to invariant generation than traditional software testing. The most recent version of this work attempts to parameterize (generalize) the result of their approach to simplify the user’s task [11]. However, this creates the possibility of producing *false positives*, where a resulting parameterized input/assertion can indicate faults when none exist (5.9% to 12.7% during evaluation), further changing the user’s task. With respect to evaluation, no comparisons against baseline methods of automated oracle selection are performed; developer tests+assertions are compared against, but the cost, i.e., number of developer tests/assertions, is not controlled, and thus relative cost-effectiveness cannot accurately be assessed.

Our work chiefly differs in that we are trying to *support* creation of a test oracle, rather than completely automate it. The domain and type of oracles generated also differ, as does the nature of the test inputs used in our evaluation.

Mutation analysis was originally introduced as a method of evaluating the effectiveness of test input selection methods [7]. Subsequent work also explored the use of mutation analysis as a means of generating tests. In [16], Jia and Harman summarize both the technical innovations and applications related to mutation testing. To the best of our knowledge, only Fraser and Zeller have leveraged this to address test oracles.

III. ORACLE DATA SELECTION

Our approach for selecting the oracle data set is based on the use of mutation testing for selecting test inputs [16]. In mutation testing, a large set of programs, termed *mutants*, are created by seeding various faults (either automatically or by hand) into a system. A test input capable of distinguishing the mutant from the original program is said to *kill* the mutant. In our work, we adopt this approach for oracle creation support. Rather than generate test inputs that kill the mutants, however, we generate an oracle data set that—when used in an expected value oracle, and with a fixed set of test inputs—kills the mutants. To accomplish this, we perform the following basic steps:

- 1) Generate several mutants, called the *training set*, from our system under test.
- 2) Run test inputs, provided by the tester, over the training set and the original system, determining which variables distinguish each mutant from the original system.
- 3) Process this information to create a list of variables ordered in terms of apparent fault finding effectiveness,

the the *variable ranking*.

- 4) Examine this ranking, along with the mutants and test inputs, to estimate (as X) how large the oracle data set should be. Alternatively, the tester can specify X based on the testing budget.
- 5) Select the top X variables in the ranking as the oracle data.

While conceptually simple, there are several relevant parameters that can be varied for each step. The following subsections will outline these parameters, as well as the rationale for the decisions that we have made for each step.

A. Mutant Generation and Test Input Source

During mutation testing, *mutants* are created from an implementation of a system by introducing a single fault into the program. Each fault is created by either inserting a new operator into the system or by replacing an operator or variable with a different operator or variable. This mutation generation is designed such that all mutants produced are both syntactically and semantically valid: no mutant will “crash” the system under test. The mutation testing operators used in this experiment are similar to those used by other researchers, for example, arithmetic, relational, and boolean operator replacement, boolean variable negation, constant replacement, and delay introduction (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value). A detailed description is available in [21].

The type of faults used to create mutants may impact the effectiveness of the selected oracle data when used to test the actual system under test. Note that the type of mutants used in the evaluation in this report are similar to those used in [1] where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments. This offers evidence that our use of mutation testing will support the creation of oracles useful for real systems.

Our approach can be used with any set of test inputs. In this work, we assume the tester is equipped with an existing set of test inputs and wishes to determine what oracle data is likely to be effective with said test inputs. This assumption allows the numerous existing methods of test input selection to be paired with our approach for oracle data selection. Furthermore, this scenario is the most likely within our domain of interest.

B. Variable Ranking

Once we have generated mutants, we then run the test inputs over both the mutants and the original program. During execution of these inputs, we collect the values for every variable, at every step of every test (i.e., the complete state at every point of the execution). We term this resulting data as the *trace data*. A variable is said to have detected a fault when the variable value in the original “correct”

system differs from the variable value produced by a mutant, for some test. We track which mutants are killed by which variables. Note that duplicate detections, in which a variables detects the same mutant in multiple tests, are not counted.

Once we have computed this information, we can produce a set of variables ranked according to effectiveness. One possible method of producing this ranking is simply to order variables by the number of mutants killed. However, the effectiveness of individual variables can be highly correlated. For example, when a variable v_a is computed using the value of a variable v_b : if v_b is incorrect for some test input, it is highly probable that v_a is also incorrect. Thus, while v_a and v_b may be highly effective when used in the oracle data set, the combination of both is likely to be only marginally more effective than the use of either alone.

To avoid selecting a set of variables that are individually effective, but ineffective as a group, we have elected to use a greedy algorithm for solving the *set covering problem* [6] to produce a ranked set of variables. In the set covering problem, we are given several sets with some elements potentially shared between the sets. Our goal is to select the minimum set of elements such that one element from each set has been selected. In this problem each set represents a mutant, and each element of the set is a variable capable of detecting the mutant for at least one of the test inputs. Calculating the smallest possible set covering is an NP-complete problem [12]. Consequently, we employ a well-known effective greedy algorithm to solve the problem [5]: (1) select the element covering the largest number of sets, (2) remove from consideration all sets covered by said element, and (3) repeat until all sets are covered.

In our case, each element removed corresponds to a variable. These variables are placed in a ranking in the order they were removed. The resulting ranking can then be used to produce an oracle data set of size n —simply select the top n elements of the list.

C. Estimating Useful Oracle Data Size

Once we have a calculated the ranked list of variables, we can select an oracle data set of size 1, 2, etc. up to the maximum number of variables in the system, with the choice of size likely made according to some testing budget. In some scenarios, the tester may have little guidance as to the appropriate size of the oracle data. In such a scenario, we would like to offer a recommendation to the tester; we would like to select an oracle data set such that the size of the set is justifiable, i.e., not so small that potentially useful variables are omitted, and not so large that a significant number of variables not adding value are selected.

To accomplish this, we determine the fault finding effectiveness of oracle data sets of size 1, 2, 3, etc. over our training set of mutants. The fault finding effectiveness of these oracles will increase with the oracle's size, but the increases will diminish as the oracle size increases. Consequently, it is

possible to define a natural cutoff point for recommending an oracle size; if the fault finding improvement between an oracle of size n and size $n + 1$ is less than some threshold, we recommend an oracle of size n .

In practice, establishing a threshold will depend on factors specific to the testing process. Therefore, in our evaluation, we explore two potential thresholds: 5% and 2.5%.

IV. EVALUATION

We wish to evaluate if our oracle creation support approach yields effective oracle data sets. Preferably, we would directly compare against existing algorithms for selecting oracle data; however, to the best of our knowledge, no such methods exist. We therefore compare our technique against two potential *baseline approaches* for oracle data set selection, detailed later, as well as an idealized version of our own approach. We wish to explore the following research questions:

Question 1: *Is our approach more effective in practice than current baseline approaches to oracle data selection?*

Question 2: *What is the maximum potential effectiveness of the mutation-based approach, and how effective is the realistic application of our approach in comparison?*

Question 3: *How does the choice of test input data impact the effectiveness of our approach?*

A. Experimental Setup Overview

In this research, we have used four industrial systems developed by Rockwell Collins engineers. All four systems were modelled using the Simulink notation from Mathworks Inc. [18] and were automatically translated into the Lustre synchronous programming language [14] in order to take advantage of existing automation³.

Two systems, *DWM1* and *DWM2*, represent distinct portions of a Display Window Manager (DWM) for a commercial cockpit display system. Two other systems, *Vertmax_Batch* and *Latctl_Batch*, describe the vertical and lateral mode logic for a Flight Guidance System.

	Subsyst.s	Blocks	Outputs	Internals
DWM1	128	429	9	115
DWM2	3109	11,439	7	569
Vertmax_Batch	396	1,453	2	415
Latctl_Batch	120	718	1	128

Table I
CASE EXAMPLE INFORMATION

All four systems represent sizable, operational systems. Information related to these systems is provided in Table I.

For each case example, we performed the following steps:

- (1) generated structural test input suites (detailed in Section IV-B below),
- (2) generated training sets (Section IV-C),
- (3) generated evaluation sets (Section IV-C),
- (4) ran test suite

³In practice, Lustre would then be automatically translated to C code, but this is a syntactic transformation. Our usage of Lustre is purely for convenience; if applied to C the results here would be identical.

on mutants (Section IV-E), (5) generated oracle rankings (Section IV-D), and (6) assessed fault finding ability of each oracle and test suite combination using evaluation sets (Section IV-E).

B. Test Suite Generation

As noted previously, we assume the tester has an existing set of test inputs. Consequently, our approach can be used with any method of test input selection. Since we are studying the effectiveness using avionics systems, two structural coverage criteria are likely to be employed: branch coverage and MC/DC coverage [26]. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is frequently used within the avionics community [4].

We use a counterexample-based test generation approach to generate tests satisfying these coverage criteria [13], [24]. This approach is guaranteed to generate a test suite that achieves the maximum possible coverage. We have used the NuSMV model checker in our experiments [19] for its efficiency and we have found the tests produced to be both simple and short [15].

Counterexample-based test generation results in a separate test for each coverage obligation. This results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. Such an unnecessarily large test suite is unlikely to be used in practice. We therefore reduce each generated test suite while maintaining coverage. We use a simple randomized greedy algorithm. We begin by determining the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test input from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been removed from the full set of tests. We produce 10 different test suites for each case example/coverage criterion to control for the impact of randomization.

C. Mutant Generation

For each case example, 250 mutants are created by introducing a single fault into the correct implementation. We then produce 10 *training sets* by randomly selecting 10 subsets of 125 mutants. For each training set, the 125 mutants *not* selected for the training set are used to construct an evaluation set. We then remove functionally equivalent mutants from each evaluation set, resulting in a reduction of 0-9 mutants.

We remove *functionally equivalent* mutants using the NuSMV model checker. This is possible due to the nature of the systems in our study—each system is finite, and thus determining equivalence is decidable, and in practice fast⁴. The removal of equivalent mutants is done for the evaluation sets as it represents a potential threat to validity

⁴Equivalence checking is fairly routine in the hardware domain; a good introduction can be found in [30].

in our evaluation; our method is effective only if it detects faults which can impact the external behavior. Note that the removal of equivalent mutants for training sets is possible for the case examples examined (and would likely improve the effectiveness of our approach), but is *not* done as it may not be practical for sufficiently large systems and is generally undecidable for systems that are not finite.

D. Oracle Data Set Generation

For each set of mutants generated (training sets and evaluation sets), we generated an oracle ranking using the approach described in Section III. The rankings produced from training sets reflect how our approach would be used in practice; these sets are used in evaluating our research questions. The rankings produced from evaluation sets represent an idealized testing scenario, one in which we already know the faults we are attempting to detect. Rankings generated from the evaluations sets, termed *idealized rankings*, hint at the maximum potential effectiveness of our approach and the maximum potential effectiveness of oracle data selection in general, and are used to address Question 2.

We limited each ranking to contain m variables (where m is 10 or twice the number of output variables, whichever was larger) since test oracles significantly larger than output-only oracles were deemed unlikely to be used in practice, and using larger oracles makes visualizing the more relevant and thus interesting oracle sizes difficult.

To answer Questions 1 and 3, we compare against two baseline rankings. First, the *output-base* approach creates rankings by first randomly selecting output variables, and then randomly selecting internal state variables. Thus, the output-base rankings always list the outputs first (i.e., more highly ranked) followed by the randomly-selected internal state variables. This ranking was chosen to reflect what appears to be the current approach to oracle data selection: select the outputs first, and if resources permit, select one or more of the internal state variables. Second, to provide an unbiased method, the *random approach* creates completely random oracle rankings. The resulting rankings are simply a random ordering of the internal state and output variables.

E. Data Collection

For each given case example, we ran the test suites against each mutant and the original version of the program. For each execution of the test suite, we recorded the value of every internal variable and output at each step of every test using an in-house Lustre simulator. Note that the time required to produce a single oracle ranking—generate mutants, run tests, and apply the greedy set cover algorithm—is quite small, less than one hour for every case example. This raw trace data is then used by our algorithm and our evaluation.

The fault finding effectiveness of an oracle is computed as the percentage of mutants killed versus the number of mutants in the evaluation set used. We perform this analysis

for each oracle and test suite for every case example, and use the information produced by this analysis to evaluate our research questions.

V. RESULTS & DISCUSSION

In this section, we discuss our results in the context of our three research questions: (Q1) “*Is our approach more effective than existing baseline approaches for oracle data selection?*”, (Q2) “*What is the maximum potential effectiveness of our mutation-based technique, and how does the real-world effectiveness compare?*”, and (Q3) “*How does the choice of test input data impact the effectiveness of our approach?*”.

In Figure 2, we plot the median fault finding effectiveness of expected value test oracles for increasing oracle sizes⁵. Four ranking methods are plotted: both baseline rankings, our mutation-based approach, and the idealized mutation-based approach. Median values were used for these plots as plotting all test suite/training set combinations (using boxplots, scatter-plots, etc.) yields figures that are very difficult to interpret. For each subfigure, we plot the number of outputs as a dashed, vertical blue line. This line represents the size of an output-only oracle; this is the oracle size that would generally be used in practice. We also plot the 5% and 2.5% thresholds for recommending oracle sizes as solid orange lines (see Section III-C). Note that the 2.5% threshold is not always met for the oracle sizes explored.

In Table III, we list the median relative improvement in fault finding effectiveness using our proposed oracle data creation approach versus the output-base ranking. In Table IV we list the median relative improvement in fault finding effectiveness using the idealized mutation-based approach (an oracle data set built and evaluated on the same mutants) versus our mutation-based approach. As shown in Figure 2, random oracle data performs poorly, and detailed comparisons were thus deemed less interesting and are omitted.

A. Statistical Analysis

Before discussing the implications of our results, we would like to first determine which differences observed are statistically significant. That is, we would like to determine with statistical significance, at what oracle sizes and for which case examples, (1) the idealized performance of a mutation-based approach outperforms the real-world performance of the mutation-based approach, and (2) the mutation-based approach outperforms the baseline ranking approaches. We evaluated the statistical significance of our results using a two-tailed bootstrap permutation test. We begin by formulating the following statistical hypotheses⁶:

⁵For readability, we do not state “median” relative improvement, “median” fault finding, etc. in the text, though this is what we are referring to.

⁶As we evaluate each hypothesis for each case example and oracle size, we are essentially evaluating a set of statistical hypotheses.

H_1 : For a given oracle size m , the idealized approach outperforms the standard mutation-based approach.

H_2 : For a given oracle size m , the standard mutation-based approach outperforms the output-base approach.

H_3 : For a given oracle size m , the standard mutation-based approach outperforms the random approach.

We then formulate the appropriate null hypotheses:

H_{01} : For a given oracle size m , the fault finding numbers for the idealized approach are drawn from the same distribution as the fault finding numbers for the standard mutation-based approach.

H_{02} : For a given oracle size m , the fault finding numbers for the standard mutation-base approach approach are drawn from the same distribution as the fault finding numbers for the output-base approach.

H_{03} : For a given oracle size m , the fault finding numbers for the standard mutation-base approach approach are drawn from the same distribution as the fault finding numbers for the random approach.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate our null hypotheses without any assumptions on the distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution assumptions [9], [17]) with 250,000 samples, with median as the test statistic. Per our experimental design, each evaluation set has a paired training set, and each training set has paired baseline rankings (output-base and random). Thus, for each case example and coverage criteria combination, we can pair each test suite T + idealized ranking with T + training set ranking (for H_{01}), and each test suite T + training set ranking with T + random or output-base ranking (H_{02}, H_{03}). We then apply the statistical test for each case example, coverage criteria, and oracle size with $\alpha = 0.05$ ⁷.

Our results indicate that null hypotheses H_{01} and H_{03} can be rejected for all combinations of case examples, coverage criteria, and oracle sizes, with $p < 0.001$. We therefore accept H_1 and H_3 for all combinations of case examples, coverage criteria, and oracle sizes. In the case of H_{02} , there exist combinations in which the differences are not statistically significant, or marginally statistically significant (near α). These combinations mostly correspond to points of little to no relative improvements over output-base oracles. We list these p-values results in Table II⁸ and

⁷Note that we do not generalize across case examples or coverage criteria as the appropriate statistical assumption—random selection from the population of case examples and coverage criteria—is not met. Furthermore, we do not generalize across oracle sizes as it is possible our approach is statistically significant for some sizes, but not others. The statistical tests are employed to where observed differences between oracle data selection methods are unlikely to be due to chance.

⁸Also, for *Latctl_Batch*, when using MC/DC tests and an oracle of size one, H_{02} cannot be rejected as $p = 1.0$.

Oracle Size	Branch	MC/DC		
		DWM_2	DWM_2	DWM_1
1		0.104	0.013	< 0.001
2		1.0	0.203	< 0.001
3		1.0	1.0	< 0.001
4		0.009	1.0	< 0.001
5		0.047	0.408	< 0.001
6		0.293	0.002	< 0.001
7		0.063	0.063	< 0.001
8		0.475	0.470	1.0
9		0.081	1.0	0.407
10		0.133	0.003	0.248
11		1.0	0.008	0.407
12		1.0	< 0.001	1.0
13		0.294	< 0.001	1.0
14		0.004	< 0.001	0.246

Table II
 H_02 P-VALUES

highlight all results without statistical significance with a * in Table III. P-values not specified are < 0.001.

B. Evaluation of Practical Effectiveness (Q1)

When designing an oracle creation support method, the obvious question to ask is, “*Is this better than current best practice?*” As we can see from Figure 2, every oracle generated outperforms the random approach with statistical significance, often by a wide margin. We thus can immediately discard random oracle data selection as a useful method of oracle data selection, as both our approach and the output-base approach outperform it in all scenarios. We do not discuss the random approach further.

We can also see that for both coverage criteria and every case example, nearly every oracle generated for three of four systems outperforms the output-base approaches with statistical significance. The pattern goes as follows: for oracles smaller than the output-only oracle, our approach tends to perform relatively well compared to the output-base approach, with improvements ranging from 0.0 to 145.8%. This reflects the strength of prioritizing variables: we generally select more effective variables for inclusion into the oracle data earlier than the output-base approach. As the test oracle size grows closer in size to the output-only oracle, the relative improvement decreases, but often (4 of 8 case example/coverage combinations) still outperforms the output-only oracle, up to 26.4%. Finally, as the test oracle grows in size beyond the output-only oracle, our relative improvement when using our approach again grows, with improvements of 2.2-45% for largest oracles.

This observation is illustrated best using the *DWM_1* case example. Here we see that, for both coverage criteria, while the output-base ranking method performs relatively well for small oracle sizes, the set-covering approach nevertheless ranks the *most* effective variable first, locating roughly 80.% and 145.8% more faults than the variable chosen by the output-base technique for the branch and MC/DC coverage criteria, respectively. The set-covering oracle continues on to select a handful of variables that find additional faults, but as additional outputs are added to the output-base approach

the relative improvement decreases, becoming statistically insignificant in the case of MC/DC coverage. This indicates the wisdom of the current approach, but also demonstrates room for improvement: for branch coverage, the proposed approach is, at worst, 5.9% better than the output-oracle. For the MC/DC approach, we show no improvement, though we achieve similar fault finding to the output-only oracle using *smaller* oracles. Finally, as the oracle grows, incorporating (by necessity) internal state variables, our approach fares well, as effective internal state variables continue to be added, reaching up to 14.2% relative improvement.

The only exception occurs for the *DWM_2* system. For this case example, although mutation-based oracles tend to be roughly equivalent in effectiveness as output-base oracles (we generally cannot reject H_02 at $\alpha = 0.05$), only for small or large oracles (approximately +6 sizes from the output-only oracle) does our approach do relatively well. Examining the composition of these oracles indicates why: the ranking generated using our approach for this case example begins mostly with output variables, and thus oracles of generated using our approach are very similar to those generated using the output-base approach. The performance gain of mutation-base oracles at small sizes suggest that certain output variables are far more important than others, but what is crucially important at all levels up to the total number of outputs is *to choose output variables for the oracle*.

However, in a few instances, our approach in fact does worse (with statistical significance) than the output-base approach. The issue appears to be the greedy set-coverage algorithm: for *DWM_2*, the approach tends to select a highly effective internal state variable first, which prevents a computationally related output variable from being selected for larger oracle data. Given an optimal set cover algorithm, this issue would likely be avoided. However, eventually, for larger oracle sizes, this issue is corrected, with statistically significant improvements of 11.7% and 18.1% observed.

Despite this inconsistency, it seems clear that our approach can be effective in practice. We can consistently generate oracle data sets that are effective over *different* faults, generally with higher effectiveness than existing ranking approaches. In cases where our approach does not do better, the difference observed tends to be small, if present. Furthermore, we can provide the tester with recommendations of the cost effective oracle sizes, thus avoiding the need for testers to manually estimate an effective oracle size.

C. Potential Effectiveness of Oracle Data Selection Approach (Q2)

As noted previously, there is limited empirical work on test oracle effectiveness. Consequently, it is difficult to determine what constitutes effective oracle data selection—clearly performing well relative to a baseline approach indicates our approach is effective, but it is hard to argue the approach is effective in the absolute sense. We therefore

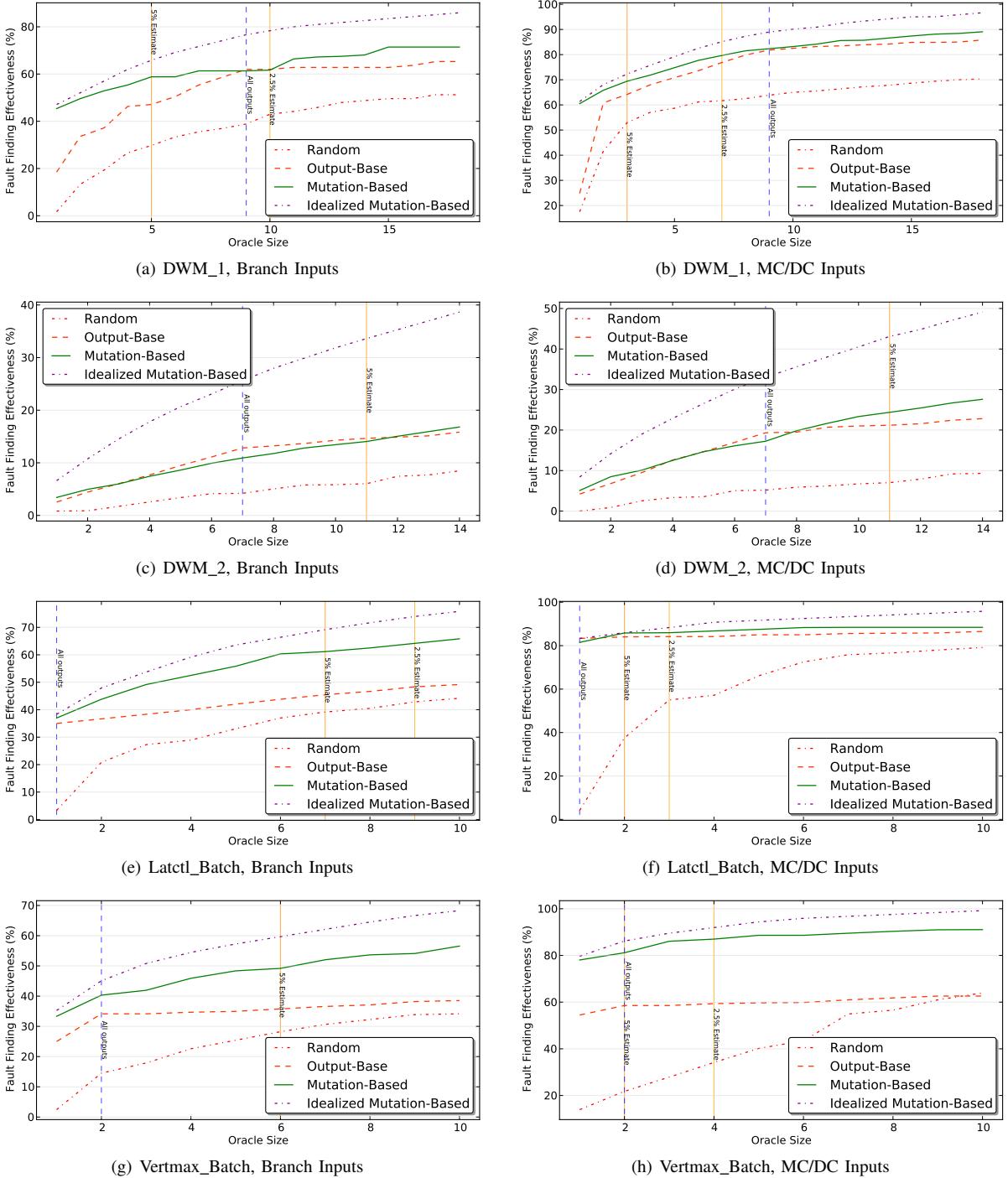


Figure 2. Median Effectiveness of Various Approaches to Oracle Data Selection

posed *Q2*: what is the *maximum* potential effectiveness of a mutation-based approach? To answer this question, we applied our approach to the same mutants used to evaluate the oracles in *Q1* (as opposed to generating oracles from a disjoint training set). This represents an idealized testing scenario in which we already know what faults we are attempting to find, and thus is used to estimate the maximum potential of our approach.

The results can be seen in Figure 2 and Table IV. We can observe from these results that while the *potential* performance of a mutation-based oracle is (naturally) almost always higher than the real-world performance of our method, the gap between a realistic implementation of our approach and the ideal scenario is often quite small. Indeed, apart from the *DWM_2* system, for most case examples and oracle sizes, the difference between the idealized and

Oracle Size	Branch				MC/DC			
	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch
1	80.0%	50.0%*	34.6%	2.7%	145.8%	40.0%	44.4%	0.0%*
2	17.1%	0.0%*	26.4%	17.6%	9.3%	25.0%*	44.4%	3.8%
3	12.5%	0.0%*	30.5%	25.0%	7.5%	0.0%*	48.3%	3.8%
4	11.1%	-10.0%	30.5%	27.5%	5.7%	0.0%*	46.6%	3.8%
5	10.5%	-10.0%	38.2%	30.2%	5.3%	0.0%*	49.0%	3.8%
6	10.5%	-11.1%*	41.1%	32.6%	5.2%	-10.0%	47.5%	3.6%
7	8.1%	-10.0%*	44.4%	34.7%	2.6%	-11.7%*	47.4%	3.5%
8	5.9%	-8.3%*	43.2%	32.5%	0.0%*	-4.5%*	46.6%	3.5%
9	7.3%	0.0%*	40.0%	33.3%	-1.2%*	0.0%*	44.2%	3.4%
10	7.2%	0.0%*	42.1%	32.6%	-1.1%*	8.6%	45.0%	3.4%
11	7.3%	0.0%*	—	—	-1.1%*	12.0%	—	—
12	8.8%	0.0%*	—	—	0.0%*	17.3%	—	—
13	10.6%	6.6%*	—	—	1.1%*	17.3%	—	—
14	12.1%	11.7%	—	—	2.2%*	18.1%	—	—
15	14.4%	—	—	—	2.3%	—	—	—
16	14.5%	—	—	—	2.2%	—	—	—
17	14.2%	—	—	—	2.2%	—	—	—
18	14.2%	—	—	—	2.2%	—	—	—

Table III
MEDIAN RELATIVE IMPROVEMENT USING MUTATION-BASED SELECTION OVER OUTPUT-BASE SELECTION

realistic scenarios is *less than* the gap between the output-base approach and our approach. Thus we can conclude that while there is clearly room for improvement in oracle data selection methods, our approach appears to often be quite effective in terms of *absolute* performance.

D. Impact of Coverage Criteria (Q3)

Our final question concerns the impact of varying the coverage criteria—and thus the test inputs—on the relative effectiveness of oracle selection. In this study, we have used two coverage criteria of varying strength. Intuitively, it seems likely that when using stronger test suites (those satisfying MC/DC in this study), the potential for improving the testing process via oracle selection would be less, as the test inputs should do a better job of exercising the code.

Precisely quantifying likeness is difficult; however, as shown in Figure 2, for each case example the general relationship seems (to our surprise) to be roughly the same. For example, for the *DWM_1* system, we can see that despite the overall higher levels of fault finding when using the MC/DC test suites, the general relationships between the output-base baseline approach, our approach, and the idealized approach remain similar. We see a rapid increase in effectiveness for small oracles, followed by a decrease in the relative improvement of our approach versus the output-base baseline as we approach oracles of size 10 (corresponding to an output-only oracle), followed by a gradual increase in said relative improvement. In some cases relative improvements are higher for branch coverage (*Latctl_Batch*) and in others they are higher for MC/DC (*Vertmax_Batch*).

These results indicate that, perhaps more than the test inputs used, characteristics of the system under test are the primary determinant of the relative effectiveness of our approach. Unfortunately, it is unclear exactly what these characteristics are. Testers can, of course, estimate the effectiveness of our applying approach by automating our study, applying essentially the same approach used to provide the

user a suggested oracle size⁹. However, this is potentially expensive and provides no insight concerning why our approach is very good (relative to baseline approaches, and in an absolute sense) for some systems and merely equivalent for others. Developing metrics that can allow us to *a priori* estimate the effectiveness of our approach is an area for potential future work.

VI. THREATS TO VALIDITY

External Validity: Our study is limited to four synchronous reactive critical systems. Nevertheless, we believe these systems are representative of the class of systems in which we are interested and our results are therefore generalizable to other systems in the domain.

We have used Lustre [14] as our implementation language rather than a more common language such as C or C++. However, as noted previously, systems written in Lustre are similar in style to traditional imperative code produced by code generators used in embedded systems development. A simple syntactic transformation suffices to translate Lustre code to C code that would be generally be used.

In our study, we have used test cases generated to satisfy two structural coverage criteria. These criteria were selected as they are particularly relevant to the domain of interest. However, there exist many methods generating test inputs, and it is possible that tests generated according to some other methodology would yield different results. For example, requirements-based black-box tests may be effective at propagating errors to the output, reducing the potential improvements for considering internal state variables. We therefore avoid generalizing our results to other methods of test input selection, and intend to study how our approach generalizes to these methods in future work.

⁹Indeed, automating our study to estimate the effectiveness of our approach is akin to applying mutation testing to estimate test input/oracle data effectiveness

Oracle Size	Branch				MC/DC			
	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch
1	3.7%	101.6%	3.6%	4.7%	2.5%	68.0%	2.9%	0.6%
2	8.0%	152.1%	19.5%	8.2%	2.9%	76.9%	6.4%	1.1%
3	12.5%	156.4%	15.4%	15.0%	5.9%	94.9%	6.2%	3.0%
4	14.6%	158.6%	16.1%	13.7%	7.1%	96.0%	7.4%	5.0%
5	16.2%	152.1%	21.0%	13.6%	7.5%	94.6%	7.1%	5.5%
6	17.5%	146.9%	21.9%	12.4%	8.3%	93.5%	8.3%	6.4%
7	19.7%	140.4%	20.9%	17.7%	9.1%	94.3%	9.2%	7.0%
8	22.3%	139.3%	21.7%	20.9%	8.3%	85.7%	10.2%	7.9%
9	22.5%	140.8%	19.6%	14.2%	8.8%	84.2%	9.8%	8.8%
10	20.2%	143.5%	20.2%	16.1%	9.5%	80.8%	10.5%	8.9%
11	21.0%	147.9%	—	—	9.8%	82.5%	—	—
12	18.6%	144.2%	—	—	9.8%	81.0%	—	—
13	19.3%	144.2%	—	—	9.8%	83.9%	—	—
14	17.0%	139.3%	—	—	9.8%	83.0%	—	—
15	17.3%	—	—	—	9.7%	—	—	—
16	17.3%	—	—	—	9.5%	—	—	—
17	18.0%	—	—	—	9.7%	—	—	—
18	19.0%	—	—	—	10.1%	—	—	—

Table IV
MEDIAN RELATIVE IMPROVEMENT USING IDEALIZED MUTATION-BASED SELECTION OVER REALISTIC MUTATION-BASED SELECTION

The tests used are effectively unit tests for a Lustre module, and the behavior of oracles may change when using tests designed for large system integration. Given that we are interested in testing systems at this level, we believe our experiment represents a sensible testing approach and is applicable to practitioners.

We have generated approximately 250 mutants for each case example, with 125 mutants used for training sets and up to 125 mutants used for evaluation. These values are chosen to yield a reasonable cost for the study. It is possible the number of mutants is too low. Nevertheless, based on past experience, we have found results using less than 250 mutants to be representative [22], [23]. Furthermore, pilot studies showed that the results change little when using more than this 100 mutants for training or evaluation sets.

Construct Validity: In our study, we measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software. Given that our approach to selecting oracle data is also based on the mutation testing, it is possible that using real faults would lead to different results. This is especially likely if the fault model used in mutation testing is significantly different than the faults we encounter in practice. Nevertheless, as mentioned earlier, Andrews et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults in similar fault finding experiments [1].

VII. CONCLUSION AND FUTURE WORK

In this study, we have explored a mutation-based method for supporting oracle creation. Our approach automates the selection of oracle data, a key component of expected value test oracles. Our results indicate that our approach is successful with respect to alternative approaches for selecting oracle data, with improvements up to 145.8% over output-base oracle data selection, with improvements in the 10%-30% range relatively common. In cases where our approach

is not more effective, it appears to be comparable to the output-base approach. We have also found that our approach performs within an acceptable range from the theoretical maximum.

While our results are encouraging, and demonstrate that our approach can be effective using real-world avionics systems, there are a number of questions that we would like to explore in the future, including:

- **Estimating Training Set Size:** Our evaluation indicates that for our case examples, a reasonable number of mutants are required to produce effective oracle data sets. However, when applying our approach to very large systems, or when using a much larger number of test inputs, we may wish to estimate the needed training set size.
- **Oracle Data Selection Without Test Inputs:** In our approach, we assume the tester has already generated a set of test inputs before generating oracle data. We may wish to generate oracle data without knowing the test inputs that will ultimately be used. Is oracle data generated with one set of test inputs effective for other, different test inputs?

VIII. ACKNOWLEDGEMENTS

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583 and CNS-0931931, CNS-1035715, an NSF graduate research fellowship, and the WCU (World Class University) program under the National Research Foundation of Korea and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

REFERENCES

- [1] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pages 402–411, 2005.

- [2] L. Baresi and M. Young. Test oracles. In *Technical Report CIS-TR-01-02, Dept. of Computer and Information Science, Univ. of Oregon*.
- [3] L. Briand, M. DiPenta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. on Software Engineering*, 30 (11), 2004.
- [4] J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.
- [5] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.
- [6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.
- [7] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.
- [8] R. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. of the 6th Joint European Software Engineering Conference and Foundations of Software Engineering*, pages 549–552. ACM, 2007.
- [9] R. Fisher. *The Design of Experiment*. New York: Hafner, 1935.
- [10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Prc. of the 19th Int'l Symp. on Software Testing and Analysis*, pages 147–158. ACM, 2010.
- [11] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Prc. of the 20th Int'l Symp. on Software Testing and Analysis*, pages 147–158. ACM, 2011.
- [12] M. Garey and M. Johnson. *Computers and Intractability*. Freeman, New York, 1979.
- [13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.
- [14] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [15] M. P. Heimdahl, G. Devaraj, and R. J. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.
- [16] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, (99):1, 2010.
- [17] P. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley-Interscience, 2007.
- [18] Mathworks Inc. Simulink product web site. <http://www.mathworks.com/products/simulink>.
- [19] The NuSMV Toolset, 2005. Available at <http://nusmv.irst.itc.it/>.
- [20] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. *ECCOP 2005-Object-Oriented Programming*, pages 504–527, 2005.
- [21] A. Rajan, M. Whalen, and M. Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, 2008. Available at <http://crisis.cs.umn.edu/ICSE08.pdf>.
- [22] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.
- [23] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proc. of the 10th Int'l Conf. on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.
- [24] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.
- [25] D. J. Richardson, S. L. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. In *Proc. of the 14th Int'l Conference on Software Engineering*, pages 105–118. Springer, May 1992.
- [26] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
- [27] M. Staats, M. Whalen, and M. Heimdahl. New ideas and emerging results track: Better testing through oracle selection. In *Proc. of NIER Workshop, Int'l. Conf. on Software Engineering (ICSE) 2011*. ACM New York, NY, USA, 2011.
- [28] M. Staats, M. Whalen, and M. Heimdahl. Programs, testing, and oracles: The foundations of testing revisited. In *Proc. of Int'l. Conf. on Software Engineering (ICSE) 2011*. ACM New York, NY, USA, 2011.
- [29] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Automated Software Engineering*, 2008, pages 407–410. IEEE, 2008.
- [30] C. Van Eijk. Sequential equivalence checking based on structural similarities. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 19(7):814–819, 2002.
- [31] J. Voas and K. Miller. Putting assertions in their place. In *Software Reliability Engineering, 1994., 5th Int'l Symposium on*, pages 152–157, 1994.
- [32] Q. Xie and A. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 16(1):4, 2007.

Observable Modified Condition/Decision Coverage

Michael Whalen, Gregory Gay, Dongjiang You

Mats P.E. Heimdahl

Department of Computer Science and Engineering

University of Minnesota, USA

Matt Staats

Division of Web Sciences & Technology

Korea Advanced Institute of Science & Technology, South Korea

Abstract—In many critical systems domains, test suite adequacy is currently measured using structural coverage metrics over the source code. Of particular interest is the modified condition/decision coverage (MC/DC) criterion required for, e.g., critical avionics systems. In previous investigations we have found that the efficacy of such test suites is highly dependent on the structure of the program under test and the choice of variables monitored by the oracle. MC/DC adequate tests would frequently exercise faulty code, but the effects of the faults would not propagate to the monitored oracle variables.

In this report, we combine the MC/DC coverage metric with a notion of *observability* that helps ensure that the result of a fault encountered when covering a structural obligation propagates to a monitored variable; we term this new coverage criterion *Observable MC/DC (OMC/DC)*. We hypothesize this path requirement will make structural coverage metrics 1.) more effective at revealing faults, 2.) more robust to changes in program structure, and 3.) more robust to the choice of variables monitored. We assess the efficacy and sensitivity to program structure of OMC/DC as compared to masking MC/DC using four subsystems from the civil avionics domain and the control logic of a microwave. We have found that test suites satisfying OMC/DC are significantly more effective than test suites satisfying MC/DC, revealing up to 88% more faults, and are less sensitive to program structure and the choice of monitored variables.

I. INTRODUCTION

Test adequacy metrics defined over the structure of a program, such as branch coverage and modified condition/decision coverage (MC/DC) have been used for decades to assess the adequacy of test suites. Such criteria can be useful tools when evaluating a testing effort. Nevertheless, these criteria are quite sensitive to the *structure* of the program under test, e.g., the complexity of Boolean expressions [18].

In our work we have been particularly interested in the coverage criterion Modified Condition/Decision Coverage (MC/DC) [4] since it is used as an exit criterion when testing software for critical software in the avionics domain. For certification of such software, a vendor must demonstrate that the test suite provides MC/DC coverage of the source code [21]. In previous investigations, we have found that the effectiveness of MC/DC is highly dependent on the syntactic structure of the code under test. A simple syntactic transformation, such as *inlining* variables—eliminating intermediate Boolean values to create more complex decisions—can dramatically improve the effectiveness of the MC/DC criterion with increases in fault detection of up to 89% [29].

When examining the discrepancy in fault finding between test suites for non-inlined and inlined programs, we often found that the test case encountered a fault in the code, e.g., an erroneous Boolean operator, leading to a corrupted internal state, but this state was masked out in a subsequent condition and did not propagate to an output. This effect was far more prevalent in programs with many small Boolean expressions whose results were stored in intermediate values (a non-inlined implementation). Furthermore, in both non-inlined and inlined programs, it was common that a test case encountered a fault leading to a corrupted internal state, but the test case was too short to allow the corrupted state to propagate to an output; the test case terminated before the corrupted state became visible in a variable monitored by the test oracle.

The underlying issue is that structural coverage criteria such as MC/DC require only that each syntactic element—in the case of MC/DC, a particular truth assignment of a decision—is covered. Nevertheless, covering an element does not ensure faults found will be observed by a test oracle. In the case of MC/DC, the effects of masking and test length can be overcome if the test oracle monitors all variables in the program under test, i.e., all internal state variables as well as all outputs [25], but this is often prohibitively expensive. Instead, we would prefer to use a coverage criterion requiring that the result of the covered syntactic structure, e.g., a condition, be likely to propagate to the test oracle variables.

To address this issue, we have defined Observable Modified Condition/Decision Coverage (OMC/DC). OMC/DC combines the coverage of decisions required by MC/DC with a path condition that increases the likelihood that a fault encountered when executing the decision will propagate to a monitored variable. Unlike previous extensions to MC/DC [27], this path condition does not increase the number of test obligations over MC/DC; instead, it makes the existing obligations more difficult to satisfy, since the possibility of propagating a fault revealed by the MC/DC obligation must also be demonstrated. We hypothesize that this additional observability obligation will improve the effectiveness of the MC/DC criterion, particularly when used as a test generation target for automated tools, paired with output-based test oracles.

The idea of observability has been explored in hardware testing [7], [9], but our ideas extend this work in several ways. First, we provide a straightforward semantic definition of observability to ground the discussion of the metric. Second, the hardware work is *pessimistically inaccurate*; it states that

certain observable obligations, using our semantic definition, are not observable, making it unsuitable as a coverage target for critical software. Finally, we describe the close connection between the notion of observability and MC/DC.

In this paper, we present experiments conducted on four subsystems from the civil avionics domain and one example of the logic control of a microwave. Our results indicate that test suites generated to satisfy 100% achievable OMC/DC over non-inlined systems achieve between 2% and 88% better fault finding than test suites providing 100% achievable MC/DC when using an oracle observing the output variables only. When using an oracle observing all state variables, the advantage of OMC/DC diminishes, but it is still significant (1% to 14% better fault finding than MC/DC, with a median improvement of 4.5%). We also observed that OMC/DC is dramatically less sensitive to the structure of the program under test than MC/DC—a highly desirable trait of a structural test adequacy metric.

Based on these results, OMC/DC is—for the systems studied—a far more effective test adequacy coverage criterion, both in terms of fault finding and robustness to changes in program structure and variables monitored by the test oracle.

II. BACKGROUND AND PROBLEM STATEMENT

Modified Condition/Decision Coverage (MC/DC) is a white-box structural coverage metric developed as a compromise between the benefits of multiple-condition coverage and the lower number of test cases required by condition/decision coverage [4]. A test suite provides MC/DC over the structure of a program or model if every condition within a decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision’s outcome (the condition of interest cannot be masked out by the other conditions in the decision). Note that when discussing MC/DC, a decision is defined to be any Boolean expression and a condition is an atomic Boolean expression with no connectives such as and or or.

While MC/DC ensures that a condition will not be masked out in a decision, it is still possible that the condition will ultimately be masked out within some sequence of statements in a program. As an example, consider the trivial program fragment in Table I. Based on the definition of MC/DC, TestSet1 in Table I provides MC/DC over the program fragment; the test cases with `in_3 = false` (bold faced) contribute towards MC/DC of `in_1 or in_2` in `stmt1`. Nevertheless, if our oracle monitors the output variable `out_1`, the effect of `in_1` and `in_2` cannot be observed in the output since it will be masked out by `in_3 = false`. Thus, TestSet1 gives us MC/DC coverage of the program fragment, but a fault on the first line will never propagate to the output. On the other hand, TestSet2 will also give MC/DC coverage of the program, but since `in_3 = true` in the first two test cases, faults in the first statement can propagate to an output.

This masking problem can be addressed by monitoring all internal state variables, but the use of such a strong test oracle is often cost-prohibitive (or outright infeasible). An alternative

TABLE I
SAMPLE PROGRAM SUSCEPTIBLE TO MASKING

```
expr_1 = in_1 or in_2;      //stmt1
out_1 = expr_1 and in_3;    //stmt2

Sample Test Sets for (in_1, in_2, in_3):
TestSet1 = { (TFF), (FTF), (FFT), (TTT) }
TestSet2 = { (TFT), (FTT), (FFT), (TFF) }
```

approach is to strengthen the coverage criterion to include a notion of *observability* of expressions in the variables monitored by the test oracle. To this effect, in this paper we propose a new test-adequacy coverage criterion—Observable MC/DC (OMC/DC). Informally, OMC/DC establishes observability of decisions by requiring that the variable whose assignment contains a particular Boolean decision remains unmasked through a path to a variable monitored by the test oracle (commonly an output variable).

Observability is a measure of how well internal states of a system can be inferred, usually through the values of its external outputs [25]. We state that an expression in a program is *observable* in a test case if we can modify its value, leaving the rest of the program intact, and observe changes in the output of the system. If we cannot find such a value, then the expression is *not observable* for that test case.

More formally, we can view a deterministic program P containing expression e as a transformer from inputs to outputs: $P : I \rightarrow O$. We write $P[v/e_n]$ for program P where the computed value for the n^{th} instance of expression e is replaced by value v . Note that this is not substitution but akin to mutation; we are replacing a *single* instance of expression e rather than all instances. We say e is observable in test t if $\exists v. P(t) \neq P[v/e_n](t)$. This idea can be straightforwardly lifted from test cases to test suites.

This formulation is a generalization of the semantic idea behind masking MC/DC [3], lifted from decisions to programs. For masking MC/DC, the main obligation¹ is that given decision D , for each condition c in D , we want a pair of test cases t_i and t_j that ensure c is observable for both *true* and *false* values: $(D(t_i) \neq D[\text{true}/c_n](t_i)) \wedge (D(t_j) \neq D[\text{false}/c_n](t_j))$. Given this definition, one can directly lift MC/DC obligations to observable MC/DC obligations by moving the observability obligation from the decision to the program output. Given test suite T , the OMC/DC obligations are:

$$(\forall c_n \in \text{Cond}(P). (\exists t \in T . (P(t) \neq P[\text{true}/c_n](t))) \wedge (\exists t \in T . (P(t) \neq P[\text{false}/c_n](t))))$$

where $\text{Cond}(P)$ is the set of all conditions in program P .

III. TAGGED SEMANTICS

Unfortunately, the semantic definition for observability is unwieldy both for test generation and especially for test

¹The other obligations being that each decision evaluates to both true and false and that each entry and exit point has been invoked; these can be added to the observable MCDC criteria with no difficulty.

measurement. First, the analysis requires that two versions of the program run in parallel to check that the results match. Second, for test measurement, the test suite must be run separately for *each pair* of modified programs.

In order to define an observability constraint that more efficiently supports monitoring and test generation, we approximate semantic observability using a tagging semantics similar to [9]. We assign each condition a tag and then track the observability of these tags through the execution of a program. If a tag reaches the output, we consider the obligation satisfied. More accurately, we track pairs: the first is the condition tag (uniquely assigned for each condition instance in the program syntax), and the second is the Boolean outcome of the condition. The level of coverage for a test suite can be assessed by examining how many of all possible pairs within the program have reached an output in some test.

To demonstrate the generality of the approach, we define semantics both for an imperative command language and a simple dataflow language sharing a common set of expressions, shown in Table II. For presentation, we use a reduction semantics with evaluation contexts (RSEC) [10] which we machine checked for consistency using the K tool suite [20]. The rules operate over *configurations* that contain the syntax being evaluated (\mathcal{K}) and a set of labeled configuration parameters. To simplify presentation, elements of the configuration are not shown in rules if not used or modified. The rules operate by applying rewrites at positions in the syntax that are allowed by the *evaluation context* (the *Context* definition). A context is a program or fragment of a program with a *hole*, where the hole (represented by \square) defines a placeholder where a rewrite can occur. We assume appropriate definitions for maps including lookup (σx) and update $\sigma[x \leftarrow v]$ operations, the empty map \emptyset , and lists with concatenation $x.y$ and cons $elem :: x$, operators. During rewriting, additional syntax may be introduced; we distinguish this syntax from user-level syntax by formatting it against a gray background.

Expressions yield (Val, TS) pairs (where TS is a set of tags) and are evaluated in a context containing environment \mathcal{E} of type $Env = (id \rightarrow (Val \times TS))$. The expressions are standard except the $\text{tag}(t, e)$ expression, which adds a tag to the set of tags associated with the expression e . For OMC/DC, it is assumed that each condition is wrapped in a tag expression. The Boolean operators *and*, *or* define masking: given a and b , the value of a only matters if b is true, so a 's tags only propagate if b is true (and vice-versa); *or* is similar and not shown in Table II due to space constraints.

The imperative language semantics describe how tags propagate through commands. The main issue involves conditional statements: tags in conditions should propagate through *all* variables assigned in *either* branch, as a condition may affect the value of the variable by *not* assigning it. We extend the expression configuration to include $C : TS$ to store the set of condition tags. Conditional statements add tags to C that must be removed once the statement has completed. To do so, an *end* statement is appended to reset C and propagate the conditional tags to all variables assigned in the conditional

TABLE II
SYNTAX AND TAGGING SEMANTICS FOR AN IMPERATIVE AND DATAFLOW LANGUAGE

Expression syntax, context, and semantics:

$E ::=$	$Val Id E \text{ op } E \text{not } E $
	$E ? E : E \text{tag}(E, T) (Val, TS) \text{addTags}(E, TS)$
$Context ::=$	$\square Context \text{ op } E E \text{ op } Context \text{not } Context $
	$Context ? E : E \text{addTags}(Context, TS) $
	$\langle \mathcal{K} : Context, \mathcal{E} : Env, \dots \rangle$
lit	$n \Rightarrow (n, \emptyset)$
var	$\langle \mathcal{E} : \sigma \rangle [x] \Rightarrow \langle \mathcal{E} : \sigma \rangle [(\sigma x)] \text{ if } x \in \text{dom}(\sigma)$
op	$(n_0, l_0) \oplus (n_1, l_1) \Rightarrow (n_0 \oplus n_1, l_0 \cup l_1)$
and₁	$(tt, l_0) \text{ and } (tt, l_1) \Rightarrow (tt, l_0 \cup l_1)$
and₂	$(tt, l_0) \text{ and } (ff, l_1) \Rightarrow (ff, l_1)$
and₃	$(ff, l_0) \text{ and } _ \Rightarrow (ff, l_0)$
ite₁	$(tt, l_0) ? e_t : e_e \Rightarrow \text{addTags}(e_t, l_0)$
ite₂	$(ff, l_0) ? e_t : e_e \Rightarrow \text{addTags}(e_e, l_0)$
tag	$\text{tag}(t, (v, l)) \Rightarrow (v, l \cup \{(t, v)\})$
addt	$\text{addTags}((v, l_0), l_1) \Rightarrow (v, l_0 \cup l_1)$

Imperative command syntax, context and semantics:

$S ::=$	$\text{skip} S; S \text{if } E \text{ then } S \text{ else } S $
	$Id := E \text{while } E \text{ do } S \text{end } (\text{List } Id, TS)$
$Context ::=$	$\dots Id := Context \text{if } Context \text{ then } S \text{ else } S $
	$Context; S \langle \mathcal{K} : Context, \mathcal{E} : Env, C : TS \rangle$
asgn	$\langle \mathcal{E} : \sigma \rangle [x := (n, l)] \Rightarrow \langle \mathcal{E} : \sigma[x \leftarrow (n, l)] \rangle [\text{skip}]$
seq	$\text{skip}; s_2 \Rightarrow s_2$
cond₁	$\langle C : c \rangle [\text{if } (tt, l) \text{ then } s_1 \text{ else } s_2] \Rightarrow$
	$\langle C : c \cup l \rangle [s_1; \text{end}(V, c)] \text{ where } V = (\text{Assigned } s_1).(\text{Assigned } s_2)$
cond₂	$\langle C : c \rangle [\text{if } (ff, l) \text{ then } s_1 \text{ else } s_2] \Rightarrow$
	$\langle C : c \cup l \rangle [s_2; \text{end}(V, c)] \text{ where } V = (\text{Assigned } s_1).(\text{Assigned } s_2)$
while	$\text{while}(e) s \Rightarrow \text{if } (e) \text{ then } (s; \text{while}(e) s) \text{ else skip}$
endcond₁	$\langle C : c' \rangle [\text{end } (\text{nil}, c)] \Rightarrow \langle C : c \rangle [\text{skip}]$
endcond₂	$\langle \mathcal{E} : \sigma, C : c' \rangle [\text{end } (x :: V, c)] \Rightarrow \langle \mathcal{E} : \sigma', C : c' \rangle [\text{end } (V, c)]$
	where $(\sigma x) = (n, l)$ and $\sigma' = \sigma[x \leftarrow (n, l \cup c')]$
prog	$s \Rightarrow \langle \mathcal{K} : s, \mathcal{E} : \emptyset, C : \emptyset \rangle$

Dataflow program syntax, context, and semantics:

EQ	$Id = E Id = \text{pre}(E)$
Prog	$(I, Env, \text{List } EQ)$
$Context ::=$	$\dots Context; \text{List } EQ Context :: \text{List } EQ $
	$EQ :: Context Id = Context Id = \text{pre}(Context) $
	$\langle \mathcal{K} : Context, \mathcal{I} : \text{List } Env, \mathcal{O} : \text{List } Env,$
	$\mathcal{E} : Env, \mathcal{S} : Env \rangle$
comb	$\langle \mathcal{E} : \sigma \rangle eqs_0.((x = (n, l)) :: eqs_1) \Rightarrow$
	$\langle \mathcal{E} : \sigma[x \leftarrow (n, l)] \rangle eqs_0.eq_1$
state	$\langle \mathcal{S} : \sigma \rangle eqs_0.((x = \text{pre } (n, l)) :: eqs_1) \Rightarrow$
	$\langle \mathcal{S} : \sigma[x \leftarrow (n, l)] \rangle eqs_0.eq_1$
write	$\langle \mathcal{O} : \kappa, \mathcal{E} : c \rangle \text{ nil}; eqs \Rightarrow \langle \mathcal{O} : \kappa.[c], \mathcal{E} : c \rangle eqs$
cycle	$\langle \mathcal{I} : \sigma_i :: \iota, \mathcal{E} : _, \mathcal{S} : \sigma_l \rangle eqs \Rightarrow$
	$\langle \mathcal{I} : \iota, \mathcal{E} : (\sigma_i \cup \sigma_l), \mathcal{S} : \emptyset \rangle eqs; eqs$
prog	$(i, s, eqs) \Rightarrow \langle \mathcal{I} : i, \mathcal{O} : \text{nil}, \mathcal{S} : s, \mathcal{E} : \emptyset, \mathcal{K} : eqs \rangle$

body (using *(Assigned s)*, a helper function that returns the list of variables assigned by a statement *s*). Given a context containing input variable values, these rules determine the set of tags that propagate to outputs.

Dataflow languages, such as Simulink and SCADE, are popular for model-based development, and assign values to a set of equations in response to periodic inputs. To store system state, state variables ($\frac{1}{z}$ blocks in Simulink) are used. Our dataflow language consists of assignments to combinatorial and state variables, and the semantics are defined over lists (traces) of input variable values. The expression configuration is extended to contain an input trace *I*, output trace *O*, and state environments *S*. Evaluation proceeds by cycles: at the beginning of a cycle, the **cycle** rule constructs the initial evaluation environment. During a cycle, variable values are recorded using the **comb** and **state** rules. Note that the context does not force an ordering on evaluation of equations; instead, an equation can evaluate as soon as all variables it uses have been stored in the environment. When all equations have been computed, the **write** rule appends the environment to the output list. The **prog** rule, given an input list, an initial state environment, and a list of equations, initializes the configuration for the **cycle** rule. Coverage can be determined by examining the tags stored in the output environment list.

Note that both the tagging semantics are *optimistically inaccurate* with respect to observability; that is, they may report that certain conditions are observable when they are not. This is easily demonstrated by a small code fragment:

```
if (c) then out := 0 else out := 0 ;
```

The semantic model of observability will correctly report that *c* is not observable; it cannot affect the outcome of this code fragment. However, the tagging model propagates the tags of *c* to the assignments in the *then* and *else* branches.

IV. TEST GENERATION FOR DATAFLOW PROGRAMS

In the previous section, we presented an extended semantics that accounts for tags in imperative and dataflow programs. In order to generate tests, we would like to instead annotate the program and describe *trap properties* to track the tags. We will generate test obligations such that each obligation is suitable for tracking a *single* tag and determining whether it propagates to an output. This is accomplished by conjoining an MC/DC coverage obligation over a single variable (as described in [18]) with a path condition representing the variable's observability at one of the monitored variables. We describe this annotation for the dataflow language Lustre [13] in order to measure test coverage of industrial Simulink models in Section V.

A. Immediate Non-Masking Paths

A variable *x* is observable if it is not masked along some computation path (as described in the tagged semantics) to a monitored variable. If the path is entirely within one computational step (i.e., it does not go through any delays), we call it an *immediate non-masking path* and the variable is *immediately observable*. This can be defined inductively by examining

the variables that use *x* in their definition: if one of these variables *y* is immediately observable, and *x* is not masked in the definition of *y*, then *x* is immediately observable. We track these notions by defining additional variables to track this information: *x_IMM_USED_BY_y* which is true if *x* is not masked in the definition of *y*, and *x_IMM_OBSERVED* if *x* has an immediate non-masking path. Suppose we had the following equations, where *out1* is an observed variable:

```
out1 = v1 and v2 ;
v1 = true ;
v2 = if (input1) then (v3) else (v1) ;
v3 = true ;
```

In this case, we could generate additional definitions to track the observability of the variables as follows:

```
v1_IMM_USED_BY_out1 = v2 ;
v2_IMM_USED_BY_out1 = v1 ;
input1_IMM_USED_BY_v2 = true ;
v3_IMM_USED_BY_v2 = input1 ;
v1_IMM_USED_BY_v2 = (not input1) ;

out1_IMM_OBSERVED = true ;
v1_IMM_OBSERVED =
  (v1_IMM_USED_BY_out1 and out1_IMM_OBSERVED) or
  (v1_IMM_USED_BY_v2 and v2_IMM_OBSERVED) ;
v2_IMM_OBSERVED =
  v2_IMM_USED_BY_out1 and out1_IMM_OBSERVED ;
input1_IMM_OBSERVED =
  input1_IMM_USED_BY_v2 and v2_IMM_OBSERVED ;
v3_IMM_OBSERVED =
  v3_IMM_USED_BY_v2 and v2_IMM_OBSERVED ;
```

v1 is used in two equations and therefore has two immediate paths to observability: one through *v2* and another directly through *out1*, while the other variables are each used once, so have one immediate path.

B. Delayed Non-Masking Paths

Although many variables can be immediately observed, often, the effect of a variable on an output can only be observed after several steps. In each of these intermediate steps, its tag is stored in a delay, until it eventually propagates to an output. We call this a *delayed non-masking path* and the variable is *delay observable*. This situation can be broken into immediate observations: the first from a variable to a latch, the next from the latch to another latch, etc., until an output is reached. Suppose we had the following Lustre program²:

```
delay1 = 0 -> pre(v1) ;
v1 = v2 and delay2 ;
v2 = in1 ;
delay2 = 0 -> pre(in1) ;
```

In the same way that we modeled immediate observability, we could talk about immediate use by delay equations:

²In Lustre, latches are represented slightly differently than in the language in Section II: the Lustre equation *var = init -> pre(expr)* contains both the initial value of the latch *init* and the latch expression *pre(expr)*, whereas our semantics imports the initial value of latches through an initial latch environment *s*.

```

v2_IMM_USED_BY_v1 = delay2 ;
delay2_IMM_USED_BY_v1 = v2 ;
v1_DEL_USED_BY_delay1 = true ;
v2_DEL_USED_BY_delay1 = v2_IMM_USED_BY_v1
    and v1_DEL_USED_BY_delay1 ;
delay2_DEL_USED_BY_delay1 = delay2_IMM_USED_BY_v1
    and v1_DEL_USED_BY_delay1 ;

```

We now have a mechanism that defines immediate paths to latches. What is necessary is some means to knit these paths together to define a sequential path through (possibly) several delays to an output. We accomplish this using a *token* variable that describes the current delay location. Once the token is initialized to a delay variable X , it can non-deterministically move to any other delay location (as long as X is `DEL_USED_BY` that location) or to a special `COMPLETE` state (if X is immediately observed). If the token cannot move, because it is not observable at another delay or the output, the token moves to an `ERROR` state and stays there.

C. Test Obligations

An OMC/DC coverage obligation can be represented as an MC/DC obligation over a single variable conjoined with a path condition describing the observability of that variable at one of the monitored variables. For delayed paths, we have to describe the instant in which the expression was immediately observable at a delay (called *capture*). We then want to latch this fact for the rest of the execution, hoping that the token will propagate to an output. So, the test consists of the original MC/DC obligation (`v2_AT_v1_TRUE`) below and a path condition, which can be satisfied by either be an *immediate* or *delayed* path, as described in the following Lustre code:

```

v2_AT_v1_TRUE = in1 and delay2 ;
v2_AT_v1_TRUE_CAPTURE = (v2_AT_v1_TRUE and
    (v1_DEL_USED_BY_delay1 and token=delay1) ;
v2_AT_v1_TRUE_CAPTURED = v2_AT_v1_TRUE_CAPTURE ->
    (v2_AT_v1_TRUE_CAPTURE or
     pre(v2_AT_v1_TRUE_CAPTURED))
v2_true_ob = ((v2_AT_v1_TRUE and v1_IMM_OBSERVED)
    or (v2_AT_v1_TRUE_CAPTURED and token=TOK_COMPLETE))
```

V. EVALUATION & EXPERIMENT

We wish assess the *quality* in terms of fault finding of the test suites generated to satisfy OMC/DC as compared to masking MC/DC [3]. We also want to evaluate the effect of program structure on the effectiveness of test suites generated to provide OMC/DC. Thus, we address the following questions:

- 1) Are test suites generated to provide OMC/DC more effective at revealing faults than test suites generated to satisfy masking MC/DC?
- 2) How robust is the OMC/DC criterion to the structure of the program under test? Will the effectiveness of OMC/DC change as program structure changes?

Additionally, we are interested in the nature of the tests generated to satisfy the OMC/DC and MC/DC coverage criteria:

- 3) How do the length of the individual tests, the size of test suites, and the percentage of achievable coverage compare between OMC/DC and MC/DC for the systems included in our experiment?

A. Experimental Setup Overview

In this research, we have used four industrial systems developed by Rockwell Collins engineers. Two of these systems, *DWM1* and *DWM2*, represent distinct portions of a Display Window Manager (DWM) for a cockpit display system. The other two systems, *Vertmax* and *Latctl*, describe the vertical and lateral mode logic for a Flight Guidance System. In addition, we have used a Microwave System — control software for a generic microwave oven developed as a non-proprietary teaching aid at Rockwell Collins.

Each of the systems under test represent sizable, realistic industrial systems. The size of each system and number of variables are listed in Table III.

TABLE III
CASE EXAMPLE INFORMATION

	Subsys.	Blocks	Outputs	Internal Vars.
DWM1	3109	11,439	7	569
DWM2	128	429	9	115
Vertmax	396	1,453	2	415
Latctl	120	718	1	128
Microwave	22	101	4	162

For each case example, we generated inlined and non-inlined versions of each system (detailed in Section V-B below). Then, for each implementation of each system, we:

- 1) Generated 10 test input suites each for OMC/DC and MC/DC (Section V-C).
- 2) Generated 250 mutants of each system (Section V-D).
- 3) Ran test suites on mutants with output-only and maximum test oracles (Section V-E).
- 4) Assessed fault finding of each test suite and oracle combination. (Section V-E).

B. Inlined and Non-Inlined Implementations

Our subject systems are modeled using the Simulink notation from Mathworks Inc. [17] and were automatically translated into the Lustre synchronous programming language [13] in order to take advantage of existing automation. This is analogous to the automated code generation from Simulink offered by Mathworks' Real Time Workshop. Lustre can be automatically translated to C code.

When translating these systems from Simulink to Lustre, a number of options exist on how to structure the generated code. For example, one can factor complex boolean expressions through the introduction of additional variables or one can inline expressions to reduce the number of variables while increasing the complexity of the boolean expressions. As we know the structure of the program under test influences the effectiveness of the MC/DC criterion [18], we have generated both inlined (complex boolean conditions) and non-inlined systems (intermediate variables used to factor expressions).

C. Test Suite Generation

We use a counterexample-based test generation approach to generate tests satisfying masking MC/DC and OMC/DC [11][19]. This approach is guaranteed to generate a

test suite that achieves the maximum possible coverage of the system under test. We have used the Kind model checker [12] in our experiments.

The obligations generated to satisfy OMC/DC will differ depending on the set of monitored variables. In this study, we generate OMC/DC with respect to the output variables of each system, as an output-only oracle is most likely to be used in practice. Note when using the maximum oracle, all variables are observable, and so the OMC/DC obligations are equivalent to MC/DC obligations.

Counterexample-based test generation results in a separate test for each generated coverage obligation. This results in a large amount of redundancy in the generated tests, as each test case likely covers several coverage obligations. Such an unnecessarily large test suite is unlikely to be used in practice. We therefore have reduced each generated test suite while maintaining a consistent level of coverage. To generate these reduced test suites, we make use of a simple randomized greedy algorithm. We begin by determining the coverage obligations satisfied by each test generated, and initialize an empty test set, *reduced*. We then select a test input at random from the full set of tests; if this test satisfies any obligations not satisfied by the existing test inputs in *reduced*, we add it to the set. This process continues until all tests have been removed from the full set.

In these experiments, we have produced 10 different test suites for each case example and program structure to eliminate the possibility that we by accident create a very good (or very poor) test suite in the test suite reduction step.

D. Mutant Generation

Mutation testing is the practice of automatically generating *faulty* implementations of a system for the purpose of empirically examining the fault-finding potential of a test suite [6]. During mutation testing, clones of the system under test are created by introducing a single fault into the program. This method is designed such that all mutants produced are both syntactically and semantically valid. That is, the mutants will compile, and no mutant will “crash” the system under test.

The mutation operators used in this experiment are similar to those used by other researchers, for example, arithmetic, relational, and boolean operator replacement, boolean variable negation, constant replacement, and delay introduction (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value). A detailed description is available in [18].

For each case example, we created 250 mutants. We then remove *functionally equivalent* mutants from each evaluation set using the Kind model checker. This is possible due to the nature of the systems examined in this research. Each system is finite; therefore, determining equivalence is decidable (and, in practice, fast)³.

³Equivalence checking is common in the hardware domain; Van Eijk provides a nice introduction [26].

E. Test Oracles and Data Collection

For the inlined and non-inlined implementations of each case example, we ran the reduced test suites against each mutant and the original version of the system. For each test suite, we recorded the value of every internal variable and output at every step of the execution of every test case using an in-house Lustre interpreter.

To determine the fault finding effectiveness of the generated test suites, we paired each suite with two different *expected value test oracles* [25]. When using an expected value oracle, for each test input, concrete values are specified that the system is expected to produce for one or more variables monitored by the oracle (internal states and/or outputs). We have found that this form of test oracle is commonly used by our industrial partners in the testing of critical software systems. In this study, we have chosen two expected value test oracles, (1) *output-only*, an oracle that compares expected and actual values for each of the system’s output variables, and (2) *maximum*, an oracle that compares values for all internal and output variables.

We compute the fault finding of an oracle/test suite pairing as the percentage of mutants killed. We perform this analysis for each oracle and test suite for every case example.

VI. RESULTS & DISCUSSION

In this section, we address our research questions and discuss the implications of our results. We begin by presenting the median percent of seeded faults revealed by each combination of test suite and oracle type in Table IV (plotted in Figure 1).

A. RQ1: Fault Finding Effectiveness

We would first like to determine whether OMC/DC performs better than MC/DC with respect to fault finding. To address this question we formulated the following hypothesis:

- H_1 : For a given oracle and program structure, the test suite satisfying OMC/DC reveals more faults than the test suite satisfying MC/DC.

This is paired with the appropriate null hypothesis:

- H_0 : For a given oracle and program structure, the fault finding results for the test suite satisfying OMC/DC are drawn from the same distribution as the fault finding results for the test suite satisfying MC/DC.

Our observations are drawn from an unknown distribution. Therefore, we use a two-sided Mann-Whitney-Wilcoxon rank-sum test [30], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. As we cannot generalize across non-randomly selected case examples, we apply the statistical test for each pairing of case example, program structure, and oracle type with $\alpha = 0.05$.

Our results indicate that the null hypotheses can be rejected for all combinations of case examples, program structures, and oracle types, with $p < 0.001$. Furthermore, we can see in Table IV that test suites satisfying OMC/DC outperform those satisfying MC/DC in all cases, with improvements in fault

TABLE IV
PERCENT OF MUTANTS KILLED FOR EACH CASE EXAMPLE, MEDIAN OVER 10 REDUCED TEST SUITES

		DWM1	DWM2	Latctl	Vertmax	Microwave
Non-Inlined	OMC/DC Output-Only	91%	96%	95%	98%	93%
	MC/DC Output-Only	3%	77%	55%	41%	59%
	OMC/DC Maximum	100%	99%	99%	99%	95%
	MC/DC Maximum	86%	92%	96%	86%	89%
Inlined	OMC/DC Output-Only	88%	97%	97%	96%	95%
	MC/DC Output-Only	82%	95%	92%	80%	73%
	OMC/DC Maximum	100%	99%	100%	100%	95%
	MC/DC Maximum	99%	98%	97%	89%	93%

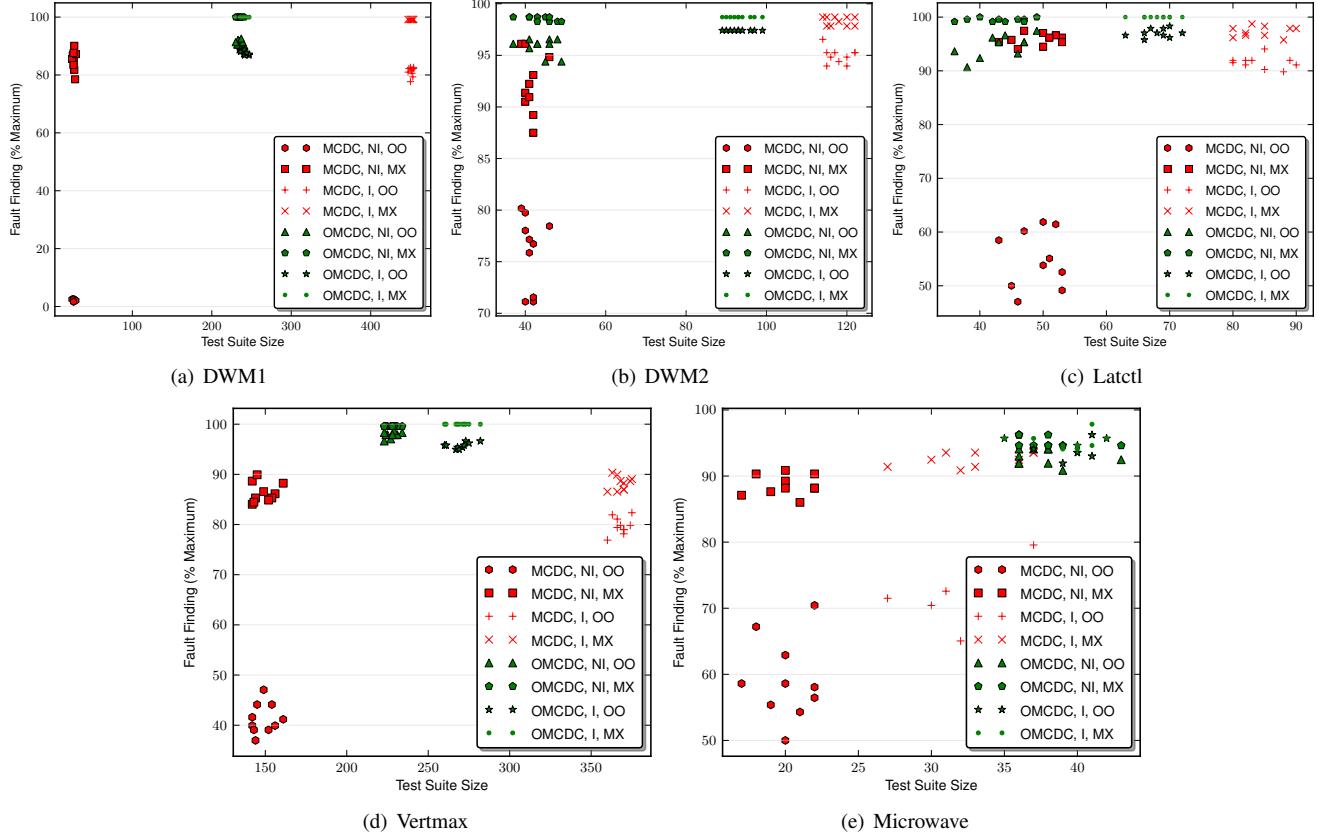


Fig. 1. Percent of mutants killed, plotted against reduced test suite size, for each implementation and oracle pairing for each system.

finding of 19-88% for non-inlined systems using an output-only oracle (3-14% when using a maximum oracle) and 2-22% for inlined systems when using an output-only oracle (1-11% when using a maximum oracle).

We therefore accept H_1 —for all combinations of system, program structure, and oracle type—and conclude that test suites satisfying OMC/DC provide a statistically significant and practical improvement over those satisfying MC/DC.

B. RQ2: Sensitivity to Program Structure

Previous research indicates that MC/DC is sensitive to program structure [18]; inlining a program generally makes it more difficult to achieve MC/DC over a program, but yields a large improvement in fault finding for test suites satisfying MC/DC, with one study demonstrating increases of 9-89% [29]. As seen in Table IV and summarized in Table V, we see similar increases of 14-79% for output-only oracles.

This sensitivity is undesirable for any criterion, as the value of satisfying the criterion depends heavily on how the program is written; indeed, the sensitivity of MC/DC to program structure, and its negative impact on automated test generation, is one of the motivations for the creation of OMC/DC. The cause of MC/DC's sensitivity relates to (1) errors not propagating to observed variables due to masking, coupled with (2) inlining imposing additional constraints when computing MC/DC obligations over complex expressions, which limits opportunities for masking.

In the studied examples, OMC/DC is far less sensitive to changes in program structure. When paired with an output-only oracle, we see a median improvement of 1% from inlining (and, in two cases, actually see a -2 to -3% decrease in fault finding). When using a maximum oracle, the effect is negligible. Thus, while inlining has been proposed as a solution to the masking and propagation-related deficiencies

of MC/DC [29], the addition of the path condition required by OMC/DC, which explicitly addresses issues with masking, is also an effective, and—in our opinion—cleaner and more straightforward solution.

TABLE V
MEDIAN IMPROVEMENT FROM INLINING

Case Example	Oracle	OMC/DC	MC/DC
DWM1	Output-Only	-3%	79%
	Maximum	0%	13%
DWM2	Output-Only	1%	18%
	Maximum	0%	6%
Latctl	Output-Only	2%	37%
	Maximum	1%	1%
Vertmax	Output-Only	-2%	39%
	Maximum	1%	3%
Microwave	Output-Only	2%	14%
	Maximum	0%	4%

On a related note: the use of a maximum oracle has also been proposed as a method of addressing issues with propagation. As we can see in Table IV, generally test suites satisfying OMC/DC with an output-only oracle perform the same or better than suites satisfying MC/DC with a maximum oracle (though, gains are low to negligible). Only in one case—*DWM1* when inlined—do suites satisfying MC/DC achieve significantly higher median fault finding (99% versus 88%).

C. RQ3: Test Suites and Coverage Obligations

Our final question is concerned with the cost and challenge of satisfying OMC/DC. In particular, does satisfying OMC/DC require more test inputs, must the test inputs be longer, and what percentage of obligations are uncoverable, i.e., cannot be achieved by any test input? Note that as OMC/DC requires the same obligations as MC/DC (with an additional path condition), the total number of obligations required to cover a system will be the same for both criteria.

From the information in Table VI, we can see that (1) there is no definitive pattern for test suite size; sometimes reduced suites for OMC/DC are larger than those for MC/DC and vice-versa, and (2), OMC/DC obligations are more likely to be uncoverable, with as low as 57.5% of obligations coverable.

With respect to size, we can see that for non-inlined systems, OMC/DC generally requires more—sometimes many more—test inputs than MC/DC. This reflects the strength of OMC/DC: the observability requirements ensure test cases are more diverse with respect to which paths they must take; thus, it is less likely that several test cases will cover the same obligation. Naturally, more test cases are needed to satisfy 100% achievable OMC/DC. Also, we observed that OMC/DC test case lengths (not shown for space reasons) are slightly longer than that of MC/DC, as test cases must take additional steps through delays to propagate certain variables to outputs.

Nevertheless, while we naturally do not wish to increase the cost of testing, we believe ensuring propagation during testing is necessary; otherwise, why bother to exercise a condition at all? Furthermore, we believe the substantial improvements in testing effectiveness justify the cost, particularly in the domain of critical systems.

The increase in uncoverable obligations is more concerning. For MC/DC, a coverage obligation is uncoverable if it is impossible to demonstrate that a condition can independently affect the outcome of a decision. This situation can occur through interrelationships between conditions in a decision making certain truth assignments impossible. For OMC/DC, this can also cause uncoverable obligations, as can the inability to propagate values to monitored variables. The path constraints accounts for the increase in uncoverable obligations.

Uncovered obligations for structural coverage criteria may reflect problems with the code (conditions that are not properly influencing decisions) or simply eccentricities in the code (e.g., dead code or interrelationships between conditions in a decision). In our case, since we are using a verification tool for test generation, the tool will simply produce maximum achievable coverage with a proof that the uncovered obligations are truly uncoverable. However, if OMC/DC was adopted as a coverage criterion in conjunction with other test generation techniques (e.g., manual or based on heuristic search), it may become necessary—as it currently is with MC/DC in the context of critical avionics systems—to demonstrate that all necessary test have been found, i.e., to demonstrate that all uncovered obligations are indeed uncoverable. This is already a challenging task for MC/DC, and would be made more challenging with the addition of path constraints; how to address this problem is a topic for future research.

VII. RELATED WORK

Lustre and Function Block Diagram (FBD) are data-flow languages that describe how inputs are transformed into outputs instead of describing the control flow of the program. Researchers studying coverage metrics for Lustre [15] and FBD [14] implicitly investigated observability by examining variable propagation from the inputs to the outputs.

Structural coverage metrics for Lustre are based on *activation conditions* that are defined as the condition upon which a data flow is transferred from the input to the output of a path. When the activation condition of a path is true, any change in input causes modification of the output within a finite number of steps [15]. Coverage metrics for FBD are based on a *d-path condition* that is similar to activation conditions in Lustre [14].

Coverage metrics in Lustre and FBD are different from OMC/DC in several respects. First, these metrics check if specific inputs affect the outputs and measure the coverage of variable propagation on all possible paths. OMC/DC, on the other hand, checks if each atomic condition in a Boolean expression affects the monitored variables, and determines if a path exists which propagates the effect of the condition. Second, OMC/DC (as well as MC/DC) is stronger in terms of how a decision must be exercised.

Observability has been studied in testing of hardware logic circuits. Observability-based code coverage metric (OCCOM) is a technique where tags are attached to internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (corrupted state) [7], [9]. A variable is tagged when there is a possible change in the value of the

TABLE VI
NUMBER OF ACHIEVABLE TEST OBLIGATIONS AND MEDIAN REDUCED TEST SUITE SIZE

Case Example	Structure	Total Obligations	OMC/DC Achievable	MC/DC Achievable	OMC/DC Reduced Size	MC/DC Reduced Size
DWM1	Non-Inlined	2038	2036 (99.9%)	2038 (100%)	236	26
	Inlined	2894	1987 (68.7%)	2838 (98.1%)	244	450
DWM2	Non-Inlined	382	343 (89.8%)	364 (95.3%)	43	40
	Inlined	830	477 (57.5%)	538 (64.8%)	92	118
Latctl	Non-Inlined	380	355 (93.4%)	380 (100%)	47	51
	Inlined	260	241 (92.7%)	259 (99.6%)	70	80
Vertmax	Non-Inlined	1732	1700 (98.2%)	1732 (100%)	228	152
	Inlined	1232	1188 (96.4%)	1221 (99.1%)	272	360
Microwave	Non-Inlined	472	325 (68.9%)	467 (98.9%)	36	22
	Inlined	468	338 (72.2%)	441 (94.2%)	40	32

variable due to an fault. The observability coverage can be used to determine whether erroneous effects that are activated by the inputs can be observed at the outputs.

The key differences between OMC/DC and OCCOM are twofold: (1) OMC/DC investigates variable value propagation, while OCCOM investigates fault propagation and (2) OCCOM has pessimistic inaccuracy because of tag cancelation. When both positive and negative tags exist in the same assignment (e.g., different tags in an ADDER or the same tags in a COMPARATOR cancel each other out), no tag is assigned [7] or an unknown tag “?”[9] is used. Variables without tags or with unknown tags are not considered to carry an observable error. In OMC/DC, since we do not make a distinction between positive and negative tags, we do not have tag cancelation or the corresponding pessimistic inaccuracy. Extended work in [8] may fix pessimistic inaccuracy by producing test vectors with specific values, but is highly infeasible.

Dynamic taint analysis, or dynamic information flow analysis, marks and tracks data in a program at runtime, similar to our tagging semantics. This technique has been used in security as well as software testing and debugging [16], [5]. Taint propagation occurs in both explicit information flow (i.e., data dependencies) and implicit information flow (control dependencies). Although the way in which markings are combined varies based on the application, the default behavior is to union them [5]. Thus, dynamic taint analysis is conservative and does not consider masking. More accurate techniques for information flow modeling, such as [28], define path conditions quite similar to those used in this paper to prove *non-interference*, that is, the non-observability of a variable or expression on a particular output.

Dynamic program slicing [1] computes a set of statements that influence the variables used at a program point for a particular execution. This can identify all variables that contribute to a specific program point, including output. However, similarly to dynamic taint analysis, it does not consider masking. Checked coverage uses dynamic slicing to assess oracle quality, where oracles are program assertions [22]. Given a test suite, it yields a percentage of all statements that contribute to the value of any assertion (i.e., are observable at that assertion) vs. the total number of statements covered by the test suite. This work is designed to assess the *oracle*, not the test suite.

VIII. THREATS TO VALIDITY

External Validity: We have chosen to focus on five synchronous reactive critical systems. We believe these systems are representative of the avionics domain and our results are, therefore, generalizable to other systems in this domain.

We have used Lustre [13] as an implementation language rather than a more common language, such as C or C++. As noted in Section V-B, in this domain, systems written in Lustre are similar in structure to systems written in C or C++. Thus, we believe our results are applicable to programs written in more traditional imperative languages.

We have generated approximately 250 mutants for each program structure of each case example. This number was chosen to yield a reasonable cost for the study. It is possible the number of mutants is too low. Based on past experience, however, we have found results using fewer than 250 mutants to be representative [23], [24]. Additional studies have yielded evidence that results plateau when using over 100 mutants.

Internal Validity: We have used a model checker (Kind [12]) to generate test cases. This generation approach provides the shortest test cases that provide the desired coverage. It is possible that test cases derived by hand or through some other automated means, e.g., through heuristic search, may provide different results.

Construct Validity: We measure the fault finding over seeded faults, rather than real faults encountered during development; it is possible that using real faults would lead to different results. However, Andrews et al. have shown that the use of seeded faults like ours leads to conclusions similar to those obtained using real faults in similar experiments [2].

IX. CONCLUSIONS & FUTURE WORK

Structural coverage metrics, such as MC/DC, are commonly used to measure the adequacy of test suites. Such criteria require only that certain code structures, such as a particular Boolean assignment of a decision, be exercised, without requiring the resulting value to affect an observable point in the program. As a result, test suites satisfying these criteria can produce corrupted internal state without revealing a fault, resulting in wasted testing effort.

To address this, we have proposed Observable MC/DC, a combination of traditional MC/DC testing with a notion

of observability—an additional path constraint, which helps ensure that faults will be observed through a *non-masking* path from the point the obligation is satisfied to a variable monitored by the test oracle. Our results indicate that test suites generated to satisfy achievable OMC/DC locate a median of 17.5% (and up to 88%) more faults than test suites providing MC/DC coverage when paired with an oracle observing only the output variables. Furthermore, we have also observed that OMC/DC is less sensitive to the structure of the program under test than MC/DC, and also provides the benefits of using a very strong test oracle with MC/DC coverage.

While our results are encouraging, there are a number of areas open to explore in future research, including:

- **Oracle data selection:** OMC/DC test obligations are defined in terms of both the system structure and a test oracle. In this work, we have paired the case examples with an output-only oracle, but an intelligently selected set of internal and output variables (such as [24]) could potentially yield more cost-effective test suites.
- **Comparison to other coverage metrics:** We have directly compared the performance of OMC/DC to MC/DC. However, there exist many other test adequacy metrics. In particular, we would like to compare the effectiveness and cost of generation of OMC/DC to black-box requirements metrics, which have been previously shown to be adept at propagating faults to the output level [23].
- **Applying observability to other metrics:** The current notion of observability is general and could be adapted to orthogonal metrics such as boundary-value coverage.

ACKNOWLEDGMENT

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A; NSF grants CCF-0916583, CNS-0931931, and CNS-1035715; an NSF graduate fellowship; and the World Class University program under the National Research Foundation of Korea (Project No: R31-30007).

We also thank the Advanced Technology Center at Rockwell Collins Inc. for granting access to industrial case examples.

REFERENCES

- [1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, 1990.
- [2] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608 –624, aug. 2006.
- [3] J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.
- [4] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193–200, September 1994.
- [5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.
- [6] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.
- [7] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, pages 418–425, 1996.
- [8] F. Fallah, P. Ashar, and S. Devadas. Functional vector generation for sequential HDL models under an observability-based code coverage metric. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(6):919–923, 2002.
- [9] F. Fallah, S. Devadas, and K. Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.
- [10] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.
- [11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.
- [12] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [13] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [14] E. Jee, J. Yoo, S. Cha, and D. Bae. A data flow-based structural testing technique for FBD programs. *Information and Software Technology*, 51(7):1131–1139, 2009.
- [15] A. Lakehal and I. Parissis. Structural test coverage criteria for Lustre programs. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 35–43, 2005.
- [16] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 198–209, 2004.
- [17] Mathworks Inc. Simulink product web site. <http://www.mathworks.com/products/simulink>.
- [18] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.
- [19] S. Rayadurgam and M. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.
- [20] G. Rosu and T. F. Serbanuta. An overview of the k semantic framework. *J. of Logic and Algebraic Programming*, 79(6):397 – 434, 2010.
- [21] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
- [22] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.
- [23] M. Staats. *The Influence of Multiple Artifacts on the Effectiveness of Software Testing*. PhD thesis, University of Minnesota, 2011.
- [24] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 870–880, 2012.
- [25] M. Staats, M. Whalen, and M. Heimdahl. Better testing through oracle selection (nietr track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 892–895, 2011.
- [26] C. Van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2002.
- [27] S. Vilkomir and J. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. *Lecture Notes in Computer Science*, 2272:291–308, 2002.
- [28] M. W. Whalen, D. A. Greve, and L. G. Wagner. *Model Checking Information Flow*. Springer-Verlag, Berlin Germany, March 2010.
- [29] M. W. Whalen, M. P. Heimdahl, A. Rajan, and M. Staats. On MC/DC and implementation structure: An empirical study. In *Proceedings of the 27th Digital Avionics Systems Conference*, October 2008.
- [30] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.

Community-Assisted Software Engineering Decision Making

Gregory Gay and Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
greg@greggay.com, heimdahl@cs.umn.edu

1. INTRODUCTION

The field of artificial intelligence (AI) as applied to software engineering problems is constantly growing, with conferences devoted to the topic—such as the annual Mining Software Repositories conference—and large open-source data repositories [8]. AI techniques have seen regular use in the software engineering toolbox. For instance, search-based techniques such as genetic algorithms are effectively used to find test cases that satisfy a predefined test adequacy criterion, e.g., branch coverage [4]. Nevertheless, before such techniques can be applied to problems such as test generation, there are a plethora of higher level decisions that must be made. For example, we must determine what coverage criterion will be the most effective when testing the particular system under test, we must determine which search technique will be the most effective given system characteristics and the coverage criterion, and we must determine what variables the test oracle should monitor during the testing process.

At every stage of the development process, similar high-level decisions must be made. Which design modeling notation is most suitable for the problem at hand? Which verification tool is most suitable for the current model? Will agile techniques be suitable in our situation? Facing such questions, from our experience it seems like most organizations make—in hindsight—highly questionable decisions. It would be desirable for one to get recommendations as to which processes, methods, tools, and techniques should be employed to address the software engineering challenges in a particular situation. Traditional AI approaches, such as case-based reasoning or association rule learning, have been successful at making predictions about specific questions that arise during development—questions such as how many defects to expect given certain source code attributes. To date, however, these algorithms have not been widely applied toward making recommendations about broad, high-level problems (such as our development practice dilemma). We envision a future where a *Software Engineering Recommender* can help an organization make informed development decisions even when they are facing a new—and often unique—situation.

We see two main problems in achieving our vision:

1. The factors influencing the decisions to be made are not clear. For example, even for a relatively simple decision such as which test adequacy criterion would be effective, we do not currently know what characteristics of the system affect the criterion efficacy.
2. The recommendations from our SE Recommender do not depend only on the immediate artifacts surrounding the decision at hand. For example, the choice of test adequacy criterion is dependent on the structure of the program under test [10]. However, other factors such as the function computed by the program (the shape of its state-space), the selection of hardware (affecting observability and controllability of tests), percentage of the variable space monitored, availability of tools in the organization, etc. all affect the choice of coverage criterion. A system basing its recommendations only on the program under test will—in the best case—be suboptimal, and—in the worst case—harmful.

The first problem is particularly important because solution quality is often highly dependent on the quality of the data analyzed. Small changes to data sets (such as additions, deletions, or even preprocessing) often lead to conclusion changes [3]. Data sets that include projects developed across multiple companies are infamous for missing, poorly recorded, or unhelpful attributes, and generally require a nearest-neighbor filtering [12] or feature optimization [2] before they become useful.

Inspiration for overcoming these hurdles could be drawn from the field of recommender systems [7]. Such systems often eschew traditional predictive models in favor of collections of weighted keyword vectors. The user's query is transformed into one of these vectors, and recommendations are made based on the contents of or actions taken by similar items in the data model (where similarity is decided by algorithms from the text mining and information retrieval fields).

One could imagine a data model full of project descriptions, where the user is presented with a list of practice recommendations based on the choices made by similar projects. This model could effectively circumvent our second problem by presenting a *series* of recommendations, determined by the most similar project (or, with some calculation, a series of the similar projects). By basing recommendations on textual similarities, we can set up a functioning data model at a low start-up cost. Accuracy of such a model would become an issue. One additional feature of recommender systems, however, relevance feedback, could allow us to address our first problem by *building evidence over time*.

Recommender systems often allow for an iterative process where users offer feedback on recommendations, which is then used to refine the produced results. With slight enhancements, this feedback could be factored into the underlying data model. When presented with recommendations, a user could supplant their *yes* or *no* verdicts with structured feedback on *why* or *why not*. In addition to reshaping results in the current usage cycle, this feedback could also enhance the model itself—adding evidence and context to the recommendations offered to the system. At the end of each feedback cycle, the user-input query information could even be added to the body of data stored in the model.

We propose that the transformation of static predictive models into dynamic community-driven models could assist not only with our goal of making high-level development decisions, but with many of the data problems in software engineering. Relevance feedback could lead to lower initial costs, as smaller amounts of data are needed to seed a model. Over time, these models could be continually updated with new examples and revisions based on practitioner feedback. This allows us to address data quality issues, collect new measurements, and filter out irrelevant factors.

Therefore, we believe that a future research direction in AI-based software engineering will be the usage of feedback—not just in recommender systems, but in many AI techniques—to improve and reshape both the produced analytical results and the underlying prediction models.

2. EXAMPLE: RECOMMENDER SYSTEMS

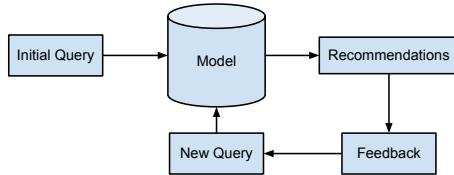


Figure 1: Model of a typical recommender system.

Recommender systems are information filtering systems that run a user-supplied query against a predictive model and offer new items of potential interest [7]. These systems can be broadly classified into one of two forms. The first is a *content-based* recommender system, which takes in a query and recommends items with similar features [11]. One example of a content-based system is the Pandora radio program, which seeds the data model with a chosen song or musician and then plays tracks deemed to be similar in musical style to the initially seeded item. The other common type of recommender system utilizes *collaborative* filtering—recommendations are made based on the actions taken by other users of the system. This can be seen on Amazon.com, where users that look at a particular item will see additional recommended purchases based on what other users bundled with that item. These two types of systems are not mutually exclusive—many recommender systems filter content-based recommendations based on the actions of other users. The video streaming service Netflix recommends movies based both on similarity to what you watched and the viewing habits of other users.

Recommender systems are already in use in software engineering research. An entire workshop was devoted to work on this topic recently, with demonstrations of systems that recommended design patterns given project characteristics [9] or elements in model libraries that could be reused to save coding effort [5]. One of the authors of this report previously created a recommender system for locating faults based on linking bug reports with source code [1].

Earlier, we expressed a desire for high-level recommendations on what practices to employ at various stages of development. One could envision an approach to this problem taking the form of a recommender system. At the heart of such a system could be a data model containing (ideally) dozens to hundreds of entries pairing corpora of data on past software projects (descriptions, requirements documents, design specifications, or even source code) with lists of the practice decisions made throughout development. For instance, in the testing stage, a few of these decisions may include program structure choices (e.g. whether to separate compound Boolean decisions into intermediate expressions), test suite size, test generation technique, test adequacy metric, or what percentage of the variable space to monitor during test execution.

This data model differs from many common ones slightly—it may not contain measured attributes justifying each of the individual decisions (although, if available, it certainly could). Project similarity could be calculated instead by comparing the corpora of project information. Such a model allows for two things: (1) a broader, more generalized view of the data space and (2) a lower start-up cost—initial recommendations can be made without much prior knowledge of the factors leading to decisions and, if recommendations are presented based solely on the *most similar* project, without needing to consider the dependence of one decision on another (if solutions are based on multiple similar projects, more care will need to go into recommendations). Indeed, such a model may be more palatable to privacy-conscious data sources; they could offer a list of recommendations along with as little or as much project information as they wish (of course, more data will generally lead to better solutions).

The operation of this recommender system (and of recommender systems in general) is visualized in Figure 1. The user inputs an initial set of information relating to the project under development (a description, solicited requirements, domain-specific rules, tool decisions mandated by business rules or previous development, etc). This input is transformed into a weighted vector of key terms and passed into the data model, which would output practice recommendations based on the decisions made during the development of similar projects.

The next step is of particular interest; the user can offer feedback on these recommendations. This typically comes in the form of a positive or negative response (either a binary choice or a numeric scale). This feedback is used to *alter* the user’s original query, employing relevance filtering algorithms such as Rocchio [6] to increase or decrease the weights or add and remove terms. This process produces new recommendations, calculated using the altered query. This feedback and recommendation loop continues until the user is satisfied with the offered solutions.

2.1 Enhancing Relevancy Feedback

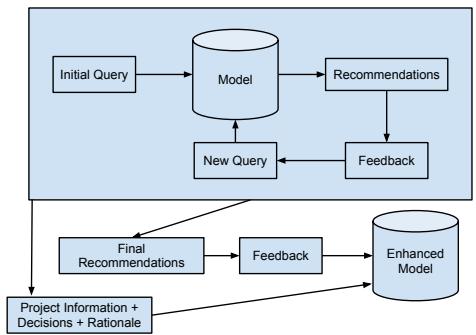


Figure 2: Model of an enhanced recommender system.

Although the data model proposed in the previous section allows for lower initial costs, as predictions are based on textual similarities rather than concrete knowledge of individual factors, the produced recommendations may not be accurate. Even if the recommended practices were successful in projects with similar goals, designs, or product domains, other internal conditions may require the use of different tools. Providing feedback to the recommender system in the standard “yes” or “no” format could improve accuracy, but such recommendations would still lack anything more than anecdotal evidence.

An extension to the relevancy feedback loop pictured in Figure 1 could transform the underlying static prediction model into a dynamic one where evidence is grown over time to support the produced recommendations. This enhanced loop is visualized in Figure 2.

In this new usage model, a user does not simply provide positive or negative responses to recommendations. Instead, they reply with data on why each attempted recommendation would work or why it wouldn’t. This feedback would need a standardized structure. The correctness verdict (yes or no) could be accompanied by suggestions for measurements that contributed to that verdict as well as values for a series of data attributes already suggested by other users—in the form of the numeric measurements or textual responses familiar to users of standard data sets. The user could additionally provide feedback on the existing attributes.

A simple example can be seen in Table 1 where the recommender system suggests the use of MC/DC as a test suite adequacy metric. The user states that this was a relevant suggestion, and fills in values for two existing evidence-providing attributes. The user also suggests the measurement of the ratio of Boolean expressions to numeric calculations as

Table 1: Feedback Sample

Recommendation	Relevant	Num. Boolean Expressions	Num. Numeric Calculations	New Attribute Suggestions and Values
MC/DC for Test Suite Adequacy	Yes	219	73	Ratio of Boolean to Numeric, 3:1

another potentially helpful attribute in making the recommendation of a test suite adequacy metric.

This feedback could, of course, still be used in the typical manner—reshaping the user’s query to the existing predictive model. Once the recommendation cycle is finished, however, the user’s feedback could be factored back into the predictive model, through a combination of automatic and manual techniques, as new evidence for the model data. With consent, the project data and chosen practices provided by the user could also be added into this enhanced data model.

In essence, this creates small predictive models underneath each of the individual recommendations that the overall data model would offer—models that could be mined to provide backing evidence for why this recommendation would be the correct choice to make. Rather than the standard static data set, this represents a model that can evolve over time in both quantity of data and the quality of the provided results.

3. IMPLICATIONS AND CHALLENGES

The use of feedback mechanisms to transform static prediction models into evolving ones is exciting for multiple reasons. First, it allows for lower start-up costs. Even if the data attributes to study are not well-known, a data model could be set up connecting outcomes to generalized information about software projects (things like descriptions, product domain rules, and design documents). Even if such models are not initially accurate, they have the potential to improve as new evidence is added.

Second, such models could help solve the data quality and quantity issues that are brought up frequently in AI research. Unhelpful data attributes could be automatically removed over time as negative reports mount, lessening the need for data preprocessing. New attributes will be added, creating additional opportunities for model analysis. User data could be imported, growing the number of records for others to consider.

The use of feedback to improve models could even lead to a Wikipedia-like effect on data repositories—where the continual improvements to data cause a continual increase in both the number of users and the number of edits made to the model. We envision a future where software development decisions aren’t just assisted by analysis of a few data records, but by powerful community-assisted prediction models.

There, of course, exist a number of challenges to solve in implementing the ideas presented in this report:

- While recommender systems offer the inspiration for these feedback mechanisms, such techniques are by no means limited to such systems. Research will need to be pursued on how to effectively build feedback into other AI approaches. This will likely necessitate a shift from data sets as static files that sit on a desktop to cloud-based models with accompanying feedback hooks. These models could even be connected into development environments, automatically collecting data and providing advice.
- If a user suggests new attributes, when should those be added to the model and presented to other users? Similarly, when should existing attributes that are deemed unhelpful be phased out? What measures should be taken to group together and standardize similar suggestions for new attributes?
- As new attributes and evidence is added to the data model, older data in the model may lack the amount or sophistication of data that newer projects contain. Steps will need to be taken to either update older data, account for missing data, or outright remove records that

becomes less useful. Previous contributors should be frequently invited to submit revisions to their data, and some amount of manual editing on the part of data-set retainers may be necessary.

- There is the additional problem of convincing users to actually provide feedback. Academic researchers can, to some extent, be convinced to contribute by the promise that others will also make contributions, creating a stronger pool of accessible data. Industrial practitioners will need to be convinced that the effort spent providing feedback will result in a worthwhile return on investment.

4. SUMMARY

We propose that there are a class of problems—such as deciding on a series of development practices — that challenge many AI approaches for two reasons: (1) well-defined data is necessary, but it is not clear which factors are important and (2) multiple recommendations must be made, and dependence must be considered.

Recommender systems offer inspiration. First, they can make use of data models that broadly pair a series of recommendations with generalized information like project descriptions and design documents. Second, they offer feedback mechanisms to refine the calculated recommendations. Detailed feedback could be factored back into the data model to, over time, build evidence and context for recommendations.

Existing algorithms and data models would be amenable to the addition of feedback mechanisms, and the use of these dynamic models could lower start-up costs and generate more accurate results as the model grows. We believe that feedback-driven dynamic prediction models will become an exciting research topic in the field of AI-based software engineering.

5. REFERENCES

- [1] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 351–360, sept. 2009.
- [2] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Automated Software Engg.*, 17(4):439–468, Dec. 2010.
- [3] M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
- [4] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, pages 226–247, 2009.
- [5] L. Heinemann. Facilitating reuse in model-based development with context-dependent model element recommendations. In *Third International Workshop on Recommendation Systems for Software Engineering*, 2012.
- [6] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.
- [7] P. Melville and V. Sindhwani. Recommender systems. *Encyclopedia of Machine Learning*, 2010.
- [8] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. The promise repository of empirical software engineering data, June 2012.
- [9] F. Palma, H. Farzin, and Y.-G. Gueheneuc. Recommendation system for design patterns in software development: An DPR overview. In *Third International Workshop on Recommendation Systems for Software Engineering*, 2012.
- [10] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM, 2008.
- [11] F. Ricci, L. Rokach, and B. Shapira. *Recommender Systems Handbook*. Springer, 2011.
- [12] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Engg.*, 14(5):540–578, Oct. 2009.

Improving the Accuracy of Oracle Verdicts Through Automated Model Steering*

Gregory Gay, Sanjai Rayadurgam, Mats P.E. Heimdahl

Department of Computer Science & Engineering

University of Minnesota, USA

greg@greggay.com, [rsanjai,heimdahl]@cs.umn.edu

ABSTRACT

The *oracle*—a judge of the correctness of the system under test (SUT)—is a major component of the testing process. Specifying test oracles is challenging for some domains, such as real-time embedded systems, where small changes in timing or sensory input may cause large behavioral differences. Models of such systems, often built for analysis and simulation, are appealing for reuse as oracles. These models, however, typically represent an *idealized* system, abstracting away certain issues such as non-deterministic timing behavior and sensor noise. Thus, even with the same inputs, the model’s behavior may fail to match an acceptable behavior of the SUT, leading to many false positives reported by the oracle.

We propose an automated *steering* framework that can adjust the behavior of the model to better match the behavior of the SUT to reduce the rate of false positives. This *model steering* is limited by a set of constraints (defining acceptable differences in behavior) and is based on a search process attempting to minimize a dissimilarity metric. This framework allows non-deterministic, but bounded, behavior differences, while preventing future mismatches, by guiding the oracle—within limits—to match the execution of the SUT. Results show that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing false positives and, consequently, development costs.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Verification

Keywords

Software Testing, Test Oracles, Model-Based Testing

*This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE’14, September 15–19, 2014, Västerås, Sweden.

Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...\$15.00.
<http://dx.doi.org/10.1145/2642937.2642989>.

1. INTRODUCTION

When running a suite of tests, the *test oracle* is the judge that determines the correctness of the execution of a given system under test (SUT). Despite increased attention in recent years, the *test oracle problem* [16]—a set of challenges related to the construction of efficient and robust oracles—remains a major problem in many domains. One such domain is that of real-time process control systems—embedded systems that interact with physical processes such as implanted medical devices or power management systems. Systems in this domain are particularly challenging since their behavior depends not only on the values of inputs and outputs, but also on their time of occurrence [9]. In addition, minor behavioral distinctions may have significant consequences [18]. When executing the software on an embedded hardware platform, several sources of non-determinism, such as input processing delays, execution time fluctuation, and hardware inaccuracy, can result in the SUT non-deterministically exhibiting varying—but acceptable—behaviors for the same test case.

Behavioral models [21], typically expressed as state-transition systems, represent the system requirements by prescribing the behavior (the system state) to be exhibited in response to given input. Common modeling tools in this category are Stateflow [22], Statemate [15], and Rhapsody [1]. Models built using these tools are used for many purposes in industrial software development and, thus, their reuse as a test oracle is highly desirable. These models, however, provide an abstract view of the system that typically simplifies the actual conditions in the execution environment. For example, communication delays, processing delays, and sensor and actuator inaccuracies may be omitted. Therefore, on a real hardware platform, the SUT may exhibit behavior that is acceptable with respect to the system requirements, but differs from what the model prescribes for a given input; the system under test is “close enough” to the behavior described by the model. Over time, these differences can build to the point where the execution paths of the model and the SUT diverge enough to flag the test as a “failure,” even if the system is still operating within the boundaries set by the requirements. In a rigorous testing effort, this may lead to tens of thousands of false reports of test failures that have to be inspected and dismissed—a costly process.

One solution would be to filter the test results on a step-by-step basis—checking the state of the SUT against a set of constraints and overriding the oracle verdict as needed. However, filter-based approaches are *inflexible*. While a filter may be able to handle isolated non-conformance between the SUT and the oracle model, it will likely fail to account for behavioral divergence that builds over time, growing with each round of input. Instead, we take inspiration for addressing this model-SUT mismatch problem from *program steering*, the process of adjusting the execution of live

programs in order to improve performance, stability, or correctness [23]. We hypothesize that behavioral models can be adapted for use as oracles for real-time systems through the use of *steering actions* that override the current execution of the model [10, 31]. By comparing the state of the model-based oracle (MBO) with that of the SUT following an output event, we can guide the model to match the state of the SUT, as long as a set of constraints defining acceptable deviation are met. Unlike a filter, steering is *adaptable*, adjusting the live execution of the model—within the space of legal behaviors—to match the execution of the SUT. The result of steering is a widening of the behaviors accepted by the oracle, thus compensating for allowable non-determinism, without unacceptably impairing the ability of the model-based oracle to correctly judge the behavior of the SUT.

We present an automated framework for comparing and steering the model-based oracle with respect to the SUT, building on ideas first proposed in [10]. We detail the implementation of the framework, and assess its capabilities on a model for the control software of an patient controlled analgesia pump (a medical infusion pump)—a real-world systems with complex, time-based behaviors. Case study results indicate that steering improves the accuracy of the final oracle verdicts—outperforming both default testing practice and step-wise filtering. Oracle steering successfully accounts for within-tolerance behavioral differences between the model-based oracle and the SUT—eliminating a large number of spurious “failure” verdicts—with minimal masking of real faults. By pointing the developer towards behavior differences more likely to be indicative of real faults, this approach has the potential to lower testing costs and reduce development effort.

2. BACKGROUND

There are two key artifacts necessary to test software, the *test data*—inputs given to the system under test—and the *test oracle*—a judge on the resulting execution [17, 32]. A *test oracle* can be defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [16].

The most common form of test oracle is a *specified oracle*—one that judges behavioral aspects of the system under test with respect to some formal specification [16]. Commonly, such an oracle checks the behavior of the system against a set of concrete expected values [30] or behavioral constraints (such as assertions, contracts, or invariants) [7]. However, specified oracles can be derived from many other sources of information; we are particularly interested in using behavioral models, such as those often built for purposes of simulation, analysis and testing [21].

A common practice during the development and testing of software is to build models of the intended behavior of the final system. Although such models are useful at all stages of the development process—particularly during requirements analysis—they are particularly useful for addressing two problems in testing: (1) models allow analysis and testing activities to begin before the actual implementation is constructed, and (2) models are suited to the application of verification and automated test generation techniques that allow us to cover a larger class of scenarios than we can cover manually [26]. As such, models are often executable; thus, in addition to serving as the basis of test generation [21], models can be used as a source of expected behavior, that is, used as a test oracle.

Executable behavioral models can be divided into *declarative* models created in formal specification languages, and *constructive* models built with state-transition languages such as Simulink and Stateflow, Statemate, finite state machines, or other automata structures [21]. Presently, we focus our work on constructive state-

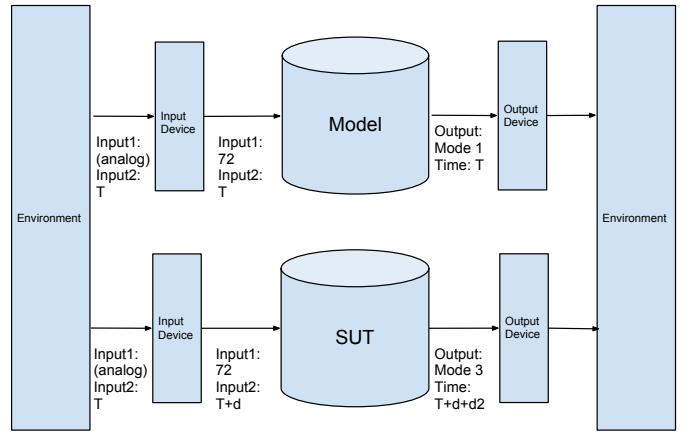


Figure 1: Illustration of abstraction induced behavioral differences between the model and the system under test.

transition systems since these are frequently used to model real-time control systems—software that monitors and interacts with a physical environment [18].

Non-determinism is a major concern in embedded real-time systems. The task of monitoring the environment and pushing signals through multiple layers of sensors, software, and actuators can introduce points of failure, delay, and unpredictability. Input and observed output values may be skewed by noise in the physical hardware, timing constraints may not be met with precision, or inputs may arrive faster than the system can process them. Often, the system behavior may be acceptable, even if the system behavior is not exactly what was captured in the model—a model that, by its very nature, incorporates a simplified view of the problem domain. A common abstraction when modeling is to omit any details that distract from the core system behavior in order to ensure that analysis of the models is feasible and useful. Yet these omitted details may manifest themselves as differences between the behavior defined in the model and the behavior observed in the implementation.

To give an example of how these differences manifest themselves, consider the model and a corresponding system depicted in Figure 1. A common abstraction when designing a model is to assume that all actions take place instantly, ignoring the reality of *computation time*. Both the model and the actual implementation receive the same stimuli from the physical environment, both process that input through the system, and then both issue output back to the environment. In the model, all actions are considered to have taken place at time step T . However, in the system, computations take time to be completed, and the processing of environmental stimulus through hardware layers and software subsystems adds an additional delay to the time at which the system receives that input. By the time that the system finishes responding to the stimulus, at time $T + (2 \text{ delay periods})$, it is unlikely that the output fed to the environment matches the output that the model issued (in this case, not only do the output timestamps not match, but the SUT made a mode selection that is entirely different from that of the model). In systems such as pacemakers or infusion pumps, time is a crucially important piece of data, and delays can lead to a behavior that is very different from the one produced by a model that abstracts such delays. Furthermore, such delays are commonly non-deterministic. Repeated application of the same stimulus may not result in the same output if processing time varies.

This raises the question—why use models as oracles? Alternative approaches could be to turn to an oracle based on explicit behavioral constraints—assertions or invariants—or to build declar-

ative behavioral models in a formal notation such as Modelica. These solutions, however, have their limitations. Assertion-based approaches only ensure that a limited set of properties hold at *particular points in the program* [7]. Further, such oracles may not be able to account for the same range of testing scenarios as a model that prescribes behavior for all inputs. Declarative models that express the computation as a theory in a formal logic allow for more sophisticated forms of verification and can better account for time-constrained behaviors [11]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [24]. Constructive models are visually appealing, easy to analyze without specialized knowledge, and suitable for analyzing failure conditions and events in an isolated manner [11]. The complexity of declarative models and the knowledge needed to design and interpret such models make widespread industrial adoption of the paradigm unlikely.

While there are challenges in using constructive model-based oracles, it is a widely held view that such models are indispensable in other areas of development and testing, such as requirements analysis or automated test generation [26, 20]. From this standpoint, the motivational case for models as oracles is clear—if these models are already being built, their reuse as test oracles could save significant amounts of time and money, and allow developers to automate the execution and analysis of a large volume of test cases. Practitioners are well-versed in these models and so these are likely to be less error-prone compared to building a new unfamiliar type of oracle. Therefore, we seek a way to use constructive model-based oracles that can handle the non-determinism introduced during system execution on the target hardware.

3. ORACLE STEERING

In a typical model-based testing framework, the test suite is executed against both the SUT and the model-based oracle, and the values of certain variables are recorded to a trace file after each execution step. The oracle’s comparison procedure examines those traces and issues a verdict for each test (*fail* if test reveals discrepancies, *pass* otherwise). When testing a real-time system, we would expect non-determinism to lead to behavioral differences between the SUT and the model-based oracle during test execution. The actual behaviors witnessed in the SUT may not be incorrect—they may still meet the system requirements—they just do not match what the model produced. We would like the oracle to distinguish between correct, but non-conforming behaviors introduced by non-determinism and behaviors that are indicative of a fault.

One approach to address this would be to augment the comparison procedure with a filtering mechanism to detect and discard acceptable differences on a per-step basis. For example, to address the simple computation-time abstraction in Figure 1, a filter could simply allow any timestamp within a certain range. However, the issue with filtering on a per-step basis is that the effect of non-determinism may linger on for several steps, leading to irreconcilable differences between the SUT and the model-based oracle. Filters may not be effective at handling growing behavioral divergence. If the time of input or output impacts behavior, such as in the case of a pacemaker—a system where even a single delayed input may impact all future commanded paces—a filter is unlikely to make an accurate judgement after the first few comparisons.

To address this problem, we take inspiration from *program steering*—the process of adjusting the execution of live programs in order to improve performance, stability, or behavioral correctness [12]. Instead of steering the behavior of the SUT, however, we *steer the oracle* to see if the model is capable of matching the SUT’s behavior. When the two behaviors differ, we backtrack and apply a

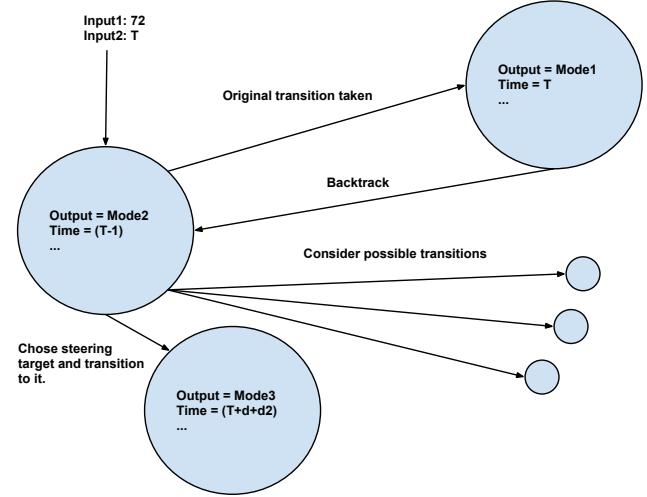


Figure 2: Illustration of steering process.

steering action—e.g., adjust timer values, apply different inputs, or delay or withhold an input—that changes the state of the model-based oracle to a state more similar to the SUT (as judged by a dissimilarity metric). Oracle steering, unlike filters, is *adaptable*. Such actions provide flexibility to handle non-determinism, while still retaining the power of the oracle as an arbiter. Of course, improper steering can bias the behavior of the model-based oracle, masking both acceptable deviations and actual indications of failures. Nevertheless, we believe that by using a series of constraints it is possible to sufficiently bound steering so that the ability to detect faults is still retained.

First, a set of **tolerance constraints** governing the allowable changes to certain variables (input, internal, or output) in the model that can be affected by steering. These constraints define bounds on the non-determinism or behavioral deviation that can be accounted for with steering. For example, a pacemaker takes as input timestamped sensed cardiac events and responds by setting the time of the next commanded pace (the pulse delivered to the heart). Then, if there are no additional sensed events within that time frame, the pacemaker will issue an electrical pulse to the appropriate chamber of the heart. If the model acts on the input after a slight delay and the model acts instantly, we might allow steering to adjust the time stamp on an input by up to four milliseconds. However, we might forbid steering from changing the pacemaker’s response to that input (i.e., if the pacemaker responds with a status “event acknowledged and acted on”, we might not allow the response to be changed to “premature sensed event”). If a pace command occurs at a different time in the system than it does in the model, we might—within a similar time window—allow steering to attempt to adjust the time that the pace command occurs.

Second, a **dissimilarity function** $Dis(model\ state, SUT\ state)$, that compares the state of the model to the observable state of the SUT. We seek a minimization of $Dis(s_m^{new}, s_{sut}) < Dis(s_m, s_{sut})$. That is, within the bounds on the search space set by the tolerance constraints, we seek the candidate solution with the lowest dissimilarity score to the state of the SUT. There are many different functions that can be used to calculate dissimilarity. Cha provides a good primer on the calculation of dissimilarity [6].

Third, a further set of general **policy decisions** on when to steer. For example, one might decide not to steer unless $Dis(s_m^{new}, s_{sut}) = 0$ —that is, one might decide not to steer at all unless there ex-

```

1. for test in Tests
2.   for step in test
3.     initialVerdict = Dis(Sm, Ssut)
4.     if initialVerdict > 0
5.       oldState = Sm
6.       targetState=searchForNewState(Model,Sm,Ssut,Constraints,Dis())
7.       while Dis(targetState, Ssut) < Dis(oldState, Ssut)
8.         oldState = targetState
9.         targetState=searchForNewState(Model,Sm,Ssut,Constraints,Dis())
10.        transitionModel(Model,targetState)

```

Figure 3: Steps in the Steering Process

ists a steering action that results in a model state identical to that observed in the SUT.

In other words, the new state of the model-based oracle following the application of a steering action must be a state that is possible to reach within a limited number of transitions from the current state of the model, must fall within the boundaries set by the tolerance constraints, and must minimize the dissimilarity function.

We can illustrate steering using the system depicted in Figure 1. This system takes as input some environmental factor and a time stamp on when the reading was taken. It then outputs a mode choice and a time stamp on when the mode choice is issued. The model has abstracted computation time, and as a result, the model issues a mode choice and time stamp that differs from the SUT (the SUT is subjected to the time delay of the initial input to pass from the environment to the software and the time to perform computations). Although there is clearly non-conformance between the model-based oracle and the SUT, the SUT may still be operating within the bounds of the system requirements. Thus, as depicted in Figure 2, we can attempt to steer the model-based oracle.

As outlined in Figure 3, after obtaining our initial verdict by calculating the dissimilarity function (i.e., the Euclidean distance between the state of the SUT and the state of the model-based oracle), the steering procedure would backtrack to the state of the model before receiving the input $Input1 = 72$, $Input2 = T$ and evaluate the set of possible transitions from that state through the use of a search algorithm. The set of candidate solutions is limited by the set of tolerance constraints—for example, we might allow $Input1 \leq Input1^{new} \leq (Input1 + 5)$ (the value $Input1^{new}$ may fall up to 5 seconds after the original value of $Input1$). Then, the search selects from the remaining candidates through the use of the dissimilarity function. Finally, the candidate solution that results in the lowest observed value of $Dis(s_m^{new}, s_{sut})$ is chosen as the new state of the model-based oracle. Execution then resumes from the chosen state.

We have implemented the basic steering approach outlined in Figure 3. Our search process is based on SMT-based bounded model checking [13], which is a natural choice for this problem. We have a series of constraints that govern steering actions and seek to locate a model state reachable in a limited number of transitions that satisfies those constraints and minimizes a dissimilarity metric. Specifically, we make use of the Kind model checker [13] and the Z3 constraint solver [8].

A solution to the constraint $Dis(s_m^{new}, s_{sut}) < Dis(s_m, s_{sut})$ would give us a model state that is more similar to the behavior of the SUT than the original transition taken by the model-based oracle, but carries no guarantee that the satisfying state minimizes the dissimilarity metric. Thus, we first check the constraint $Dis(s_m^{new}, s_{sut}) = 0$; then, if an exact match cannot be found and if the steering policy allows inexact matches, we apply the model

checker with this constraint in order to get an initial threshold, then iteratively reapply the model checker with new thresholds until we can no longer find a better solution. The best solution found then becomes the new state of the model-based oracle.

3.1 Selecting Constraints

The efficacy of the steering process depends heavily on the tolerance constraints and policies employed. If the constraints are too strict, steering will be ineffective—leaving as many “false failure” verdicts as not steering at all. On the other hand, if the constraints are too loose, steering runs the risk of covering up real faults in the system. Therefore, it is crucially important that the constraints to be employed are carefully considered.

Often, constraints can be inferred from the system requirements and specifications. For example, when designing an embedded system, it is common for the requirements documents to specify a desired accuracy range on physical sensors. If the potential exists for a model-system mismatch to occur due to a mistake in reading input from a sensor, than it would make sense to take that range as a tolerance constraint on that sensor input and allow the steering algorithm to try values within that range of the canonical test input.

We recommend that users err toward strict constraints. While it is undesirable to spend time investigating failures that turn out to be acceptable, that outcome is preferable to masking real faults. Steering will not fully account for a model that produces incorrect behavior, so steering should start with a mature, verified model.

To give an example, consider a pacemaker. The pacemaker might take as input a set of prescription values, event indicators from sensors in the patient’s heart, and timer values. We would recommend that steering be prohibited from altering the prescription values at all, as manipulation of those values might cover faults that could threaten the life of a patient. However, as electrical noise or computation delays might lead to issues, steering should be allowed to alter the values of the other inputs (within certain limits). The system requirements might offer guidance on those limits—for instance, specifying a time range from when a pace command is supposed to be issued to when it must have been issued to be acceptable. This boundary can be used as to constrain the manipulation of timer variables. Furthermore, given the critical nature of a pacemaker, a tester might also want to employ a policy where steering can only intervene if a solution can be found that identically matches the state of the SUT.

Unlike approaches that build nondeterminism into the model, steering decouples the specification of nondeterminism from the model. This decoupling allows testers more freedom to experiment with different sets of constraints and policies. If the initial set of constraints leaves false failure verdicts or if testers lack confidence in their chosen constraints, alternative options can easily be explored by swapping in a new constraint file and executing the test suite again. Using the dissimilarity function to rate the set of final test verdicts, testers can evaluate the severity and number of failure verdicts remaining after steering with each set of constraints and gain confidence in their approach.

3.2 Automated Testing Framework

In a typical testing scenario that makes use of model-based oracles, a test suite is executed against both the system under test and the behavioral model. The values of the input, output, and select internal variables are recorded to a *trace file* at certain intervals, such as after each discrete cycle of input and output. Some comparison mechanism examines those trace files and issues a verdict for each test case (generally a *failure* if any discrepancies are detected and a *pass* if a test executes without revealing any differences between

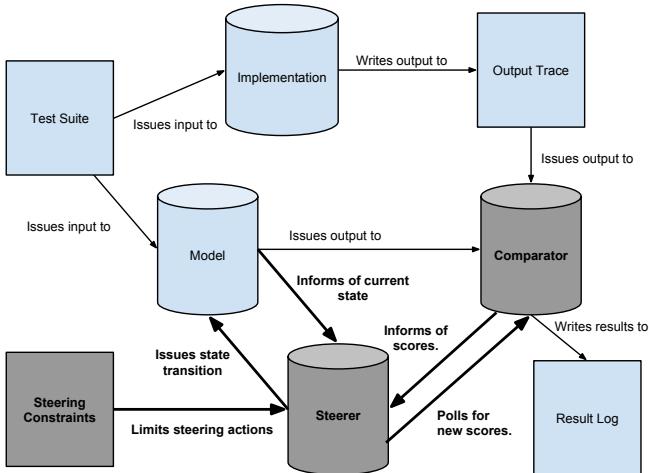


Figure 4: An automated testing framework employing steering.

the model and SUT). As illustrated in Figure 4, such a framework can be modified to incorporate automated oracle steering.

No matter the size of the model, steering will add additional overhead to the time required to execute tests. In practice, depending on the strictness of the constraints employed and the complexity of the model, execution with steering can take anywhere between a few additional seconds of execution to a few additional minutes. Therefore, we recommend that testing take place *offline*, rather than attempting to run the model and system in lockstep.

A framework would follow these steps:

1. Execute each test against the system under test, logging the values of select variables at each test step.
2. Execute each test against the model-based oracle, and for each step of the test:
 - (a) Feed input to the oracle model.
 - (b) Compare the model output to the SUT output.
 - (c) If the output does not match, the steering algorithm will interface with the model, backtracking execution and attempting to steer the model’s execution within the specified constraints (perhaps consulting dissimilarity metrics used by the comparison module)
 - (d) Compare the new output of the model to the SUT output and log the new dissimilarity score.
3. Issue a final verdict for the test.

4. RELATED WORK

Several authors have examined the use of behavioral models as test-generation targets for real-time systems [4, 20, 9, 29, 5]. If such models are used to generate tests, they can implicitly serve as a test oracle. For example, Larsen et al. [20] model a system as a non-deterministic timed automata that is constrained by an environment model. The combined model serves as an oracle during test execution. Arcuri et al. also examine the impact of the environment when testing process-control systems [4]. Their framework, which only models the environment that the system interacts with and eschews the system itself, allows limited non-determinism in the time that an action can take place, as well as certain forms of hardware-related non-determinism (through tester-provided probabilities of hardware failure). Savor and Seviora proposed a framework where the behavior of the SUT is compared to expected behaviors produced by a finite state machine derived from the system

requirements [29]. Their framework can handle non-determinism in process communication by appending signals with an interval on the time of occurrence. This interval is used to construct alternative legal event orderings. Briones and Brinksma treat quiescence—the lack of system output—as a special form of system output, and thus, offer a behavioral model that is able to capture a range of non-deterministic response times from the system under test [5].

While the above approaches consider forms of non-determinism, there are a few key differences with our proposed approach since it decouples the model from the rules governing steering. This decoupling makes non-determinism implicit and the approach more generally applicable. Explicitly specified non-deterministic behavior would limit the scope of non-determinism handled by the oracle to what has been planned for by the developer and subsequently modeled. It is difficult to anticipate the non-determinism resulting from deploying software on a hardware platform, and, thus, such models will likely undergo several revisions during development. Steering instead relies on a set of rule-based constraints that may be easier to revise over time. Additionally, by not relying on a specific model format, steering can be made to work with models created for a variety of purposes.

Oracle steering is conceptually similar to *dynamic* program steering, the automatic guidance of program execution [12, 23]. Much of the research in dynamic program steering is concerned with automatic adaptation to maintain consistent performance or a certain reliability level when faced with depleted computational resources [12]. However, Kannan et al. have proposed a framework to assure the correctness of software execution at runtime through corrective steering actions [19]. Their framework serves as a supervisor on program execution, checking observed behavior at runtime against a set of assertions and adjusting the behavior of the program whenever an assertion is violated. This is done under the assumption that the SUT is mostly correct, and that only minimal control should be exercised. Although their system bears similarities to what we are proposing, our goals are very different—rather than adjusting the behavior of the live system, we apply steering to the test oracle in order to better identify fault-indicative behaviors.

More relevant to our work is Microsoft Research’s Spec Explorer [31] test generation framework. Their framework explores the possible runs of the executable model by applying steering actions to the model and guiding it through various execution scenarios with the goal of systematically generating test suites. It can then use the model as an oracle for the generated test by checking whether the SUT produces the same behaviors, steering through different execution scenarios for test generation as opposed to adjudging system execution. Although Spec Explorer also steers the actions of a behavioral model, their application and goals greatly differ from ours. They use steering to create tests, and do not apply it when checking conformance. Therefore, their framework cannot be used to address the instances of non-conformance related to non-deterministic execution in this report.

5. CASE STUDY

We aim to gain an understanding of the capabilities of oracle steering and the impact it has on the testing process—both positive and negative. Thus, we pose the following research questions:

1. To what degree does steering lessen behavioral differences that are legal under the system requirements?
2. To what degree does steering mask behavioral differences that fail to conform to the requirements?
3. Are there situations where a filtering mechanism is more appropriate than actively steering the oracle, and vice-versa?

5.1 Experimental Setup Overview

We have based our model-based oracle on the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [25]. This subsystem takes in a prescription for a drug—as well as several sensor values—and determines the appropriate dosage of the drug to be administered to a patient each second over a given period of time. This case example, developed in the Simulink and Stateflow notations and translated into the Lustre synchronous programming language [14], is a complex real-time system of the type common in the medical device domain. Details on the size of the Simulink model and the number of lines of code in the translated Lustre code are provided in Table 1.

Table 1: Case Example Information

	# States	# Transitions	Lustre LOC
Infusion	23	50	6299

To evaluate oracle steering, we performed the following:

1. **Generated system implementations:** We approximated the behavioral differences expected from systems running on real embedded hardware by creating alternate versions of the model with non-deterministic timing elements. We also generated 50 mutated versions of both the oracle and each “SUT” with seeded faults (Section 5.2).
2. **Generated tests:** We randomly generated 100 tests for each case example, each 30 test steps in length (Section 5.2).
3. **Set steering constraints:** We constrained the variables that could be adjusted through steering and the values that those variables could take on, and established dissimilarity metrics to be minimized (Section 5.3).
4. **Assessed impact of steering:** For each combination of SUT, test, and dissimilarity metric, we attempted to steer the oracle to match the behavior of the SUT. We compare the test results before and after steering and evaluate the precision and recall of our steering framework, contrasted against the general practice of not steering and a step-by-step filtering mechanism (Section 5.4).

5.2 System and Test Generation

To produce “implementations” of the example system, we created alternative versions of the model, introducing realistic non-deterministic timing changes to the systems. We built (1) a version of the system where the exit of the patient-requested dosage period may be delayed by a short period of time, and (2) a version of the system where the exit of an intermittent increased dosage period (known as a square bolus dose) may be delayed. These changes are intended to mimic situations where, due to hardware-introduced computation delays, the system remains in a particular dosage mode for longer than expected.

For the original model and the “system under test” variants, we have also generated 50 *mutants* (faulty implementations) by introducing a single fault into each model. This ultimately results in a total of 152 SUT versions—two versions with non-deterministic timing behavior, fifty versions with faults, and one hundred versions with both non-deterministic timing and seeded faults (fifty per timing variation).

The mutation testing operators used in this experiment include changing an arithmetic operator, changing a relational operator, changing a boolean operator, introducing the boolean \neg operator, using the stored value of a variable from the previous computational cycle, changing a constant expression by adding or subtracting 1 from int and real constants (or by negating boolean constants),

and substituting a variable occurring in an equation with another variable of the same type. The mutation operators used are discussed at length in a previous report on empirical software testing research [28], and are similar to those used by Andrews et.al, where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [3].

Using a random testing algorithm, we generated 100 tests, each thirty test steps long. Each test represents thirty seconds of system activity—a length appropriate to capture a relevant range of time-sensitive behaviors, but still short enough to yield a reasonable experiment cost. These tests were then executed against each model in order to collect traces. In the models with timing fluctuations, we controlled those fluctuations through the use of an additional input variable. The value for that variable was generated non-deterministically, but we used the same value across all systems with the same timing fluctuation. As a result, we know whether a resulting behavioral mismatch is due to a seeded timing fluctuation or a seeded fault in the system. Using this knowledge, we labeled each test that fails pre-steering as failing due to an “acceptable timing deviation”, an “unacceptable timing deviation”, or a “seeded fault.”

5.3 Steering Constraints

For this particular case study, we have specified the tolerance constraints on steering in terms of limits on the adjustment of the input variables of the system (our model-based oracle steering framework allows constraints to be placed on internal or output variables as well). The chosen tolerance constraints include:

- Five of the input variables relate to timers within the system—the duration of the patient-requested bolus dose period, the duration of the intermittent square bolus dosage period, the lockout period between patient-requested bolus dosages, the interval between intermittent square bolus dosages, and the total duration of the infusion period. For each of those, we placed an allowance of $(CurrVal - 1) \leq NewVal \leq (CurrVal + 2)$. E.g., following steering, a dosage duration is allowed to fall within a three second period—between one second shorter and two seconds longer than the original prescribed duration.
- The remaining 15 input variables are not allowed to be steered.

These constraints reflect what we consider a realistic application of steering—we expect issues related to non-deterministic timing, and, thus, allow a small acceptable window around the behaviors that are related to timing. In this study, we do not expect any sensor inaccuracy, so we do not allow freedom in adjusting sensor-based inputs. Similarly, as these are medical devices that could harm a patient if misused, we do not allow any changes to the inputs related to prescription values.

In this experiment, we have made use of two different dissimilarity metrics when comparing a candidate state of the model-based oracle to the state of the SUT. The first is the Manhattan (or City Block) distance. Given vectors representing the state of the SUT and the model-based oracle—where each member of the vector represents the value of a variable—the dissimilarity between the two vectors can be measured as the sum of the absolute numerical distance between the state of the SUT and the model-based oracle:

$$Dis(s_m, s_{sut}) = \sum_{i=1}^n |s_{m,i} - s_{sut,i}| \quad (1)$$

The second is the Squared Euclidean distance. Given vectors representing the state, the dissimilarity between the vectors can be

measured as the “straight-line” numerical distance between the two vectors. The squared variant was chosen because it places greater weight on states that are further apart in terms of variable values.

$$Dis(s_m, s_{sut}) = \sum_{i=1}^n (s_{m,i} - s_{sut,i})^2 \quad (2)$$

A constant difference of 1 is used for differences between boolean variables or values of an enumerated variable. All numerical values are normalized to a 0-1 scale using predetermined constants for the minimum and maximum values of each variable.

We must choose a set of variables to compare when calculating a dissimilarity score (or oracle verdict). As we cannot assume that the internal variables of the SUT and the model are the same, we calculate similarity using the five output variables of the infusion pump: the commanded flow rate, the current system mode, the duration of active infusion, a log message indicator, and a flag indicating that a new infusion has been requested.

5.4 Evaluation

Using the generated artifacts—without steering—we monitored the outputs during each test, compared the results to the values of the same variables in the model-based oracle to calculate the dissimilarity score, and issued an initial verdict. Then, if the verdict was a failure ($Dis(s_m, s_{sut}) > 0$), we steered the model-based oracle, and recorded a new verdict post-steering. As mentioned above, the variables used in establishing a verdict are the five output variables of the system.

In Section 3, we stated that an alternative approach to steering would be to simply apply a filter on a step-by-step basis. We have implemented such a filter for the purposes of establishing a baseline to which we can compare the performance of steering. This filter compares the values of the output variables of the SUT to the values of those variables in the model-based oracle and, if they do not match, checks those values against a set of constraints. If the output—despite non-conformance to the model-based oracle—meets these constraints, the filter will still issue a “pass” verdict for the test. The filter will allow a test to pass if (despite non-conformance) values of the output variables in the SUT satisfy the following constraints:

- The current mode of the SUT is either “patient dosage” mode or “intermittent dosage” mode, and has not remained in that mode for longer than *prescribed duration* + 2 seconds.
- If the above is true, the commanded flow rate should match the prescribed value for the appropriate mode.
- All other output variables should match their corresponding variables in the oracle

As we expect a non-deterministic duration for the patient dosage and intermittent dosage modes (corresponding to the seeded issues in the SUT variants), this filter should be able to correctly classify many of the same tests that we expect steering to handle.

We compare the performance of the steering approach to both the filter and the default practice of accepting the initial test verdict. We can assess the impact of steering or filtering using the verdicts made before and after steering by calculating:

- The number of *true positives*—steps where an approach does not mask incorrect behavior;
- The number of *false positives*—steps where an approach fails to account for an acceptable behavioral difference;
- And the number of *false negatives*—steps where an approach does mask an incorrect behavior.

Table 2: Verdicts: T(true)/F(false), P(positive)/N(negative).

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	TN	FP
Fail (Due to Timing, Within Tolerance)	TN	FP
Fail (Due to Timing, Not in Tolerance)	FN	TP
Fail (Due to Fault)	FN	TP

Table 3: Initial test results when performing no steering or filtering. Raw number of test results, followed by percent of total.

Verdict	Number of Tests
Pass	11364 (74.8%)
Fail (Due to Timing, Within Tolerance)	1438 (9.4%)
Fail (Due to Timing, Not in Tolerance)	268 (1.7%)
Fail (Due to Fault)	2230 (14.7%)

The testing outcomes in terms of true/false positives/negatives are listed in Table 2. Using these measures, we calculate the *precision*—the ratio of true positives to all positive verdicts—and *recall*—the ratio of true positives to true positives and false negatives:

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

$$Recall = \frac{TP}{TP + FN} \quad (4)$$

We also calculate the *F-measure*—the harmonic mean of precision and recall—in order to judge the accuracy of oracle verdicts:

$$F\text{-measure} = 2 * \frac{precision * recall}{precision + recall} \quad (5)$$

6. RESULTS AND DISCUSSION

When running all tests over the various implementations (containing either timing deviations or seeded faults as discussed in Section 5.2) using a standard test oracle comparing the outputs from the SUT with the outputs predicted by the model-based oracle (15,200 test runs), 11,364 runs indicated that the system under test passed the test (the SUT and model-based oracle agreed on the outputs) and 3,936 runs indicated that the test failed (the SUT and model-based oracle had mismatched outputs). In an industry application of a model-based oracle, the 3,936 failed test would have to be examined to determine if the failure was due to an actual fault in the implementation, an unacceptable timing deviation from the expected timing behavior, or an acceptable timing deviation that, although it did not match the behavior predicted by the model-based oracle, was within acceptable tolerances—a costly process. Given our experimental setup, however, we can classify the failed tests as to the cause of the failure: failure due to timing within tolerances, failure due to timing not in tolerance, and failure due to a fault in the SUT. This breakdown is provided in Table 3. As can be seen, 1,438 tests failed even though the timing deviation was within what would be acceptable—these can be viewed as false positives and a filtering or steering approach that would have passed these test runs would provide cost savings. On the other hand, the steering or filtering should not pass any of the 268 tests where timing behavior falls outside of tolerance or the 2,230 tests that indicated real faults.

Results obtained from the case study showing the effect of steering on oracle verdicts are summarized in Tables 4 and 5 respectively, for the two different distance metrics studied. The numbers

Table 4: Distribution of results for steering (Squared Euclidean dissimilarity). Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	11364 (74.8%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1286 (8.4%)	152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	43 (0.3%)	2187 (14.4%)

Table 5: Distribution of results for steering (Manhattan dissimilarity). Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	11364 (74.8%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1198 (7.9%)	240 (1.6%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	43 (0.3%)	2187 (14.4%)

Table 6: Distribution of results for step-wise filtering. Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11364 (74.8%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1286 (8.4%)	152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	1253 (8.2%)	977 (6.4%)

Table 7: Precision, recall, and F-measure values.

Technique	Precision	Recall	F-measure
No Steering	0.63	1.00	0.77
Filtering	0.89	0.50	0.64
Steering - Euclidean	0.94	0.98	0.96
Steering - Manhattan	0.91	0.98	0.94

are presented as laid out in Table 2. Testing outcomes are first categorized according to the initial verdicts as determined by the model-based oracle before steering; a “fail” verdict is further delineated according to its reason—a mismatch that is attributable to either an allowable timing fluctuation, or an unacceptable timing fluctuation or a fault. For each category, the post-steering verdict is presented as both a raw number of test outcomes and as a percentage of total test outcomes. Table 6 shows the corresponding data for the step-by-step filtering approach. Data from these tables lead to the precision, recall, and F-measure values, shown in Table 7, for the default testing scenario (accepting the initial oracle verdict), steering utilizing two different dissimilarity metrics, and filtering.

6.1 Allowing Tolerable Non-Conformance

According to Table 3, 11.1% of the tests (1,706 tests) initially fail due to timing-related non-conformance. Of those, 1438 tests (9.4% of the total) fall within the tolerances set in the requirements. Steering should result in a pass verdict for all those tests.

As the tables show, for both dissimilarity metrics, steering is able to account for almost all of the situations where non-deterministic timing affects conformance while both the model-based oracle and the implementation remain within the bounds set in the system specification. For example, in Table 4 we see that steering using

the square Euclidian distance would have correctly passed 1,286 tests where the timing deviation was acceptable—tests that without steering failed. Therefore, we see a sharp increase in precision over the default situation where no steering is employed (from 0.63 when not steering, to 0.94 for the Squared Euclidean metric, and to 0.91 for the Manhattan metric, see Table 7).

Where previously developers would have had to manually inspect the more than 25% of all the tests (the sum of all “Fail” Table 4) to determine the causes for their failures (system faults or otherwise), they could now narrow their focus to the roughly 17% that still result in failure post-steering. Particularly given the large number of tests in this study, this reduction represents a significant savings in time and effort, removing between 1,198 and 1,286 tests that the developer would have needed to inspect manually. Still, there were a small number of tests that steering should have been able to account for (152 for the Squared Euclidean metric and 240 for the Manhattan metric). The reason for the failure of steering to account for allowable differences can be attributed to a combination of three factors: the tolerance constraints employed, the dissimilarity metric employed, and design differences between the SUT and the model-based oracle.

First, it may be that the tolerance constraints were too strict to allow for situations that should have been considered legal. Constraints should be relatively strict—after all, we are overriding the nominal behavior of the oracle while simultaneously wishing to retain the oracle’s power to identify faults. Yet, the constraints we apply should be carefully designed to allow steering to handle these allowed non-conformance events. In this case, the chosen constraints may have prevented steering from acting.

Second, the dissimilarity metric does appear to play a small role in the effectiveness of steering. The Squared Euclidean metric resulted in higher precision than the Manhattan metric (0.94 vs 0.91), indicating that the former was better able to account for tolerable non-conformance. As there is no difference in recall between the two metrics, it appears that—for this case example—the use of the Manhattan metric results in a slightly more conservative steering process. With the Manhattan metric, the steering approach is more likely to refuse to steer. However, given the similarity in the performance of the two metrics, it appears that the set of constraints employed plays a more dominant role in determining the final outcome of steering.

The tolerance constraints applied reduce the space of candidate targets to which the oracle may be steered. We then use the dissimilarity metric to choose a “nearest” target from that set of candidates. Thus, the relationship between the constraints and the metric ultimately determines the power of the steering process. However, no matter how powerful steering is, there may be situations where differences in the internal design of the systems render steering either ineffective or incorrect. We base steering decisions on state-based comparisons, but those comparisons can only be made on the portion of the state variables common between the SUT and oracle model. For this experiment, we only compared the output variables. As a result, there may be situations where we should have steered, but could not, as the state of the SUT depended on internal factors not in common with the oracle. In general, as the oracle and SUT are both ultimately based on the same set of requirements, we believe that some kind of relationship can be established between the internal variables of both realizations. However, in some cases, the model and SUT may be too different to allow for steering in all allowable situations. The inability of steering to account for tolerable differences for at least some tests in this case study can likely be attributed to the changes made to the SUT versions of the models.

6.2 Masking of Faults

As steering changes the behavior of the oracle and can result in a new test verdict, the danger of steering is that it will mask actual faults in the system. Such a danger is concerning, but with the proper choice of steering policies and constraints, we hypothesize that such a risk can be reduced to an acceptable level.

In this case study, steering changed a fault-induced “fail” verdict to “pass” in forty-three tests. This is a relatively small number—only 0.3% of the 15,200 tests. Although loss in recall is cause for concern when working with safety-critical systems, given the small number of incorrectly adjusted test verdicts, we hypothesize that it is unlikely for an actual fault to be entirely masked by steering on every test in which the fault would otherwise lead to a failure.

Just as the choice of tolerance constraints can explain cases where steering is unable to account for an allowable non-conformance, the choice of constraints has a large impact on the risk of fault-masking. At any given execution step, steering, as we have defined here, considers only those oracle post-states as candidate targets that are reachable from the given oracle pre-state. However, this by itself is not sufficiently restrictive to rule out truly deviant behaviors. Therefore, the constraints applied to reduce that search space must be strong enough to prevent steering from forcing the oracle into an otherwise impermissible state for that execution step. It is, therefore, crucial that proper consideration goes into the choice of constraints. In some cases, the use of additional policies—such as not steering the oracle model at all if it does not result in an exact match with the system—can also lower the risk of tolerating behaviors that would otherwise indicate faults.

Note that a seeded fault could *cause* a timing deviation (or the same behavior that would result from a timing deviation). In those cases, the failure is still labeled as being induced by a fault for our experiment. However, if the fault-induced deviation falls within the tolerances, steering will be able to correct it. In such cases, it is unlikely that even a human oracle would label the outcome differently, as they are working from the same tolerance limits.

If care is taken when deriving the tolerance constraints from the system requirements, steering should not cover any behaviors that would not be permissible under those same requirements. Still, as steering carries the risk of masking faults, we recommend that it be applied as a *focusing tool*—to point the developer toward test failures likely to indicate faults so that they do not spend as much time investigating non-conformance reports that turn out to be allowable. The final verdict on a test should come from a run of the oracle model with no steering, but during development, steering can be effective at streamlining the testing process by concentrating resources on those failures that are more likely to point to faults.

6.3 Steering vs Filtering

In some cases, allowable non-conformance events could simply be dealt with by applying a filter that, in the case of a failing test verdict, checks the resulting state of the SUT against a set of constraints and overrides the initial oracle verdict if those constraints are met. The use of a filter is tempting—if the filter is effective, it is likely to be easier to build and faster to execute than a full steering process. Indeed, the results in Table 6 appear promising. The filter performs identically to steering for the initial failures that result from non-deterministic timing differences. It does not issue a pass verdict for timing issues outside of the tolerance limits, and it does issue a pass for almost all of the tests where non-conformance is within the tolerance bounds.

However, when the results for tests that fail due to faults are considered, a filter appears much less attractive. The filter issues a passing verdict for 1,253 tests that should have failed—1,210 more

than steering. This is because a filter is a *blunt instrument*. It simply checks whether the state of the SUT meets certain constraints when non-conformance occurs. This allowed the filter to account for the allowed non-conforming behaviors, but these same constraints also allowed a large selection of fault-indicating tests to pass.

This makes the choice of constraints even more important for filtering than it is in steering. The steering process, by backtracking the state of the system, is able to ensure that the resulting behavior of the SUT is even possible (that is, if the new state is reachable from the previous state). The filter does not check the possibility of reaching a state; it just checks whether the new state is globally acceptable under the given constraints. As a result, steering is far more accurate. A filter could, of course, incorporate a reachability analysis. However, as the complexity of the filter increases, the reasons for filtering instead of steering disappear.

In fact, the success of steering at accounting for allowable non-conformance is somewhat misleading for this case example. Both filtering and steering base their decisions on the output variables of the SUT and oracle, on the basis that the internal state variables may differ between the two. For this case study, all of the output variables reflect *current conditions* of the infusion pump—how much drug volume to infuse now, the *current* system mode, and so forth. Internally, these factors depend on both the current inputs and a number of *cumulative factors*, such as the total volume infused and the remaining drug volume. Over the long term, non-conformance events between the SUT and model will build, eventually leading to wider divergence. For example, the SUT or the model-based oracle may eventually cut off infusion if the drug reservoir empties.

As the output variables reflect current conditions, mounting internal differences may be missed, and the filter may not be able to cope with large-scale behavior differences that result from this steady divergence. Steering is able to prevent and account for these long-term divergences by *actually changing the state of the oracle* throughout the execution of the test. A filter simply overrides the oracle verdict. It does not change the state of the oracle, and as a result, a filter cannot predict or handle behavioral divergences once they build beyond the set of constraints that the filter applies.

We can illustrate this effect by adding a single internal variable to the set of variables considered when making filtering or steering conditions—a variable tracking the total drug volume infused. Adding this variable causes *no change* to the results of steering seen in Tables 4 and 5. However, the addition of this internal variable dramatically changes the results of filtering. The new results can be seen in Tables 8 and 9.

Table 8: Distribution of results for step-wise filtering, (outputs + volume infused oracle). Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11311 (74.4%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	312 (2.1%)	1123 (7.4%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	598 (3.9%)	1688 (11.1%)

Because the total volume infused increases over the execution of the test, it will reflect any divergence between the model-based oracle and the SUT. As steering actually adjusts the execution of the model-based oracle, this volume counter also adjusts to reflect the changes induced by steering. Thus, steering is able to account for the growing difference in the volume infused by the model-based

Table 9: Precision, recall, and F-measure values for filtering (outputs + volume infused oracle).

Technique	Precision	Recall	F-measure
Filtering	0.64	0.76	0.70

oracle and the volume infused by the SUT. However, as the filter makes no such adjustment, it is unable to handle the mounting difference in this variable (or any other considered variable that reflects change over time). The filter, even if initially effective, will fail to account for a large number of acceptable non-conformance events—ultimately resulting in a precision value barely more effective than not doing anything at all.

6.4 Summary of Results

The precision, recall, and F-measure (a measure of accuracy) for each method—accepting the initial verdict, steering (using two different dissimilarity metrics), and filtering—are shown in Table 7. The default situation, accepting the initial verdict, results in the lowest precision value. Intuitively, not doing anything to account for allowed non-conformance will result in a large number of incorrect “fail” verdicts. However, the default practice does have the largest recall value. Again, not adjusting your results will prevent incorrect masking of faults. Filtering on a step-by-step basis results in higher precision, but due to the lack of reachability analysis and state adaptation—both of which used by the steering approach—the filter masks an unacceptably large number of faults.

Steering performs similarly for both of the dissimilarity metrics used in this study. It is able to adapt the oracle to handle almost every situation where non-conforming behaviors are allowed by the system requirements, while masking only a few faults in a small number of tests. The Squared Euclidean metric results in a higher precision, while both metrics obtain an equal recall value.

Ultimately, we find that steering—employing the Squared Euclidean dissimilarity metric—results in the highest accuracy for the final test results. Steering is able to automatically adjust the execution of the oracle to handle non-deterministic, but acceptable, behavioral divergence without covering up most fault-indicative behaviors. We, therefore, recommend the use of steering as a tool for focusing and streamlining the testing process.

7. THREATS TO VALIDITY

External Validity: Our study is limited to one case example. We are actively working with domain experts to produce additional systems for future studies. We believe this systems to be representative of the real-time embedded systems that we are interested in, and that our results will generalize to other systems in this domain.

We have used Simulink and Lustre as our modeling and implementation languages rather than more common languages such as C or C++. However, systems written in Lustre are similar in style to traditional imperative code produced by code generators used in embedded systems development. A simple syntactic transformation is sufficient to translate Lustre code to C code.

We have limited our study to fifty mutants for each version of the case example, resulting in a total of 150 mutants. These values are chosen to yield a reasonable cost for the study, particularly given the length of each test. It is possible the number of mutants is too low. Nevertheless, we have found results using less than 250 mutants to be representative [27, 28], and pilot studies have shown that the results plateau when using more than 100 mutants.

Internal Validity: Rather than develop full-featured system implementations for our study, we instead created alternative versions

of the model—introducing various non-deterministic behaviors—and used these models and the versions with seeded faults as our “systems under test.” We believe that these models are representative approximations of the behavioral differences we would see in systems running on embedded hardware. In future work, we plan to generate code from these models and execute the software on actual hardware platforms.

In our experiments, we used a default testing scenario (accepting the oracle verdict) and stepwise filtering as baseline methods for comparison. There may be other techniques—particularly, other filters—that we could compare against. Still, we believe that the filter chosen was an acceptable comparison point, and was designed as such a filter would be in practice.

Construct Validity: We measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software. Given that our approach to selecting oracle data is also based on the mutation testing, it is possible that using real faults would lead to different results. This is especially likely if the fault model used in mutation testing is significantly different than the faults we encounter in practice. Nevertheless, as mentioned earlier, Andrews et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults in similar fault finding experiments [2].

8. CONCLUSION AND FUTURE WORK

Specifying test oracles is still a major challenge for many domains, particularly those—such as real-time embedded systems—where issues related to timing, sensor inaccuracy, or the limited computation power of the embedded platform may result in non-deterministic behaviors for multiple applications of the same input. Behavioral models of systems, often built for analysis and simulation, are appealing for reuse as oracles. However, these models typically provide an *idealized* view of the system, and may struggle to differentiate unexpected—but still acceptable—behavior from behaviors indicative of a fault.

To address this challenge, we have proposed an automated *model-based oracle steering framework* that, upon detecting a behavioral difference, backtracks and transitions the model-based oracle, through a search process, to a state that satisfies certain constraints and minimizes a dissimilarity metric. This framework allows non-deterministic, but bounded, behavior differences while preventing future mismatches by guiding the model-based oracle—within limits—to match the execution of the SUT. Experiments, conducted over an infusion pump system, have yielded promising results and indicate that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing development costs. The use of our steering framework can allow developers to focus on behavioral difference indicative of real faults, rather than spending time examining test failure verdicts that can be blamed on a rigid oracle model.

There is still much room for future work. We plan to expand on the selection of case examples in terms of both program size and scope of non-determinism, as well as examining:

- The impact of different dissimilarity metrics, tolerance constraints, and steering policies on oracle verdict accuracy;
- Improvements to the speed and scalability of the steering framework, including the use of alternative search algorithms;
- The use of steering and dissimilarity metrics as methods of quantifying non-conformance and their utility in fault identification and location.

9. REFERENCES

- [1] IBM Rational Rhapsody. <http://www.ibm.com/developerworks/rational/products/rhapsody/>, 2014.
- [2] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pages 402–411, 2005.
- [3] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608 –624, aug. 2006.
- [4] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 Int'l Conf. on Testing software and systems*, pages 95–110. Springer-Verlag, 2010.
- [5] L. B. Brinksmo. A test generation framework for quiescent real-time systems. In *IN FATES'04*, pages 64–78. Springer-Verlag GmbH, 2004.
- [6] S.-H. Cha. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, 2007.
- [7] D. Coppit and J. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop, SEW '05*, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society.
- [8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [9] A. En-Nouary, R. Dssouli, and F. Khendek. Timed wp-method: testing real-time systems. *Software Engineering, IEEE Transactions on*, 28(11):1023–1038, Nov.
- [10] G. Gay, S. Rayadurgam, and M. P. Heimdahl. Steering model-based oracles to admit real program behaviors. In *Proceedings of the 36th International Conference on Software Engineering – NIER Track, ICSE '14*, New York, NY, USA, 2014. ACM.
- [11] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and E. Watanabe. Constructive model-based analysis for safety assessment. *Int'l Journal on Software Tools for Technology Transfer*, 14(6):673–702, 2012.
- [12] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29, 1994.
- [13] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [14] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [15] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
- [16] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, University of Sheffield, Department of Computer Science, 2013.
- [17] W. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293–298, 1978.
- [18] M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.
- [19] S. Kannan, M. Kim, I. Lee, O. Sokolsky, and M. Viswanathan. Run-time monitoring and steering based on formal specifications. In *Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, 2000.
- [20] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Int'l workshop on formal approaches to testing of software (FATES 04)*. Springer, 2004.
- [21] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [22] MathWorks Inc. Stateflow. <http://www.mathworks.com/stateflow>.
- [23] D. Miller, J. Guo, E. Kraemer, and Y. Xiong. On-the-fly calculation and verification of consistent steering transactions. In *Supercomputing, ACM/IEEE 2001 Conf.*, pages 8–8, 2001.
- [24] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.
- [25] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Modes, features, and state-based modeling for clarity and flexibility. In *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.
- [26] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.
- [27] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conf. on Software engineering*, pages 161–170. ACM, 2008.
- [28] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl. Requirements coverage as an adequacy measure for conformance testing, 2008.
- [29] T. Savor and R. Seviora. An approach to automatic detection of software failures in real-time systems. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 136–146, 1997.
- [30] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pages 870–880. IEEE Press, 2012.
- [31] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.
- [32] E. Weyuker. The oracle assumption of program testing.

Steering Model-Based Oracles to Admit Real Program Behaviors*

Gregory Gay, Sanjai Rayadurgam, Mats P.E. Heimdahl

Department of Computer Science & Engineering

University of Minnesota, USA

greg@greggay.com, [rsanjai,heimdahl]@cs.umn.edu

ABSTRACT

The *oracle*—an arbiter of correctness of the system under test (SUT)—is a major component of the testing process. Specifying oracles is challenging for real-time embedded systems, where small changes in time or sensor inputs may cause large differences in behavior. Behavioral models of such systems, often built for analysis and simulation, are appealing for reuse as oracles. However, these models typically provide an *idealized* view of the system. Even when given the same inputs, the model’s behavior can frequently be at variance with an acceptable behavior of the SUT executing on a real platform. We therefore propose *steering* the model when used as an oracle, to admit an expanded set of behaviors when judging the SUT’s adherence to its requirements. On detecting a behavioral difference, the model is backtracked and then searched for a new state that satisfies certain constraints and minimizes a dissimilarity metric. The goal is to allow non-deterministic, but bounded, behavior differences while preventing future mismatches, by guiding the oracle—within limits—to match the execution of the SUT. Early results show that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing development costs.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Verification

Keywords

Software Testing, Test Oracles, Model-Based Testing

1. INTRODUCTION

The *oracle* is a judge that determines the correctness of the execution of a given system under test (SUT) against a test suite.

*This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715 and an NSF Graduate Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE ’14, May 31—June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05 ...\$15.00.

Despite increased attention in recent years, the *test oracle problem* [5]—constructing efficient and robust oracles—remains a major challenge for many domains. Real-time process control systems—embedded systems that interact with physical processes such as pacemakers or power management systems—are particularly difficult to build oracles for, as their behavior depends not only on the values of inputs and outputs, but also on their time of occurrence [2], and minor behavioral distinctions may have significant consequences [6]. When executing on an embedded hardware platform, several sources of non-determinism, such as input processing delays, execution time fluctuation, and hardware inaccuracy, can make the SUT exhibit varying but acceptable behaviors.

Behavioral models [8], typically expressed as state-transition systems, represent requirements by prescribing the behavior (the system state) to be exhibited in response to each input. Such models are used for many purposes in industrial software development and so their reuse as test oracle is highly desirable. However, these models provide an abstract view of the system that typically simplifies the actual conditions in the execution environment. On a real hardware platform, the SUT may exhibit behavior that differs from what the model prescribes for a given input. Over time, these differences can build to the point where the execution paths of the model and the SUT diverge enough to flag “failure”, even if the system is operating within the system requirements.

We take inspiration for addressing this problem from *program steering*, the process of adjusting the execution of live programs in order to improve performance, stability, or correctness [9]. We hypothesize that behavioral models can be adapted for use as oracles for real-time systems through the use of *steering actions* [13]. By comparing the state of the model-based oracle (MBO) with that of the SUT following an output event, we can guide the model to match the state of the SUT, as long as a set of constraints are met. The result would be a widening of the behaviors accepted by the MBO, thus compensating for allowable non-determinism, without impairing the ability of the MBO to judge the SUT.

We propose an automated framework for comparing and steering the MBO with respect to the SUT, discuss the current prototype implementation of the framework, and assess its capabilities on a model for the control module of an infusion pump—a real-world system with complex, time-based behaviors. Results indicate that steering successfully accounts for all allowable differences between the MBO and the SUT, eliminating a large number of spurious “failure” verdicts. However, steering also masks a small number of faults, indicating a need to future work.

To the best of our knowledge, this is the first work proposing the automated steering of the test oracle. While this research is still in its infancy, the initial results are promising. If successful, this approach would lower testing costs and reduce development

```

1. for test in Tests
2.   for step in test
3.     initialVerdict = Sym(Sm, Ssut)
4.     if initialVerdict > 0
5.       oldState = Sm
6.       targetState=callModelChecker(model,Sm,Ssut,Sym())
7.       while Sym(targetState, Ssut) < Sym(oldState, Ssut)
8.         oldState = targetState
9.         targetState=callModelChecker(model,Sm,Ssut,Sym())
10.        transitionModel(model,targetState)

```

Figure 1: Steps in the Steering Process

effort for real-time control systems by enabling reuse of behavioral models as oracles.

2. PROBLEM DEFINITION

A *test oracle* is defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [5]. Although oracles can be derived from many sources of information, we are particularly interested in using behavioral models, such as those often built for purposes of simulation, analysis and testing [8].

Executable behavioral models can be divided into *declarative* models created in formal specification languages, and *constructive* models built with state-transition languages such as Simulink & Stateflow, Statemate, finite state machines, or other automata structures [8]. Presently, we focus on constructive state-transition systems, as these are frequently used to model real-time control systems. Real-time control systems monitor and interact with a physical environment. Examples of such systems include pacemakers, traffic-control systems, and power plant management software.

Non-determinism is a major concern for systems that interact with physical processes. The task of monitoring the environment and pushing signals through multiple layers of sensors and actuators can introduce additional points of failure, delay, and unpredictability. Input and observed output values may be skewed by hardware noise, timing constraints may not be met with precision, or inputs may arrive faster than the system can process them. Often, the system behavior may be acceptable, even if such behavior is not what was captured in the model—which, by its very nature, incorporates a simplified view of the problem domain. A common abstraction when modeling is to elide any details that distract from the core system behavior in order to ensure that model-based analyses are feasible and useful. Yet these details manifest as differences between the model and the implemented system.

This raises the question—why use models as oracles? Alternative approaches could be to turn to an oracle based on explicit behavioral constraints—assertions or invariants—or to build declarative behavioral models in a formal notation such as Modelica. However, these solutions have their limitations. Assertion-based approaches only ensure that a limited set of properties hold at *particular points in the program*. Further, such oracles may not be able to account for the same range of testing scenarios as a model that prescribes behavior for all inputs. Declarative models that express the computation as a theory in a formal logic allow for more sophisticated forms of verification and can better account for time-constrained behaviors [3]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [10]. Constructive models are visually appealing, easy to analyze without specialized knowledge, and suitable for analyzing failure conditions and events in an isolated manner [3]. The complexity of declarative models and the knowl-

edge needed to design and interpret such models make widespread industrial adoption of the paradigm unlikely.

While there are challenges in using constructive model-based oracles, it is a widely held view that such models are indispensable in other areas of development and testing, such as automatic test generation [7]. From this standpoint, the motivational case for models as oracles is clear—if these models are already being built, their reuse as test oracles could save a large amount of time and money. Practitioners are well-versed in these models and so these are likely to be less error-prone compared to building a new unfamiliar type of oracle. Therefore, we seek a way to use constructive model-based oracles that can handle the non-determinism introduced during system execution on the target hardware.

3. ORACLE STEERING

In a typical model-based testing framework, the test suite is executed against both the SUT and the MBO, and the values of certain variables are recorded to a trace file after each execution step. The oracle’s comparison procedure examines those traces and issues a verdict for each test (*fail* if test reveals discrepancies, *pass* otherwise). When testing a real-time system, we would expect non-determinism to lead to behavioral differences between the SUT and the MBO during test execution. The actual behaviors witnessed in the SUT may not be incorrect—they may still meet the system requirements—but they just do not match what the model produced. We would like the oracle to distinguish between correct, but variant behaviors introduced by non-determinism and behaviors that are incorrect.

An approach to address this would be to augment the comparison procedure with a filtering mechanism to detect and discard acceptable differences on a per-step basis. The issue with this is that the effect of non-determinism may linger on for several steps, leading to irreconcilable differences between the SUT and MBO. Filters may not be effective at handling growing behavioral divergence.

We take inspiration from *program steering*—however, instead of steering the SUT, we *steer the oracle* to see if the model is capable of matching the SUT’s behavior. When the two behaviors differ, we backtrack and apply a *steering action*—e.g., adjust timer values, apply different inputs, delay or withhold an input—that changes the state of the MBO to a state more similar to the SUT (as judged by a dissimilarity metric). Oracle steering, unlike filters, is *adaptable*. Such actions provide flexibility to handle non-determinism, while still retaining the power of the oracle as an arbiter. Improper steering can bias the behavior of the MBO, masking both acceptable deviations and actual indications of failures. However, we believe that it is possible to sufficiently bound steering such that the ability to detect faults is still retained, using a series of constraints:

1: A set of tolerance constraints governing the allowable changes to certain variables (input, internal, or output) in the model that can be effected by steering. These rules define the level of non-determinism or behavioral deviation that can be accounted for with steering. “No change allowed to the system’s operational mode” or “computed output may be produced between x and y seconds” are examples of such tolerance constraints.

2: A dissimilarity function $Sym(model\ state, SUT\ state)$, that compares the state of the model to the observable state of the SUT. We seek a minimization of $Sym(s_m^{new}, s_{sut}) < Sym(s_m, s_{sut})$.

3: A further set of general policy decisions on when to steer. For example, one might decide not to steer unless $Sym(s_m^{new}, s_{sut}) = 0$ — that is, unless there exists a steering action that results in a model state identical to that observed in the SUT.

In other words, the new state of the MBO following the application of a steering action must be one that is possible to reach within

a limited number of transitions from the current state of the model, must fall within the boundaries set by the tolerance constraints, and must minimize the dissimilarity function.

We have implemented the basic steering approach outlined in Figure 1. It is a search process based on SMT-based bounded model checking [4], which is a natural choice for this problem. We have a series of constraints that govern steering actions and seek to locate a model state reachable in a limited number of transitions that satisfies those constraints and minimizes a dissimilarity metric. Specifically, we make use of the Kind model checker [4].

Note that the constraint $Sym(s_m^{new}) < Sym(s_m)$ would give us a model state that is more similar to the behavior of the SUT than the original transition taken by the MBO, but carries no guarantee that the satisfying state minimizes the dissimilarity metric. Thus, we apply the model checker with this constraint in order to get an initial threshold, then iteratively reapply the model checker with new thresholds until we can no longer find a better solution. The best solution found then becomes the new model to that state.

4. RELATED WORK

Several authors have addressed model-based conformance testing for real-time systems [1, 7, 2]. For example, Larsen et al. [7] model a system as a non-deterministic timed automata that is constrained by an environment model. The combined model serves as an oracle during test execution.

While several approaches consider limited non-determinism, there are a few key differences with our proposed approach, which decouples the model from the rules governing steering. This makes non-determinism implicit and the approach more generally applicable. Explicitly specified non-deterministic behavior would limit the scope of non-determinism handled by the oracle to what has been modeled. It is difficult to anticipate the non-determinism resulting from deploying software on a hardware platform, and thus, such models will likely undergo several revisions during development. Steering instead relies on a set of rule-based constraints that may be easier to revise over time. Also, by not relying on a specific model format, steering can be made to work with models created for a variety of purposes.

Oracle steering is conceptually similar to *dynamic* program steering, the automatic guidance of program execution [9]. The most relevant in terms of the techniques employed is Spec Explorer [13], where an executable behavioral model is steering through different execution scenarios for test generation as opposed to adjudging system execution.

5. PILOT STUDY

We aim to gain an understanding of the capabilities of oracle steering and the impact it has on the testing process—both positive and negative. Thus, we pose the following research questions:

1. To what degree does steering lessen behavior differences that are legal under system requirements?
2. To what degree does steering mask behavior differences that fail to conform to the requirements?

Models and Test Generation: We have based our MBO on the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [11]. This model, developed in the Simulink and Stateflow notations and translated into the Lustre synchronous programming language, is a complex real-time system of the type common in the medical domain. This subsystem controls the operational mode of the infusion pump, including the flow rate of the medicine administered to the patient.

Table 1: Verdicts: T(true)/F(false), P(positive)/N(negative).

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	TN	FP
Fail (Due to Timing, Within Tolerance)	TN	FP
Fail (Due to Timing, Not in Tolerance)	FN	TP
Fail (Due to Fault)	FN	TP

In order to produce “implementations” of this system, we created two clones of the model, each introducing a realistic non-deterministic timing change to the system: (1) a version of the system where the exit of the patient-requested dosage period may be delayed by a short length of time, and (2), a version of the system where the exit of an intermittent dosage period may be delayed. We have also created 50 *mutants* (faulty implementations) of the original system and the two implementation models by introducing a single fault into each model¹.

Using a random testing algorithm, we generated 100 tests—each 30 steps long—and ran them against each model in order to collect traces. In the models with timing fluctuations, we controlled those fluctuations through the use of an additional input variable. The value for that variable was generated non-deterministically, but we used the same value across all systems with the same -timing fluctuation. As a result, we know whether a behavior is due to a timing fluctuation or a seeded fault in the system.

Steering Constraints: For this system, we specified tolerance constraints in terms of the input variables of the system (although constraints can be placed on internal or output variables as well):

- Five of the input variables relate to timers within the system. For each of those, we placed an allowance of $(Current\ Value - 1) \leq New\ Value \leq (Current\ Value + 2)$. For example, a dosage duration is allowed to fall within a four second period—between one second shorter and two seconds longer than the prescribed duration.
- The remaining 15 input variables are not allowed to be steered.

These constraints reflect what we consider a realistic application of steering—we allow a small window around the behaviors that we accept that are related to timing, but as we do not expect any sensor noise, we do not allow freedom in adjusting sensor-based inputs.

The dissimilarity metric used in this experiment is the L^1 – Norm. Given vectors representing the state of the SUT and the MBO—where each member of the vector represents the value of a variable—the dissimilarity between the two vectors can be measured as the absolute numerical distance between the state of the SUT and the MBO. A constant difference of 1 is used for differences between boolean variables.

Experiment Overview: Using the generated artifacts—without steering—we monitored the outputs during each test, compared the results to the values of the same variables in the MBO to calculate the similarity score, and issued an initial verdict. Then, if the verdict was a failure ($score > 0$), we steered the MBO, and recorded a new verdict post-steering.

We can assess the impact of steering using the verdicts made before and after steering by calculating the number of *true positives*—steps where steering does not mask incorrect behavior—the number of *false positives*—the number of steps where steering fails to account for an acceptable behavioral difference—and the number of *false negatives*—the number of steps where steering does masks an incorrect behavior.

¹The mutation operators used are discussed at length in [12].

Table 2: Distribution of results.

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	17835 (70.2%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	2443 (9.6%)	0 (0.0%)
Fail (Due to Timing, Not in Tolerance)	266 (1.0%)	536 (2.1%)
Fail (Due to Fault)	399 (1.5%)	3921 (15.4%)

Table 3: Precision, recall, and F-measure values.

Technique	Precision	Recall	F-measure
No Steering	0.66	1.00	0.80
With Steering	1.00	0.87	0.93

The testing outcomes in terms of true/false positives/negatives are listed in Table 1. Using these measures, we calculate the *precision*—the ratio of true positives to all positive verdicts—and *recall*—the ratio of true positives to true positives and false negatives. We also calculate the *F-measure*—the harmonic mean of precision and recall—in order to judge the accuracy of oracle verdicts.

Results: Table 2 shows the results of steering—a pass or fail—following each of the pre-steering testing outcomes—a pass, a fail due to allowable timing fluctuations, a fail due to unacceptable timing fluctuations, or a fail due to a fault in the system. The precision, recall, and F-measure values for steering and the default testing scenario (issuing a verdict without steering) are listed in Table 3.

For the case example studied, steering is able to account for all situations where non-deterministic timing affects conformance while both the MBO and the implementation remain within the bounds set in the system specification. Therefore, we see a sharp increase in precision over the default situation where steering is not applied. Previously, a developer would have to manually inspect nearly 30% of the tests for faults in the system (all pre-steering failures in Table 2). Now, they would be asked to focus on 17.5% of the test results (all post-steering failures).

Note, however, that 665 tests that should have still been labeled as failures post-steering are now labeled as passing. This is a relatively small number—only 2.5% of the test results—but any loss in recall is cause for concern when working with safety-critical systems. Still, as can be seen from the F-measure, the application of steering results in greater *accuracy* in test classifications than not steering. These results seem promising, and further examination of constraints and dissimilarity metrics should improve recall.

As steering does carry the risk of masking faults, we recommend that it be applied as a *focusing tool*—to point the developer toward definite faults so that they do not spend as much time investigating potential faults. The increase in oracle verdict accuracy and the large decrease in failure verdicts—from 7565 to 4457 tests—after steering should result in a substantial decrease in the amount of time that developers spend investigating failure verdicts are actually instances of allowable non-conformance.

6. CONCLUSION AND FUTURE WORK

The results of the initial pilot study are quite promising. If steering can be scaled to larger systems without loss in accuracy, then the potential for improvements in the time and effort spent on testing real-time embedded systems are significant. The use of steering can allow developers to focus on addressing definite faults, rather than spending time examining test failure verdicts that can be blamed on a rigid oracle model.

However, there is much room for future work. We plan to expand the case examples in terms of both program size and scope of

non-determinism. The dissimilarity metric used in this study was deliberately simplistic. In future work, we will study more sophisticated metrics, including ones that admit equivalent behaviors. We plan to examine the impact of different sets of tolerance constraints and overall steering policy decisions on the accuracy of steering. We would also like to examine the cost and time savings that could result from the application of steering.

7. REFERENCES

- [1] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, pages 95–110. Springer-Verlag, 2010.
- [2] A. En-Nouary, R. Dssouli, and F. Khendek. Timed wp-method: testing real-time systems. *Software Engineering, IEEE Transactions on*, 28(11):1023–1038, Nov.
- [3] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and E. Watanabe. Constructive model-based analysis for safety assessment. *International Journal on Software Tools for Technology Transfer*, 14(6):673–702, 2012.
- [4] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [5] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, University of Sheffield, Department of Computer Science, 2013.
- [6] M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.
- [7] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *International workshop on formal approaches to testing of software (FATES 04)*. Springer, 2004.
- [8] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [9] D. Miller, J. Guo, E. Kraemer, and Y. Xiong. On-the-fly calculation and verification of consistent steering transactions. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 8–8, 2001.
- [10] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.
- [11] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Modes, features, and state-based modeling for clarity and flexibility. In *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.
- [12] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proc. of the 10th Int'l Conf. on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.
- [13] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

Moving the Goalposts: Coverage Satisfaction Is Not Enough^{*}

Gregory Gay*, Matt Staats[†], Michael W. Whalen*, and Mats P.E. Heimdahl*

*Department of Computer Science & Engineering
University of Minnesota, USA
greg@greggay.com,
[whalen,heimdahl]@cs.umn.edu

[†]Interdisciplinary Centre for
Security, Reliability and Trust
University of Luxembourg
matthew.staats@uni.lu

ABSTRACT

Structural coverage criteria have been proposed to measure the adequacy of testing efforts. Indeed, in some domains—e.g., critical systems areas—structural coverage criteria must be satisfied to achieve certification. The advent of powerful search-based test generation tools has given us the ability to generate test inputs to satisfy these structural coverage criteria. While tempting, recent empirical evidence indicates these tools should be used with caution, as merely achieving high structural coverage is not necessarily indicative of high fault detection ability. In this report, we review some of these findings, and offer recommendations on how the strengths of search-based test generation methods can alleviate these issues.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Verification

Keywords

Software Testing, Automated Test Generation, Structural Coverage

1. INTRODUCTION

Central to software test selection are *structural coverage criteria*, which measure test suite adequacy in terms of coverage over the structural elements of the system under test, such as statements or control flow branches. These criteria are so trusted that they are required for certification when testing avionics systems.

These adequacy metrics can be easily adapted as objective functions for search-based optimization algorithms, and as a result, there has been rapid progress in the creation of automated test generation tools that direct the generation process to satisfy structural coverage criteria [1]. Such tools promise to improve coverage and reduce the cost associated with test creation.

*This work has been supported by NASA Ames Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583, CNS-0931931, and CNS-1035715, and the Fonds National de la Recherche, Luxembourg (FNR/P10/03).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SBST '14, June 2 – June 3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2852-4/14/06 ...\$15.00.

In principle, this represents a success for software engineering—an arduous engineering task has been automated. However, in actuality, the effectiveness of test suites automatically generated to satisfy various structural coverage criteria has not been firmly established in practice. Of crucial importance is the *method* of test generation: while evidence establishing that coverage is positively correlated with fault detection, such evidence typically holds the method of test generation fixed, e.g., demonstrating that using structural coverage to guide random test generation provides better tests than purely random tests [13]. Consequently, much of the work in test generation (search-based and otherwise) fails to address the effectiveness of the generated tests, and the studies that have examined this issue have returned mixed results [7, 5, 3].

In recent work, we have examined the effectiveness of automated test generation in depth and found that test inputs generated specifically to satisfy structural coverage criteria via counterexample-based test generation were typically *less effective* than randomly generated test inputs [13]. These results are disconcerting—given the central role of coverage criteria in testing research and practice, and the rise of automated tools that can provide such coverage, the temptation exists to automate the entire testing process. Our results indicate that it is not sufficient to simply maximize the level of code coverage when generating tests. Recent research suggests that a number of factors that are currently not well understood can strongly impact the effectiveness of the testing process, such as the oracle used or the structure of the program under test [16].

These findings lead us to the conclusion that code coverage is merely one factor in the broader objective of generating effective test input. Search-based test generation algorithms must evolve to take into account additional factors such as fault propagation, system structure, and the execution points monitored by the test oracle. In this report, we discuss our findings and make recommendations on the next generation of search-based test-generation approaches.

2. THE CENTRAL QUESTION

Advances in search-based test generation should be celebrated; however, we seek evidence that using such generation techniques to maximize structural coverage yields test suites that are more effective at fault detection than naïve generation methods such as random testing. That is, are tests automatically generated to achieve coverage actually useful for finding faults in the system under test?

Empirical studies comparing structural coverage criteria have mixed results. Juristo et al. provide a survey of much of the existing work [8]. With respect to branch coverage, they note that some authors (such as Hutchins et al. [7]) find that branch coverage outperforms random testing, while others (such as Frankl and Weiss [5]) discover the opposite. Namin and Andrews have found coverage levels are positively correlated with fault finding effectiveness [11].

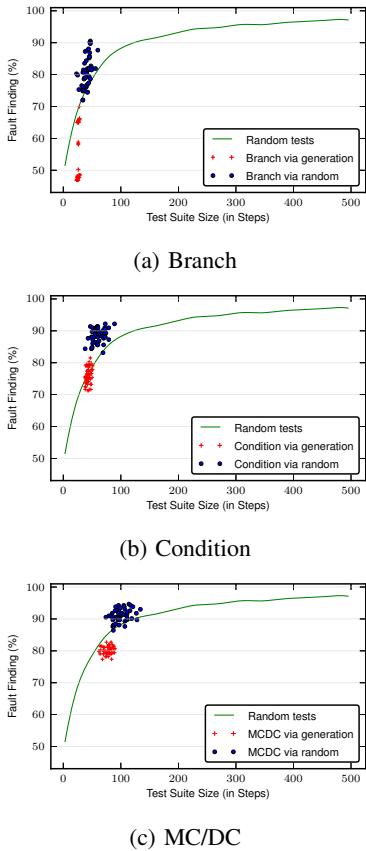


Figure 1: Percentage of faults identified compared to test suite size for generated test suites, coverage-satisfying subsets of randomly generated tests, and pure random test generation.

Theoretical work comparing the effectiveness of partition testing against random testing yields similarly mixed results. Chen and Yu indicate that partition testing is not necessarily more effective than random testing [3]. Later theoretical work by Gutjahr [6], however, provides a stronger case for partition testing. Kandl and Kirner evaluate MC/DC using an example from the automotive domain and note less than ideal fault finding [9]. Dupuy and Leveson evaluate the MC/DC as a complement to functional testing, finding that the use of MC/DC improves the quality of tests [4]. None of these studies, however, compare the effectiveness of test generation to satisfy MC/DC to other forms of test generation. They, therefore, do not indicate if test suites satisfying MC/DC are truly effective, or if they are effective merely because MC/DC test suites are generally quite large. Arcuri et al. [2] recently demonstrated that in many scenarios, random testing is more predictable and cost-effective at reaching high levels of structural coverage than previously thought.

Most studies concerning automated test generation for structural coverage maximization are focused on how to generate tests quickly or on improving coverage [1]. Comparisons of the fault finding effectiveness of the resulting test suites against other methods of test generation are few. Those that exist apart are largely, to the best of our knowledge, studies in concolic execution [12]—a combination of random testing with symbolic execution.

Recently, we have conducted a large-scale case study to evaluate the effectiveness of test suites generated to satisfy branch and MC/DC coverage using counterexample-based test generation and a random generation approach, contrasted against purely random test suites of equal size [13]. Our results yielded two key conclu-

sions. First, coverage criteria satisfaction alone is a poor indication of fault finding effectiveness, with random test suites of equal size providing similar—and often higher—levels of fault finding. Second, the use of structural coverage as a supplement—rather than a target—for test generation can have a positive impact, with random test suites reduced to a coverage-providing subset detecting up to 135% more faults than test suites generated to achieve coverage. A typical result from this study is illustrated in Figure 1, where for the three coverage criteria, random tests and a random test suite reduced to a coverage-satisfying subset outperform test suites automatically generated to achieve coverage of the same size.

3. KEY ISSUES

The existing body of research on this topic leads us to the observations that *it is not sufficient to simply maximize a structural coverage metric when automatically generating test inputs*, and therefore in practice *how coverage is achieved matters*.

The key underlying issues relate to how structural coverage criteria are formulated and how automatic test generation tools operate. Traditional coverage criteria are formulated over specific elements in the source code. To cover an element, (1) execution must reach the element and (2) exercise the element in a specific way. However, commonly used structural coverage criteria typically leave a great deal of leeway to how the element is reached, and—more importantly, in our experience—place no constraints whatsoever on how the test should evolve after the element is exercised. Automatic test generation tools typically use this freedom to do just enough work to satisfy coverage criteria, without consideration of, for example, how the faults are to be detected by the test oracle.

First, let us consider the path to satisfy a coverage obligation, e.g., a branch of a complex conditional. Structural coverage criteria require only that the point of interest is reached and exercised. We have found that automatically generated tests often take a shortest-path approach to satisfying test obligations, and manipulate only a handful of input values, leaving other inputs at default values. This is a cost effective method of satisfying coverage obligations; why tinker with program values that do not impact the coverage achieved? However, we and other authors have observed that variations provided by (for example) simple random testing result in test suites which are nearly as effective in terms of coverage, and moreover produce more interesting behavior capable of detecting faults [2, 13]. As illustrated in Figure 1, even with less coverage achieved, randomly generated test inputs can outperform automatically generated test suites in terms of fault finding.

Second, for the most commonly used structural coverage criteria, there is no directive concerning how tests should evolve after satisfying the structural element. Given this lack of direction, automatic test generation tools typically do not consider the path from the covered element to an output/assertion/observable state when generating a test, and therefore test inputs may achieve high coverage but fail to demonstrate faults that exist within the code. One reason for this failure is *masking*, which occurs when expressions/computations in the system are prevented from influencing the observed output, i.e., do not reach a variable or assertion monitored by the test oracle. More generally, this is related to the distinction between incorrect program state and program output: just because a test triggers a fault, there is no guarantee this will manifest as a *detected* fault. Indeed, in our experience, care must be taken to ensure that this occurs.

These two high level issues result in the generation of test inputs that may indeed be effective at encountering faults, but may make actually observing them—that is, actually detecting the fault—very difficult or unlikely. This reduces the effectiveness of any testing

process based on structural coverage criteria, as we can easily satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

We believe that these issues raise serious concerns about the efficacy of coverage-directed automated testing. As mentioned, the focus in automatic test generation work is currently on efficiently achieving coverage without carefully considering how achieving coverage impacts fault detection. We therefore run the risk of producing tools that are satisfying the letter of what is expected in testing, but not the spirit. Nevertheless, the central role of coverage criteria in testing is unlikely to fade, as demonstrated by the increasing use of coverage criteria for certification.

Hence, the key is to improve upon the base offered by these criteria and existing technology. We have come to the conclusion that the research goal in search-based test generation should not be developing methods of maximizing structural code coverage, but rather determining how to *maximize fault-finding effectiveness*. We propose that test generation is, in fact, a multi-objective optimization problem, and that algorithms built for test generation must evolve to take into account factors beyond the naive execution of individual elements of the code—factors such as masking, program structure, and the execution points monitored by the test oracle.

4. RECOMMENDATIONS

To support the broader objective of effective testing, we believe that search-based test generation approaches must evolve to take into account multiple factors—including not only satisfaction of structural coverage criteria, but also the propagation of faults to execution points monitored by the test oracle, and the underlying structure of the system under test. Given the two core issues identified in Section 3, we recommend two approaches.

First, we could improve, or replace, existing structural coverage criteria, extending them to account for factors that influence test quality. Automated test generation has improved greatly in the last decade, but the targets of such tools have not been updated to take advantage in this increase in power. Instead, we continue to rely on criteria that were originally formulated when manual test generation was the only practical method of ensuring 100% coverage.

Second, search-based test generation tools could be improved to avoid pitfalls when using structural coverage criteria. This could take many forms, but one straightforward approach would be to develop additional heuristics or rules that could operate alongside existing structural coverage criteria. For instance, tools could be encouraged to generate longer test cases, increasing the chances that a corrupted internal state would propagate to an observable output (or other monitored variable). Also, important factors specific to individual domains, e.g. web testing vs embedded systems, could be empirically identified and formalized as heuristics.

4.1 Use More Robust Coverage Criteria

In our own work, the issues of criteria formulation and masking motivated the development of the Observable MC/DC coverage criterion [16]. OMC/DC coverage explicitly avoids issues related to masking by requiring that its test obligations both demonstrate the independent impact of a condition on the outcome of the decision statement, and follow a non-masking propagation path to some variable monitored by the test oracle. OMC/DC, by accounting for the program structure and the selection of the test oracle, can, in our experience, address some of the failings of traditional structural coverage criteria within the avionics domain, allowing for the generation of test suites achieving up to 26% better fault detection than random test suites of equal size. Similarly, Vivanti et.al have demonstrated evidence that the use of data-flow coverage as a goal

for test generation results in test suites with higher fault detection capabilities than suites generated to achieve branch coverage [15].

The primary problem with many existing structural coverage criteria is that all effort is expended on covering structures internal to the system, and no further consideration is paid to how the effect of covered structure reaches an observable point in the system. These results indicate that it is possible to overcome some of the issues we have highlighted by working with stronger coverage criteria or extending existing criterion to take into account issues such as fault propagation. OMC/DC considers the observability aspect for Boolean expressions (using MC/DC) by appending a path condition onto each test obligation in MC/DC. Similar extensions could be applied to a variety of other existing coverage metrics, e.g., boundary value testing.

4.2 Algorithmically Improve Test Selection

Extensions to coverage criteria are not without downsides: for stronger metrics, programs will contain *unsatisfiable* obligations where there is no test that can be constructed to satisfy the obligation. Depending on the search strategy, the test generator may never terminate on such obligations. Further, the higher cost and difficulty of generating OMC/DC satisfying test suites—relative to generating the weaker branch, condition, or MC/DC test suites—makes the use of strong coverage criteria as targets for test generation harder to universally recommend.

Instead, another possible method of ensuring test quality is to use a traditional structural coverage metric as the objective of test generation, and augment this by considering other factors empirically established to impact fault detection effectiveness. In the context of search-based test generation, this means adding additional objective functions to the search strategy, rather than adding additional constraints. For instance, an algorithm could both work to maximize an existing structural coverage criterion and minimize the propagation distance between the assignment of a value and its observation by the oracle (an algorithm that minimizes this distance for test prioritization purposes already exists [14]).

As an example, consider purely random testing. Random testing does not employ an objective function, but it is possible to use one to approximate how well the system’s state space is being covered. We have seen systems containing bottlenecks (pinch-points) in the state space where randomly generated tests perform very poorly. Such bottlenecks require certain specific input sequences to reach a large portion of the state space. However, approaches such as concolic testing [12]—combining random testing with symbolic execution—are able to direct the generation of tests around such bottlenecks. Similar approaches could be employed to direct test generation towards, for example, propagation paths from variable assignment to oracle-monitored portions of the system.

While work exists considering test suite generation as a multi-objective problem, we believe that there is still much need for research in this area. By pursuing multiple objectives, we could potentially offer stronger tests that satisfy MC/DC obligations, even when it would be impossible to generate a test that satisfies the corresponding—but stricter—OMC/DC obligation. Embedding aspects of test selection into the objective function—or even into the search algorithm itself—may allow for improved efficiency. The search method can use, say, masking as a pruning mechanism on paths through the system, and the algorithm would not have to track as much symbolic information related to the objective metric itself.

4.3 Tailor an Approach to the Domain

It is important to emphasize that there is no “one size fits all” solution to test generation. The size and shape of the state space of

a system varies dramatically between domains and programming paradigms, and, as a result, it is difficult to tailor universal testing strategies. Much of our work has focused on embedded systems that run as cyclic processes. In this area, a common issue is that the impact of exercising a code path on the system’s output is delayed; only several cycles after a fault occurs can we observe it. If the goal of test generation is only to cause the code path to be executed, many of the tests will not cause a visible change in system behavior. In object-oriented systems, a central, but related issue is that a method call may change internal state that, again, is not visible externally until another method call produces output. As a result, choosing appropriate method sequences and their ordering becomes a major challenge.

Thus while general rules and heuristics for improving test generation are valuable, we believe there are large improvements to be found in tailoring the approach to the testing challenge at hand. For example, two often overlooked factors are the cost of generating tests and the cost of running tests. It is hard to outperform random testing in terms of the cost of generating tests, because doing so requires very little computation. If it is also cheap to run tests, then for many systems it is difficult to outperform straight random testing. On the other hand, if it is expensive to run tests, e.g., for embedded systems, this may require access to a shared hardware ‘rig’. In this case, using search-based techniques to generate tests for a specific strong coverage criterion (such as OMC/DC) may be very sensible, because the number of required tests can be dramatically smaller than the number of random tests required to achieve the same level of fault finding.

Another overlooked factor is the “reasonableness” of generated tests. Automated test generation methods should deliver tests that not only find faults, but are also meaningful to the domain of interest. Coverage-based techniques take the path of least resistance when generating tests, but can produce tests that make little sense in the context of the domain. Techniques that can generate inputs with meaning to the human testers are valuable in reducing the “human oracle cost” associated with checking failing test results [10].

Addressing factors such as these must be done on a per domain level, and indicate that code coverage should be one of several goals in test generation. We suspect that, in the long run, effective automatic test generation tools will consist of both general techniques and heuristics and additional *test generation profiles* for each domain. Determining the *correct* objective functions for test generation for each domain is still an open research question, one requiring both technical advancements in search-based test generation and empirical studies.

5. CONCLUSION

While coverage criteria are a central aspect of software testing, both in practice and in research, there is an increasing body of work indicating that high structural coverage alone does not lead to effective testing—how the tests are generated also matters. To achieve effective testing, we should consider other factors, both general (e.g. test case length, path to test oracles) and domain specific.

Search-based test generation approaches, with their use of potentially multiple objective functions to guide search, are particularly well-suited to moving beyond simple structural coverage. We therefore believe that careful selection of the algorithms employed, the tuning of the parameters, and the objective functions used all have important roles to play. Moving in this direction requires work both empirical, determining exactly what factors influence fault detection the most for each software domain, and technical, translating that empirically acquired knowledge into heuristics and objective functions.

6. REFERENCES

- [1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.
- [2] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ISSTA*, pages 219–230, 2010.
- [3] T. Chen and Y. Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering*, 22(2), 1996.
- [4] A. Dupuy and N. Leveson. An empirical evaluation of the MC/DC coverage criterion on the hete-2 satellite software. In *Proc. of the Digital Aviation Systems Conference (DASC)*, Philadelphia, USA, October 2000.
- [5] P. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proc. of the Symposium on Testing, Analysis, and Verification*, 1991.
- [6] W. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.
- [7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proc. of the 16th Int’l Conf. on Software Engineering*. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.
- [8] J. Juristo, A. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1):7–44, 2004.
- [9] S. Kandl and R. Kirner. Error detection rate of MC/DC for a case study from the automotive domain. *Software Technologies for Embedded and Ubiquitous Systems*, pages 131–142, 2011.
- [10] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, STOV ’10, pages 1–4, New York, NY, USA, 2010. ACM.
- [11] A. Namin and J. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. of the 18th Int’l Symp. on Software Testing and Analysis*. ACM, 2009.
- [12] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the 10th European Software Engineering Conf. / 13th ACM SIGSOFT Int’l. Symp. on Foundations of Software Engineering*. ACM New York, NY, USA, 2005.
- [13] M. Staats, G. Gay, M. W. Whalen, and M. P. Heimdahl. On the danger of coverage directed test case generation. In *15th International Conference on Fundamental Approaches to Software Engineering (FASE)*, April 2012.
- [14] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *ISSRE*, pages 311–320, 2012.
- [15] M. Vivanti, A. Mis, A. Gorla, and G. Fraser. Search-based data-flow test generation. In *ISSRE’13: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*. IEEE Press, Nov. 2013.
- [16] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 International Conference on Software Engineering*. ACM, May 2013.

The Risks of Coverage-Directed Test Case Generation

Gregory Gay, Member, IEEE, Matt Staats, Michael Whalen, Senior Member, IEEE, and Mats P.E. Heimdahl, Senior Member, IEEE

Abstract—A number of structural coverage criteria have been proposed to measure the adequacy of testing efforts. In the avionics and other critical systems domains, test suites satisfying structural coverage criteria are mandated by standards. With the advent of powerful automated test generation tools, it is tempting to simply generate test inputs to satisfy these structural coverage criteria. However, while techniques to produce coverage-providing tests are well established, the effectiveness of such approaches in terms of fault detection ability has not been adequately studied.

In this work, we evaluate the effectiveness of test suites generated to satisfy four coverage criteria through counterexample-based test generation and a random generation approach—where tests are randomly generated until coverage is achieved—contrasted against purely random test suites of equal size. Our results yield three key conclusions. First, coverage criteria satisfaction alone can be a poor indication of fault finding effectiveness, with inconsistent results between the seven case examples (and random test suites of equal size often providing similar—or even higher—levels of fault finding). Second, the use of structural coverage as a supplement—rather than a target—for test generation can have a positive impact, with random test suites reduced to a coverage-providing subset detecting up to 13.5% more faults than test suites generated specifically to achieve coverage. Finally, Observable MC/DC, a criterion designed to account for program structure and the selection of the test oracle, can—in part—address the failings of traditional structural coverage criteria, allowing for the generation of test suites achieving higher levels of fault detection than random test suites of equal size.

These observations point to risks inherent in the increase in test automation in critical systems, and the need for more research in how coverage criteria, test generation approaches, the test oracle used, and system structure jointly influence test effectiveness.

Index Terms—Software Testing, System Testing

1 INTRODUCTION

In software testing, the need to determine the adequacy of test suites has motivated the development of several classes of test coverage criteria [1]. One such class is *structural coverage criteria*, which measure test suite adequacy using the coverage over the structural elements of the system under test, such as statements or control flow branches. In the critical systems domain—particularly in avionics—demonstrating structural coverage is required by standards [2].

In recent years, there has been rapid progress in the creation of automated test generation tools that direct the generation process towards the satisfaction of certain structural coverage criteria [3], [4], [5], [6]. Such tools promise to improve coverage and reduce the cost associated with test creation.

In principle, this represents a success for software engineering—a mandatory, and potentially arduous, engineering task has been automated. Nevertheless, while there is evidence that using structural coverage to guide random test generation provides better tests than purely random tests [7], the effectiveness of test suites automatically generated to

satisfy various structural coverage criteria has not been firmly established.

In previous work, we found that test inputs generated specifically to satisfy three structural coverage criteria via counterexample-based test generation were *less effective* than random test inputs [8]. Additionally, we found that reducing larger test suites for a given coverage metric—in our study, MC/DC—while maintaining the same level of coverage reduced their fault finding significantly, hinting that it is not always wise to build test suites solely to satisfy a coverage criterion [9]. These findings were confirmed in a larger study, where we found that test suites generated to provide branch and MC/DC coverage were less effective than random test suites of the same size [7]. The same study found that using structural coverage as a supplement to random testing was a far more effective practice than generating tests specifically for satisfying that criterion.

More recent work suggests that a number of factors that are currently not well understood can strongly impact the effectiveness of the testing process—for example, the oracle used, the structure of the program under test, etc. [10], [11], [12]. The results of these studies indicate that adequacy criteria that do not take such factors into account may be at a disadvantage to those that do.

These results are concerning. Given that common standards in critical systems domains require test suites to satisfy certain structural coverage metrics—and the rise of automated tools that can provide such coverage—the temptation exists to automate the entire testing process. Before we can recommend

G. Gay is with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: greg@greggay.com

M. Whalen and M. Heimdahl are with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: whalen@cs.umn.edu, heimdahl@cs.umn.edu

M. Staats is with Google, Inc. E-Mail: staatsm@gmail.com

This work has been supported by NASA Ames Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583, CNS-0931931, and CNS-1035715, and the Fonds National de la Recherche, Luxembourg (FNR/P10/03). We would additionally like to thank Rockwell Collins for their support.

such action, however, is it essential to establish the efficacy of the generated test suites.

In earlier work, we reported the results of a study measuring the fault finding effectiveness of automatically generated test suites satisfying two structural coverage criteria: decision coverage and Modified Condition/Decision Coverage (MC/DC coverage), as compared to randomly generated test suites of the same size on four production avionics systems and an example system from NASA [7]. In the work presented in this paper, we have expanded the initial pilot study to cover a wider range of structural coverage criteria, with the goal of making new observations and establishing more generally applicable conclusions.

In this study, we generate tests for seven industrial critical systems—four Rockwell Collins systems from previous work, a NASA system, as well as two subsystems of a complex real-time infusion pump—using both a random test generation approach and counterexample-based test generation [13]—directed to satisfy condition, decision, MC/DC and Observable MC/DC coverage (a coverage metric designed to propagate the impact of condition choices to a program point where they can be observed by the test oracle [12]). Both the generated test suites and random test suites were reduced while maintaining coverage and compared to purely random test suites of equal size. We determined effectiveness of the resulting test suites through mutation analysis [14]—and, for two systems, a set of real faults—using two expected value test oracles: an output-only test oracle and a maximally powerful test oracle (an oracle observing all output and internal state variables).

Our results show that for three of the four coverage criteria—in four of our industrial systems—the automatically generated test suites perform *significantly worse* than randomly generated test suites of equal size (up to 40.6% fewer faults found when coupled with an output-only oracle). For the NASA example and the two infusion pump systems, which were selected specifically because their structures were significantly different from the Rockwell Collins systems, and the Observable MC/DC (OMC/DC) criterion—which was selected for its potential to overcome the shortcomings of MC/DC—test suites generated to satisfy structural coverage—performed better, matching or improving on randomly generated test suites of the same size. However, these tests were still not always effective in an absolute sense, generally finding fewer than 50% of the faults with the standard output-only oracle. Similar trends can be observed when examining systems with real faults, with coverage-directed test generation yielding up to 93.4% worse fault-detection performance.

Finally, we found that for most combinations of coverage criteria and case examples, randomly generated test suites reduced while maintaining structural coverage sometimes find more faults than pure randomly generated test suites of equal size (finding up to 13.1% more faults).

We draw three conclusions from these results. First, automatic test generation to satisfy structural coverage does not, for many of the systems investigated, yield effective tests relative to their size for the commonly-used condition, decision, and MC/DC coverage criteria. This, indicates that satisfying even a “rigorous” coverage criterion can be a poor indication of test

suite effectiveness. Furthermore, even when coverage-directed tests yield superior performance, the tests may still miss a large number of potentially severe faults.

Second, the use of structural coverage as a supplement—not a target—for test generation (as Chilensky and Miller recommend in their seminal work on MC/DC [15]) can be an effective practice.

Finally, OMC/DC, unlike other coverage criteria, generally provides the same or better fault finding than random test suites of equal size in our study, indicating that the extensions to consider propagation at least partly—though not completely—address issues related to MC/DC.

These observations have serious implications and point to the risks inherent in the increased use of test automation. The goal of this study is not to discourage the use of one particular form of automated test generation—or to recommend another—but to raise awareness of the risks of assuming that code coverage equates to effective testing. To the best of our knowledge, this paper is the largest such study to date. It demonstrates the potential for automatic test generation to reduce the fault finding effectiveness of test suites and coverage criteria relative to random testing, and it is one of the few such studies that use real-world avionics systems. While our focus is on critical systems, test generation techniques and coverage measurements are used in the verifications of systems across many disparate domains; the issues raised are relevant to those domains as well.

Our results highlight the need for more research in how the coverage criterion, the test generation approach, the chosen test oracle, and the structure of the system under test jointly influence the effectiveness of testing. The increasing availability and use of advanced test-generation tools coupled with the increased use of code coverage in certification—and our lack of knowledge of the effectiveness of such tools and metrics—is worrisome and careful attention must be paid to their use and acceptance.

2 RELATED WORK

A number of empirical studies exist comparing structural coverage criteria with random testing, with mixed results. Juristo et al. provide a survey of much of the existing work [16]. With respect to branch coverage, they note that some authors (such as Hutchins et al. [17]) find that branch coverage outperforms random testing, while others (such as Frankl and Weiss [18]) discover the opposite. Namin and Andrews have found coverage levels are positively correlated with fault finding effectiveness [19]. However, recent work from Inozemtseva and Holmes found low-to-moderate correlation when the number of test cases is controlled for and that stronger forms of coverage do not necessarily lead to stronger fault-finding results [20]. Theoretical work comparing the effectiveness of partition testing against random testing yields similarly mixed results. Weyuker and Jeng [21], and Chen and Yu [22], indicate that partition testing is not necessarily more effective than random testing. Hamlet and Taylor additionally found that partition testing—as commonly used—is often ineffective and has little value in gaining confidence in a

system [23]. Later theoretical work by Gutjahr [24], however, provides a stronger case for partition testing. Arcuri et al. [25] recently demonstrated that in many scenarios, random testing is more predictable and cost-effective at reaching high levels of structural coverage than previously thought. The authors have also demonstrated that, when cost is taken into account, random testing is often more effective at detecting faults than a popular alternative—adaptive random testing [26].

Most studies concerning automatic test generation for structural coverage criteria are focused on how to generate tests quickly and/or improve coverage [27], [3]. Comparisons of the fault-finding effectiveness of the resulting test suites against other methods of test generation are few. Gopinath et al. compared a number of manual and automatically-generated test suites for statement, block, branch, and path coverage for their ability to find fault [28]. They concluded that statement coverage led to the highest level of fault-finding effectiveness. Others that exist apart from our own limited previous work and Gopinath’s study are, to the best of our knowledge, studies in concolic execution [4], [5]. One concolic approach by Majumdar and Sen [29] has even merged random testing with symbolic execution, though their evaluation only focused on two case examples, and did not explore fault finding effectiveness.

Despite the importance of the MC/DC criterion [15], [2], studies of its effectiveness are few. Yu and Lau study several structural coverage criteria, including MC/DC, and find MC/DC is cost effective relative to other criteria [30]. Kandl and Kirner evaluate MC/DC using an example from the automotive domain, and note less than perfect fault finding [31]. Dupuy and Leveson evaluate the MC/DC as a complement to functional testing, finding that the use of MC/DC improves the quality of tests [32]. None of these studies, however, compare the effectiveness of MC/DC to that of random testing. They therefore do not indicate if test suites satisfying MC/DC are truly effective, or if they are effective merely because MC/DC test suites are generally quite large.

More concerning than the negative results regarding the ability of structural coverage to enhance fault finding is the overall lack of consensus one way or the other. Certain coverage metrics are used as though their use guarantees effective testing when, in practice, there is no universal evidence of their utility.

Our study applies counterexample-based test generation using the JKind model checker [13], [33] to directly generate test inputs for multiple coverage criteria. In this work, we find issues with the effectiveness of tests generated using this approach. Several problems that can arise when using model checkers to generate tests are discussed by Fraser et al. [34]; however, issues regarding fault-finding effectiveness are not among them. Counterexample-based test generation is simply one method of automated test generation—others include symbolic [35] and concolic execution [4], model-based test generation [36], combinatorial testing [37], and search-based testing [38], among others. For a comprehensive survey on automated test generation, see [6].

The work presented in this paper is an extension of a conference publication [7]. Our current work differs primarily in

the number of criteria explored (four, rather than two), and—in particular—the use of OMC/DC [12], a coverage criterion whose definition was in part motivated by the issues raised in our earlier work [7]. We also expand on the programs studied, and include real faults—as opposed to seeded mutations—for two of the systems.

3 EXPERIMENT

We are interested in two approaches for test generation: random test generation and directed test generation. As the name implies, in random test generation, tests are randomly generated. Suites of these tests can later be reduced with respect to the coverage criterion—this is akin to the practice of using coverage as an adequacy criterion, where one tests until coverage is achieved. Such an approach that is useful as a gauge of the value of a coverage criterion—if tests randomly generated and reduced with respect to a coverage criterion are more effective than pure randomly generated tests, we can safely conclude the use of the criterion led to the improvement. Unfortunately—other than our previous work [7]—evidence demonstrating this is, at best, mixed for coverage metrics such as decision or condition coverage [16], and non-existent for more stringent forms of coverage, such as MC/DC.

In directed test generation, tests are created specifically for the purpose of satisfying a coverage criterion. Examples include heuristic search methods [38] and approaches based on reachability [27], [3], [4]. Such techniques have advanced to the point where they can be effectively applied to production systems. Although these approaches can be slower than random testing, they offer the potential to improve the coverage of the resulting test suites.

It has been suggested that structural coverage criteria should *only* be used as adequacy metrics—to determine if a test suite has failed to cover functionality in the source code [1], [19]. However, an adequacy criterion can always be transformed into a test suite generation target. In mandating that a coverage criterion be used for measurement, it seems inevitable that some testers will opt to perform generation to speed the testing process, and tools have been built for that purpose [39].

Therefore, in our study, we aim to determine if using existing directed generation techniques with these criteria results in test suites that are more effective at fault detection than randomly generated test suites. We expect that a test suite satisfying the coverage criterion to be, at a minimum, at least as effective as randomly generated test suites of equal size. Given the central—and mandated—role the coverage criteria play within certain domains (e.g., DO-178C for airborne software [40]), and the resources required to satisfy them, this area requires additional study. We thus seek answers to the following research questions:

- RQ1: *Are random test suites reduced to satisfy various coverage criteria more effective than purely randomly generated test suites of equal size?*
- RQ2: *Are test suites directly generated to satisfy various coverage criteria more effective than randomly generated test suites of equal size?*

	# Simulink Subsystems	# Blocks
DWM_1	3,109	11,439
DWM_2	128	429
Vertmax_Batch	396	1,453
Latctl_Batch	120	718

	# States	# Transitions	# Vars
Docking_Approach	64	104	51
Infusion_Mgr	27	50	36
Alarms	78	107	60
Infusion_Mgr (faulty)	30	47	34
Alarms (faulty)	81	101	61

TABLE 1
Case Example Information

3.1 Experimental Setup Overview

In this study, we have used four industrial systems developed by Rockwell Collins Inc., a fifth system created as a case example at NASA, and two subsystems of an infusion pump created for medical device research [41]. The Rockwell Collins systems were modeled using the Simulink notation and the remaining systems using Stateflow [42], [43]; all were translated to the Lustre synchronous programming language [44] to take advantage of existing automation. In practice, Lustre would be automatically translated to C code. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). The NASA system, *Docking_Approach*, was selected due to its structure, which differs from the Rockwell Collins systems in ways relevant to this study (discussed later). *Docking_Approach* describes the behavior of a space shuttle as it docks with the International Space Station. The remaining two systems, *Infusion_Mgr* and *Alarms*, were chosen because they come with a set of real faults that we can use to assess real-world fault-finding. These systems represent the prescription management and alarm-induced behavior of an infusion pump device¹.

Information related to these systems is provided in Table 1. We list the number of Simulink subsystems, which are analogous to functions, and the number of blocks, analogous to operators. For the examples developed in Stateflow, we list the number of Stateflow states, transitions, and variables. As we have both faulty and corrected versions of *Infusion_Mgr* and *Alarms*, we list information for both.

Note that Lustre systems, and the original Simulink systems from which they were translated, operate in a sequence of steps. In each step, input is received, internal computations are done sequentially, and output is produced. Within a step, no iteration or recursion is done—each internal variable is defined, and the value for it computed, exactly once. The system itself operates as an large loop.

For each case example, we performed the following steps:

1. These two models are available to download from <http://crisys.cs.umn.edu/PublicDatasets.shtml>

- 1) **Generated mutants:** We generated 250 mutants, each containing a single fault, and removed functionally equivalent mutants. (Section 3.2.)
- 2) **Generated structural tests:** We generated test suites satisfying condition, decision, MC/DC, and Observable MC/DC coverage using counterexample-based test generation. (Section 3.4.)
- 3) **Generated random tests:** We generated 1,000 random tests of test lengths between 2-10 steps. (Section 3.4.)
- 4) **Reduced test suites:** We generated reduced test suites satisfying condition, decision, MC/DC, and OMC/DC coverage using the test data generated in the previous two step. (Section 3.5.)
- 5) **Random test suites:** For each test suite satisfying a coverage criterion, we created a single random test suite of equal size. In addition, we created test suites of sizes evenly distributed from sizes 1 to 1,000. (Section 3.5.)
- 6) **Computed effectiveness:** We computed the fault finding effectiveness of each test suite using both an output-only oracle and an oracle considering all outputs and internal state variables (a *maximally powerful* oracle) against the set of mutants and—for the infusion pump examples—against the set of real faults. (Section 3.6.)

3.2 Mutant Generation

We have created 250 *mutants* (faulty implementations) for each case example by automatically introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. Constructing the specific mutants involved a randomized process in which a list of possible mutants was enumerated. From this list, 250 mutants were selected for generation, with a roughly even distribution of fault types across the system occurring naturally.

The mutation operators used in this study are fairly typical and are discussed at length in [45]. They are similar to the operators used by Andrews et al. where they conclude that mutation testing can be an adequate proxy for real faults for the purpose of investigating test effectiveness [46].

One risk of mutation testing is *functionally equivalent* mutants—the scenario in which faults exist, but these faults cannot cause a *failure* (an externally visible deviation from correct behavior). This presents a problem when using oracles that consider internal state—we may detect failures that can never propagate to the output. We have used the JKind model checker [13] to detect and remove equivalent mutants for the four Rockwell Collins systems². This is made possible thanks to our use of synchronous reactive systems—each system is finite, and thus determining equivalence is decidable³.

The cost of determining non-equivalence for the *Docking_Approach*, *Infusion_Mgr*, and *Alarms* system is, unfortunately, prohibitive. However, for every mutant reported as killed in our study for the output-only oracle, there exists

2. The percentage of mutants removed is very small, 2.8% on average

3. Equivalence checking is fairly routine on the hardware side of the reactive system community; a good introduction can be found in [47].

at least one trace that can lead to a user-visible failure, and all fault finding measurements for that oracle indeed measure faults detected.

3.3 Real Faults

For both of the infusion pump systems—Infusion_Mgr and Alarms—we have two versions of each case example. One is an untested—but feature-complete—version with several faults, the second is a newer version of the system where those faults have been corrected. We can use the faulty version of each system to assist in determining the effectiveness of each test suite. As with the seeded mutants, effective tests should be able to surface and alert the tester to the residing faults.

For the Infusion_Mgr case example, the older version of the system contains seven faults. For the Alarm system, there are three faults. Although there are a relatively small number of faults for both systems, several of these are faults that required code changes in several locations to fix. The real faults used in this experiment are non-trivial faults—these were not mere typos or operand mistakes, require specific conditions to trigger, and extensive verification efforts were required to identify these faults. Faults of this type are ideal, as we do not want the generated test cases to trivially fail on the faulty models.

A brief description of the faults can be seen in Table 2.

3.4 Test Data Generation

In this research, we explore four structural coverage criteria: condition coverage, decision coverage, Modified Condition/Decision Coverage (MC/DC) [16], [15], and Observable Modified Condition/Decision Coverage (OMC/DC) [12].

Condition coverage is a coverage criterion based on exercising complex Boolean conditions (such as the ones present in many avionics systems). For example, given the statement $((a \text{ and } b) \text{ and } (\text{not } c \text{ or } d))$, achieving condition coverage requires tests where the individual atomic boolean conditions a , b , c , and d evaluate to true and false.

Decision coverage is a criterion concerned with exercising the different outcomes of the Boolean decisions within a program. Given the expression above, $((a \text{ and } b) \text{ and } (\text{not } c \text{ or } d))$, tests would need to be produced where the expression evaluates to true and the statement evaluated to false, causing program execution to traverse both outcomes of the decision point. Decision coverage is similar to the commonly-used branch coverage. Branch coverage is only applicable to Boolean decisions that cause program execution to branch, such as that in if or case statements, whereas decision coverage requires coverage of all Boolean decisions, whether or not execution diverges. Improving branch coverage is a common goal in automatic test generation.

Modified Condition/Decision Coverage further strengthens condition coverage by requiring that each decision evaluate to all possible outcomes (such as in the expression used above), each condition take on all possible outcomes (the conditions shown in the description of condition coverage), and that each condition within a decision be shown to independently impact the outcome of the decision. Independent

effect is defined in terms of *masking*, which means that the condition has no effect on the value of the decision as a whole; for example, given a decision of the form x and y , the truth value of x is irrelevant if y is false, so we state that x is masked out. A condition that is not masked out has *independent effect* for the decision.

Suppose we examine the independent affect of d in the example; if $(a \text{ and } b)$ evaluates to false, than the decision will evaluate to false, masking the effect of d ; Similarly, if c evaluates to false, then $(\text{not } c \text{ or } d)$ evaluates to true regardless of the value of d . Only if we assign a , b , and c true does the value of d affect the outcome of the decision.

MC/DC coverage is often mandated when testing critical avionics systems. Accordingly, we view MC/DC as likely to be effective criteria, particularly for the class of systems studied in this report. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [48].

Observable MC/DC (OMC/DC) is an enhanced version of MC/DC that requires that tests not only exercise the Boolean conditions and decisions within program expressions, but that tests also offer a path of propagation from that condition to the program output as well. One can view MCDC as determining independent affect of a condition within a decision; OMC/DC requires an analogous effect on some observable quantity for the test (such as a program output variable). While MC/DC ensures that Boolean faults will not be masked at the decision level, it is often the case that the decision will itself be masked before propagating to an output variable. OMC/DC tests have the potential to overcome this weakness by requiring that a path of propagation exists between the condition and an output. For example, consider the following block of pseudocode:

```
x = ((a and b) and (not c or d));
y = x or c;
output = y and (b or d);
```

In MC/DC it is necessary to show that the result of d influences the outcome of x . In OMC/DC, we must not only demonstrate the independent impact of d on x , but the independent impact of x on y and the independent impact of y on output —thus establishing a propagation path for $(\text{not } c \text{ or } d)$ from x to the program output.

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying the four coverage criteria [27], [3]. In this approach, each coverage obligation is encoded as a temporal logic formula and a model checker can be used to detect a counterexample (test case) illustrating how the coverage obligation can be covered. By repeating this process for each coverage obligation for the system, we can use the model checker to automatically derive test sequences that are guaranteed to achieve the maximum possible coverage of the model.

This coverage guarantee is why we have elected to use counterexample-based test generation, as other directed approaches (such as concolic/SAT-based approaches) do not offer such a straightforward guarantee. In the context of avionics systems, the guarantee is highly desirable, as achieving maximum coverage is required [2]. We have used the JKInd

Infusion_Mgr	
1	When entering therapy mode for the first time, infusion can begin if there is an empty drug reservoir.
2	The system has no way to handle a concurrent infusion initiation and cancellation request.
3	If the alarm level is ≥ 2 , no bolus should occur. However, intermittent bolus mode triggers on alarm ≤ 2 .
4	Each time step is assumed to be one second.
5	When patient bolus is in progress and infusion is stopped, the system does not enter the patient lockout. Upon restart, the patient can immediately request an additional dosage.
6	If the time step is not exactly one second, actions that occur at specific intervals might be missed.
7	The system has no way to handle a concurrent infusion initiation and pause request.

Alarms	
1	If an alarm condition occurs during the initialization step, it will not be detected.
2	The Alarms system does not check that the pump is in therapy before issuing therapy-related alarms.
3	Each time step is assumed to be one second.

TABLE 2
Real faults for infusion pump systems

model checker [13], [33] in our experiments because we have found that it is efficient and produces tests that are easy to understand [8].

For the systems with real faults, we generate tests twice. When calculating fault-finding effectiveness on generated mutants, we generate tests using the corrected version of the system (as the Rockwell Collins systems are free of known faults). However, when assessing the ability of the test suites to find the real faults, we generate the tests using the faulty version of the system. This reflects real-world practice, where—if faults have not yet been discovered—tests have been generated to provide coverage over the code as it currently exists.

We have also generated a single set of 1,000 random tests for each case example. The tests in this set are between two and ten execution steps (evenly distributed in the set). For each test step, we randomly selected a valid value for all inputs. As all inputs are scalar, this is trivial. We refer to this as a *random test suite*. After generating coverage-based test suites, we resample from this test suite to create random test suites of equal size for each coverage-based test suite. Note that as all of our case examples are modules of larger systems, the tests generated are effectively *unit tests*.

3.5 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [19], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naively generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different reduced test suites for each case example using the process described below.

Per *RQ1*, we have also created tests suites satisfying the coverage criteria by reducing the random test suite with respect to the coverage criteria (that is, the suite is reduced while maintaining the coverage level of the unreduced suite). Again, we produce 50 tests suites satisfying each coverage criterion.

For both counterexample-based test generation and random testing reduced with respect to a criterion, reduction is done using a simple greedy algorithm. We determine the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been examined in the full set of tests.

For each of our existing reduced test suites, we also produce a purely random test suite of equal size using the set of random test data. Recall that each system operates as a large loop receiving input and producing output. Each generated test is thus a finite number of “steps”, with each step corresponding to a set of inputs received by the system. We measure test suite size in terms of the number of total test steps, rather than the number of tests, as random tests are on average longer than tests generated using counterexample-based test generation. These random suites are used as a baseline when evaluating the effectiveness of test suites reduced with respect to coverage criteria. We also generate random test suites of sizes varying from 1 to 1,000 steps. These tests are not part of our analysis, but provide context in our illustrations.

When generating tests suites to satisfy a structural coverage criterion, the suite size can vary from the minimum required to satisfy the coverage criterion (generally unknown) to infinity. Previous work has demonstrated that test suite reduction can have a negative impact on test suite effectiveness [9]. Despite this, we believe the test suite size most likely to be used in practice is one designed to be small—reduced with respect to coverage—rather than large (every test generated in the case of counterexample-based generation or, even more arbitrarily, 1,000 random tests). Note that one could build a counterexample-based test suite generation tool that, upon generating a test, removes from consideration *all* newly covered obligations, and randomly selects a new uncovered obligation to try to satisfy, repeating until finished. Such a tool would produce test suites equivalent to our reduced test suites, and

thus require no reduction; alternatively, we could view such test suites as pre-reduced.

3.6 Computing Effectiveness

In our study, we use what are known as *expected value oracles* as our test oracles [49]. Consider the following testing process for a software system: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgement; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. Past experience with industrial practitioners indicates that such oracles are commonly used in testing critical systems, such as those in the avionics or medical device fields.

We explore the use of two types of oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice. Both oracles have been used in previous work, and thus we use both to allow for comparison [10], [49]. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or “killed”).

For all seven of our example systems, we assess the fault-finding effectiveness of each test suite and oracle combination by calculating the ratio of mutants killed to total number of mutants (with any known non-equivalent mutants removed).

For Infusion_Mgr and Alarms, we also assess the fault-finding effectiveness of each test suite and oracle combination against the version of the model with real faults by measuring the ratio of the number of tests that fail to the total number of tests for each test suite. We use the number of tests rather than number of real faults because all of the real faults are in a single model, and we do not know which specific fault led to a test failure. However, we hypothesize that the test failure ratio is a similar measure of the *sensitivity* of a test suite to the mutant kill ratio.

4 RESULTS AND DISCUSSION

In Tables 3, 4, 5, and 6, we present the fault finding results from our experiments when using the mutant kill ratio as the evaluation criteria. These tables list—for each case example, coverage criterion, test generation method, and oracle—the median percentage of found faults for test suites reduced to satisfy a certain criterion, next to the median percentage of found faults for random test suites of equal size⁴, the relative change in median fault finding when using the test suites satisfying the coverage criterion versus the random test suite of equal size, and the p-value for the statistical analysis below. To give an example, test suites generated to satisfy decision coverage for the Latctl_Batch system find a median of 89.3% of faults, while purely random test suites of the same size find a median of 88.1% of faults—a 1.4% improvement in fault finding. In Tables 7, 8, 9, and 10, we display the results for

⁴ “Random of Same Size” refers only to random test suites generated specifically to be the same size in terms of number of test steps as those suites reduced with respective coverage.

the systems where real faults were available. In this case, we display the median percentage of tests to fail in the generated test suites. Note that negative values for % Change indicate the test suites satisfying the coverage criterion are less effective on average than random test suites of equal size.

We present the coverage achieved by the coverage-directed test generation in Table 11 and the randomly-generated test suites in Table 12. For decision, condition, and MC/DC coverage, the random suites are able to reach or come close to reaching 100% coverage of the test obligations for the Rockwell systems. OMC/DC is a stronger coverage criterion, and it is more difficult to achieve full coverage. However, the random test suites reduced to satisfy OMC/DC are still able to come within 20 percentage points of full coverage for the Rockwell systems. Random testing is less capable of achieving coverage on the Docking_Approach, Infusion_Mgr, and Alarms systems, covering—at most—around 70% of the decision coverage obligations for Alarms.

4.1 Statistical Analysis

For both *RQ1* and *RQ2*, we are interested in determining if test suites satisfying structural coverage criteria outperform purely random test suites of equal size. We begin by formulating statistical hypotheses H_1 and H_2 :

- H_1 : A test suite generated using random test generation to provide structural coverage will find more faults—or, for real faults, find more failing test cases—than a pure random test suite of similar size.
- H_2 : A test suite generated using counterexample-based test generation to provide structural coverage will find more faults—or, for real faults, find more failing test cases—than a random test suite of similar size.

We then formulate the appropriate null hypotheses:

- H_{01} : The fault finding results of test suites generated using random test generation to provide structural coverage and pure random test suites of similar size are drawn from the same distribution.
- H_{02} : The fault finding results of test suite generated using counterexample-based test generation to provide structural coverage and random test suites of similar size are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H_{01} and H_{02} without any assumptions on the distribution of our data, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [50], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. As we cannot generalize across non-randomly selected case examples, we apply the statistical test for each pairing of case example, coverage criterion, and oracle type with $\alpha = 0.05^5$.

⁵ Note that we do not generalize across case examples, oracles or coverage criteria, as the needed statistical assumption, random selection from the population of case examples, oracles, or coverage criteria, is not met. The statistical tests are used to only demonstrate that observed differences are unlikely to have occurred by chance.

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying Condition	Random of Same Size	% Change	p-val	Satisfying Condition	Random of Same Size	% Change	p-val
Latctl_Batch	MX	93.8%	97.9%	-4.2%	1.00	98.8%	97.1%	1.7%	< 0.01
	OO	48.6%	83.5%	-41.9%		81.5%	80.2%	1.5%	0.06
Vertmax_Batch	MX	100.0%	95.6%	4.6%	< 0.01	100.0%	96.0%	4.2%	< 0.01
	OO	45.6%	76.2%	-40.2%		81.9%	75.8%	7.9%	
DWM_1	MX	92.8%	92.8%	0.0%	0.61	97.5%	96.6%	0.8%	0.35
	OO	18.2%	25.0%	-27.1%		34.3%	33.1%	3.6%	
DWM_2	MX	94.7%	92.6%	2.3%	< 0.01	99.2%	94.2%	5.2%	< 0.01
	OO	75.7%	77.8%	-2.6%		88.9%	82.7%	7.5%	
Docking Approach	MX	70.7%	19.4%	264.4%	< 0.01	18.5%	18.5%	0.0%	0.79
	OO	21.7%	2.0%	985.0%		2.0%	2.0%	0.0%	1.00
Infusion_Mgr	MX	68.6%	25.1%	173.3%	< 0.01	20.7%	20.6%	0.5%	0.11
	OO	27.0%	10.1%	167.3%		7.3%	7.3%	0.0%	0.42
Alarms	MX	74.9%	47.4%	58.0%	< 0.01	47.0%	47.0%	0.0%	0.46
	OO	40.5%	14.6%	177.4%		15.0%	14.2%	5.6%	< 0.01

TABLE 3

Median percentage of faults identified, condition coverage criterion. OO = Output-Only, MX = Maximum Oracle

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying Decision	Random of Same Size	% Change	p-val	Satisfying Decision	Random of Same Size	% Change	p-val
Latctl_Batch	MX	89.3%	88.1%	1.4%	< 0.01	97.9%	95.9%	2.1%	< 0.01
	OO	34.2%	57.6%	-40.6%		80.7%	77.8%	3.7%	
Vertmax_Batch	MX	85.1%	70.6%	20.5%	< 0.01	87.9%	84.3%	4.3%	< 0.01
	OO	31.0%	41.12%	-24.25%		61.7%	57.3%	7.7%	
DWM_1	MX	82.6%	97.0%	-14.6%	1.00	97.5%	96.2%	1.3%	0.03
	OO	13.1%	32.2%	-59.2%		33.5%	32.6%	2.6%	
DWM_2	MX	80.2%	88.1%	-8.9%		94.7%	91.8%	3.1%	< 0.01
	OO	48.1%	70.8%	-31.9%		80.1%	77.4%	4.3%	
Docking Approach	MX	67.5%	19.4%	247.9%	< 0.01	18.5%	18.5%	0.0%	0.90
	OO	20.1%	2.0%	905%		2.0%	2.0%	0.0%	1.00
Infusion_Mgr	MX	65.2%	23.5%	177.4%		20.7%	19.8%	4.5%	0.24
	OO	25.5%	8.9%	186.5%		6.9%	6.9%	0.0%	0.54
Alarms	MX	74.9%	47.8%	56.7%	< 0.01	47.0%	47.0%	0.0%	1.00
	OO	35.7%	14.6%	144.5%		14.6%	13.8%	5.8%	< 0.01

TABLE 4

Median percentage of faults identified, decision coverage criterion. OO = Output-Only, MX = Maximum Oracle

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying MC/DC	Random of Same Size	% Change	p-val	Satisfying MC/DC	Random of Same Size	% Change	p-val
Latctl_Batch	MX	96.7%	99.6%	-2.3%	1.00	99.6%	98.8%	0.8%	< 0.01
	OO	79.8%	93.4%	-14.5%		89.7%	87.7%	2.3%	
Vertmax_Batch	MX	100.0%	96.4%	3.8%	< 0.01	100.0%	95.2%	5.1%	< 0.01
	OO	59.3%	78.2%	-24.2%		81.9%	76.6%	6.8%	
DWM_1	MX	88.6%	97.5%	-9.1%	1.00	97.9%	97.5%	0.4%	0.45
	OO	18.6%	36.0%	-48.2%		34.7%	34.3%	1.2%	
DWM_2	MX	96.3%	96.3%	0.0%	0.83	99.6%	97.1%	2.5%	< 0.01
	OO	79.8%	86.0%	-7.2%		90.9%	88.1%	3.3%	
Docking Approach	MX	72.3%	19.4%	272.7%	< 0.01	18.5%	18.5%	0.0%	0.62
	OO	23.3%	2.0%	1065.0%		2.0%	2.0%	0.0%	1.00
Infusion_Mgr	MX	69.6%	24.7%	44.9%		20.6%	21.9%	-1.3%	0.99
	OO	31.6%	11.3%	20.3%		6.9%	7.3%	-0.4%	0.02
Alarms	MX	78.9%	47.8%	65.1%	< 0.01	47.8%	47.4%	0.8%	0.01
	OO	40.5%	14.6%	177.4%		15.0%	14.6%	2.7%	< 0.01

TABLE 5

Median percentage of faults identified, MC/DC criterion. OO = Output-Only, MX = Maximum Oracle

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying OMC/DC	Random of Same Size	% Change	p-val	Satisfying OMC/DC	Random of Same Size	% Change	p-val
Latctl_Batch	MX	99.2%	99.2%	0.0%	1.00	99.6%	99.6%	0.0%	0.28
	OO	95.3%	90.1%	5.8%	< 0.01	97.1%	95.9%	1.3%	< 0.01
Vertmax_Batch	MX	99.6%	96.4%	3.3%		99.2%	97.6%	1.6%	
	OO	97.5%	77.4%	26.0%		90.7%	80.2%	13.1%	
DWM_1	MX	100.0%	100.0%	0.0%	0.16	100.0%	100.0%	0.0%	1.00
	OO	90.5%	83.5%	8.4%	< 0.01	97.5%	92.8%	5.1%	< 0.01
DWM_2	MX	98.7%	97.5%	1.2%		100.0%	100.0%	0.0%	
	OO	95.7%	88.9%	7.6%		98.4%	96.7%	1.8%	
Docking Approach	MX	73.9%	19.4%	280.9%	< 0.01	19.4%	19.4%	0.0%	0.56
	OO	26.9%	2.0%	1245.0%		2.0%	2.0%		1.00
Infusion_Mgr	MX	70.0%	27.5%	154.5%	< 0.01	23.1%	23.1%	0.0%	0.20
	OO	43.3%	12.1%	257.9%		8.5%	8.5%		0.19
Alarms	MX	81.0%	48.2%	68.0%	< 0.01	48.6%	48.6%	0.0%	0.02
	OO	58.3%	15.4%	278.6%		15.4%	15.4%		0.81

TABLE 6
Median percentage of faults identified, OMC/DC criterion. OO = Output-Only, MX = Maximum Oracle

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying Condition	Random of Same Size	% Change	p-val	Satisfying Condition	Random of Same Size	% Change	p-val
Infusion_Mgr	MX	55.8%	36.7%	52.0%	< 0.01	42.9%	36.4%	17.9%	0.16
	OO	10.5%	29.0%	-63.8%	1.00	35.7%	30.0%	19.0%	0.07
Alarms	MX	93.0%	93.8%	-0.9%	0.98	92.9%	93.0%	-0.1%	0.70
	OO	91.2%		-2.8%	1.00				

TABLE 7
Median percentage of tests failed after identifying real faults, condition coverage criterion.

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying Decision	Random of Same Size	% Change	p-val	Satisfying Decision	Random of Same Size	% Change	p-val
Infusion_Mgr	MX	42.3%	40.0%	5.8%	< 0.01	33.3%	33.3%	0.0%	0.44
	OO	6.3%	31.8%	-80.2%	1.00	28.6%	28.6%	0.0%	0.26
Alarms	MX	92.6%	94.3%	-1.8%	1.00	93.8%	94.1%	-0.3%	0.33
	OO	90.4%		-4.1%					

TABLE 8
Median percentage of tests failed after identifying real faults, decision coverage criterion.

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying MC/DC	Random of Same Size	% Change	p-val	Satisfying MC/DC	Random of Same Size	% Change	p-val
Infusion_Mgr	MX	46.0%	34.4%	33.7%	< 0.01	33.3%	30.0%	11.0%	0.02
	OO	1.8%	27.3%	-93.4%	1.00	30.0%	25.0%	20%	0.05
Alarms	MX	94.6%	93.8%	0.9%	0.05	95.2%	94.1%	1.2%	0.02
	OO	94.4%		0.6%	0.36				

TABLE 9
Median percentage of tests failed after identifying real faults, MC/DC criterion.

Case Example	Oracle	Counterexample Generation Comparison				Random Generation Comparison			
		Satisfying OMC/DC	Random of Same Size	% Change	p-val	Satisfying OMC/DC	Random of Same Size	% Change	p-val
Infusion_Mgr	MX	47.9%	38.3%	25.1%	< 0.01	42.1%	35.3%	19.3%	< 0.01
	OO	40.4%	30.3%	33.3%		30.0%	28.6%	4.9%	0.09
Alarms	MX	93.0%	94.4%	-1.5%	1.00	92.4%	94.7%	-2.4%	1.00
	OO	40.4%		-57.2%					

TABLE 10
Median percentage of tests failed after identifying real faults, OMC/DC criterion.

	Decision Coverage	Condition Coverage	MC/DC Coverage	OMC/DC Coverage
DWM_1	100.00%	100.00%	100.00%	99.90%
DWM_2	100.00%	100.00%	95.28%	89.79%
Vertmax_Batch	100.00%	100.00%	100.00%	98.15%
Latctl_Batch	100.00%	100.00%	100.00%	93.42%
Docking_Approach	97.94%	95.85%	91.94%	50.86%
Infusion_Mgr	92.06%	93.56%	92.36%	56.06%
Alarms	91.85%	93.17%	91.32%	76.24%

TABLE 11
Coverage Achieved by Coverage-Directed Test Suites Reduced to Satisfy Coverage Criteria

	Decision Coverage	Condition Coverage	MC/DC Coverage	OMC/DC Coverage
DWM_1	100.00%	100.00%	100.00%	99.85%
DWM_2	100.00%	100.00%	93.15%	89.25%
Vertmax_Batch	100.00%	99.83%	99.40%	84.18%
Latctl_Batch	100.00%	100.00%	100.00%	89.21%
Docking_Approach	58.10%	56.90%	48.59%	2.20%
Infusion_Mgr	62.77%	62.33%	48.86%	11.16%
Alarms	69.33%	68.85%	64.22%	32.72%

TABLE 12
Coverage Achieved by Randomly Generated Test Suites Reduced to Satisfy Coverage Criteria

4.2 Evaluation of RQ1

Based on the p-values less than 0.05 in Tables 3-6, we *reject* H_0_1 for nearly all of the four Rockwell case examples and the respective coverage criteria when using either oracle. Note that we do not reject H_0_1 for the *DWM_1* case example when using decision, condition, and MC/DC coverage and the output-only oracle. That is, the use of coverage as a supplement to random testing—using coverage to decide when to stop random testing—leads to improved fault-finding results. However, for the majority of coverage criterion and oracle combinations for the *Docking_Approach*, *Alarms*, and *Infusion_Mgr* systems, we fail to reject H_0_1 . For many of these combinations, there is no evidence of improvement from using coverage as an adequacy metric for random testing. Across all system, for cases with differences that are statistically significant, test suites reduced to satisfy coverage criteria are clearly more effective than purely randomly generated test suites of equal size—for these combinations, we accept H_1 .

The difference in results between the four Rockwell Collins systems and the *Docking_Approach*, *Alarms*, and *Infusion_Mgr* systems—when examining mutations—can be somewhat explained through the coverage achieved by random tests on those systems. As can be seen in Table 12, random testing is able to cover almost all of the obligations for the four coverage criteria on the Rockwell systems. In those cases, we can use coverage as a method of guiding the selection of tests. We can cut off random testing once coverage is achieved, filter out superfluous tests, and potentially present a small, powerful test suite. On the other systems, however, random testing is unable—even after generating our full pool of 1000 tests—to achieve full coverage for any of the coverage criteria. In those cases, it is hardly surprising that using coverage to guide the selection of random tests fails to lead to an improvement in fault-finding. Even if there is potential power to be gained from the guidance of coverage, failing to achieve coverage will also imply failing to gain that predictive power. It may well be that

any suite of random tests of the same size will result in similar coverage and fault-finding, and using coverage to choose those suites may be no more effective than choosing from the pool of tests at random.

When evaluating test suite effectiveness against the set of real faults for the *Infusion_Mgr* and *Alarms* systems, we see similar results—see tables 7-10. We reject H_0_1 for the MC/DC and OMC/DC criteria and the maximum oracle and the MC/DC criteria for the output-only oracle for the *Infusion_Mgr* system. However, we fail to reject H_0_1 for the remaining criteria and oracle combinations for these two systems. Note that, in several cases, we do see an improved median fault-finding result, but we do not see a corresponding achievement of statistical significance. In those cases, there is a large amount of variance in the results from the random test suites. While the median case has improved, the the distribution of results has not changed. Samples from the distributions of random tests guided by coverage are not significantly better at finding faults than samples from the distribution of purely random tests. In some cases, we see a small improvement in fault-finding effectiveness when we use coverage as an adequacy criteria and, even in the other cases, using coverage as an adequacy metric does not result in worse fault-finding results.

From our results, we can weakly confirm that all four of these coverage criteria *can* be effective metrics for test suite adequacy within the domain of critical avionics systems: reducing test suites generated via a non-directed approach to satisfy structural coverage criteria is at least not harmful, and in some instances improves test suite effectiveness relative to their size by up to 13%. Thus, the use of structural coverage as an adequacy criteria for random testing can potentially lead to a positive, albeit slight, improvement in test suite effectiveness. This indicates that the core intuitions behind these coverage metrics—i.e., covering branches, conditions and combinations of conditions—appear valid, and thus, given

a constant generation strategy, covering code yields benefits as long as coverage is actually achieved.

4.3 Evaluation of RQ2

Based on the p-values less than 0.05 in Tables 3- 5, we fail to reject H_02 for the four Rockwell Collins case examples and the decision, condition, and MC/DC coverage criteria when using the output-only oracle. For all but one of these case examples, test suites generated via counterexample-based test generation are *less* effective than pure random test suites by 2.6% to 59.2%; we therefore conclude that our initial hypothesis H_2 is false—at least, with respect to the Rockwell systems—with regard to decision, condition, and MC/DC coverage when using an output-only oracle.

When using the maximum oracle, the test suites generated via counterexample-based test generation to satisfy decision, condition, and MC/DC coverage fare better. In select instances, countereexample-based test suites outperform random test suites of equal size (notably *Vertmax_Batch*), and otherwise close the gap, being less effective than pure random test suites by at most 14.6%. Nevertheless, we note that for most combinations of the Rockwell case examples and those three coverage criteria, random test suites of equal size are still more effective. When these criteria are used as targets for test generation, the test suites produced are generally *less* effective than random testing alone, with decreases of up to 59.2%.

This indicates that decision, condition, and MC/DC coverage are—by themselves—not necessarily good indicators of test suite effectiveness; factors other than coverage impact the effectiveness of the testing process. In contrast, to the more traditional structural coverage criteria—notably MC/DC coverage—results for OMC/DC coverage are more positive in terms of the value of directed test generation. From the p-values in Table 6, we reject H_02 for all case examples except *Latctl_Batch* with the maximum oracle and *DWM_1* with the maximum oracle when examining the mutation-based faults. It appears that test suites generated to satisfy OMC/DC coverage are more effective than purely random tests of the same size, and—as when using coverage as a supplemental criteria for random testing—generating tests in order to satisfy OMC/DC is no worse than just constructing random tests. Thus, we can see that considering how covered obligations propagate to observable points can yield dividends during test generation.

The converse of H_2 —that randomly generated test suites are more effective than equally large test suites generated via counterexample-based test generation—is also not universally true, as the *Docking_Approach*, *Alarms*, and *Infusion_Mgr* examples illustrate. For the *Docking_Approach* example, random testing is effectively useless, finding a mere 2% of the faults on average when using an output-only oracle and 19.4% with the maximum oracle. Similarly, for the *Alarms* and *Infusion_Mgr* systems, the use of counterexample-based tests does improve fault-finding effectiveness by up to 278.6% over random test suites of similar size. However, it should be noted that *improved* fault-finding is not always the same as *good* fault-finding. The maximum oracle finds up to 81% of the faults for the *Alarms* system; however, maximum oracles

are often prohibitively expensive to employ, as they require a specification of correctness for all variables. The output-only oracle, a far more common option [49], only manages to find slightly over half of the faults for a single case example and coverage combination—OMC/DC testing on the *Alarms* system. There is clearly room for improvement.

Contrasting the performance of counterexample-based test generation and random test generation on the systems with real faults yields a number of observations. From the results for *Infusion_Mgr* in Table 7-10, we can see that the use of coverage-directed test generation yields a higher percentage of failing tests for the maximum oracle. However, for the output-only oracle, the opposite is true—random tests are far more capable at detecting faults than the coverage-directed tests. This difference is likely due to *masking*—some expressions in the systems can easily be prevented from influencing the outputs. When covered expressions do not propagate to an observable output, faults cannot be observed. The coverage-directed tests tend to be short, one or two test steps at most. The real faults embedded within this system require specific combinations of events to trigger, and may take some time before they influence the output. As a result, the coverage-based tests may trigger more of the difficult-to-reach faults, but are not long enough for the effects of those faults to influence the outward behavior of the system. The random tests, on the other hand, tend to be longer (up to ten steps), which may be long enough that many of the faults do propagate to the system output.

On the *Alarms* system, the random tests are more capable of detecting faults than the coverage-directed tests, with the exception of the MC/DC criterion. For the decision and condition coverage criteria, this difference is relatively small, up to a 4.1% difference. However, for the OMC/DC criterion, random testing is far more capable at detecting the embedded faults. This result can be explained by examining the type of faults that exist in this system, as listed in Table 2. In particular, the first fault—that if an alarm condition occurs during the first initialization step of execution, it will not be detected in the faulty version of the system—helps to explain the particular results that were observed. The coverage criteria employed in this experiment all, to a varying degree, require that certain combinations of Boolean conditions are satisfied. As a result, the generated tests will be biased towards particular input values. In many cases, these input values would not trigger alarm conditions immediately upon system activation. The random tests, on the other hand, tend towards extreme input values (or, at least, make use of the full range of possible input combinations) and, as a result, trigger this particular fault in almost *every* test case. As a result, random testing results in a higher percentage of failing tests, but the majority of those failures are due to the fault in the initialization step.

Another factor that helps explain the differing results on the version of *Alarms* with real faults is that coverage-directed test generation is unable to achieve a level of OMC/DC coverage as high as that achieved by MC/DC, condition, and decision coverage—tests generated on the faulty version of *Alarms* only achieve 72% OMC/DC coverage, whereas 88% MC/DC coverage is achieved. This means that some portion of the state

space *is not being explored* by the OMC/DC-directed tests. If the uncovered state space overlaps with one of the embedded faults, then the existing OMC/DC tests may not execute the tests in such a way that the fault will be triggered.

The results when examining real faults differ from those seen for the Infusion_Mgr and Alarms systems when examining seeded mutations. In the latter case, coverage-directed generation yielded stronger tests. Often, in the former case, the randomly-generated tests yielded stronger results. This shift can likely be explained by examining the types of faults seeded in both scenarios. The seeded mutations are all code-based errors—using the wrong operation, changing a constant, using a stored value for a variable from a previous computation. However, the real faults, listed in Table 2, tend to be more *conceptual* in nature. Largely, the real faults are problems of omission—the developers forgot to implement a feature or the system specification left an outcome ambiguous. Code coverage cannot be expected to account for code that does not exist, and thus, is unlikely to yield tests that account for such faults. This explains the different results for these systems when switching from seeded faults to real faults—coverage-directed tests can help find faults when the faults are the result of mistakes in the code that is being exercised, but are not guaranteed to be effective when faults are due to conceptual mistakes.

4.4 Implications

Given the important role of structural coverage criteria in the verification and validation of safety-critical avionics systems, we find these results quite troublesome. In the remainder of this section, we discuss the immediate practical implications of this as well as the implications for future work. We begin by discussing why traditional structural coverage criteria fare poorly when used as targets for test generation. We have identified several factors that contribute to this, including the formulation of structural coverage criteria; the behavior of the test generation mechanism (in this case, software model checkers); and structural properties of the case examples.

First and—given the differences observed with OMC/DC coverage—foremost, we note that traditional coverage criteria are formulated over specific elements in the source code. For each element, (1) execution must reach the element and (2) exercise the element in a specific way. This type of formulation falls short in two ways. First, it is possible to change the number and structure of each element by varying the structure of the program, which we have previously seen can significantly impact the number of tests required to satisfy the MC/DC coverage criterion [12], [51]. This is linked partly to the second issue, *masking*—some expressions in the systems can easily be prevented from influencing the outputs. When covered expressions do not propagate to an observable output (i.e., do not reach a test oracle variable), faults cannot be observed. This reduces the effectiveness of any testing process based on structural coverage criteria, as we can easily satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

Issues related to masking are in turn exacerbated by automated test generation, bringing us to our second factor. We

have found that test inputs generated using counterexample-based generation (including those in this study) tend to be short, and manipulate only a handful of input values, leaving other inputs at default values (in our case, `false` or 0) [8]. Such tests tend to exercise the program just enough to satisfy the coverage obligations for which they were generated and do not consider the propagation of values to the outputs. In contrast, random tests can vary arbitrarily in length (up to 10 steps in this study) and vary all input values; such test inputs may be more likely to overcome any masking present in the system. Rather than pointing to this as a strength of random testing, we would like to emphasize that this is a *weakness* of the coverage-directed test generation method. The use of coverage cannot be assumed to guarantee effective tests—the particulars of the method of test generation appear to have a greater impact at present.

Finally, the structure of the case examples themselves—being fairly representative of case examples within this domain—is also partly at fault. Recall that when testing the Docking_Approach, Alarms, and Infusion_Mgr systems, tests generated to satisfy structural coverage criteria sometimes dramatically outperform random test generation. This is due to the structure of these systems: large portions of these system’s behavior are activated only when very specific conditions are met. As a result the state space is both deep and narrow at multiple points, and exploration of these deep states requires relatively long tests with specific combinations of input values. Random testing is therefore highly unlikely to reach much of the state space, and indeed, less than 50% of the MC/DC obligations were covered for Docking_Approach. In contrast, the Rockwell Collins systems (while stateful) have a state space that is shallow and highly interconnected; these systems are therefore easier to cover with random testing and, thus, the potential benefits of structural coverage metrics are diminished.

It is these issues—particularly the first two issues—that motivated the development of the OMC/DC coverage criterion. Observable MC/DC coverage explicitly avoids issues related to masking by requiring in its test obligations both demonstrate the independent impact of a condition on the outcome of the decision statement, and follow a non-masking path to some variable monitored by the test oracle. Consequently, test suites generated via counterexample-based test generation to satisfy OMC/DC outperform purely randomly generated test suites of equal size by up to 42.5%.

This represents a major improvement over existing structural coverage criteria, though we still urge caution. The high cost and difficulty of generating OMC/DC satisfying test suites—relative to generating the weaker decision, condition, or MC/DC test suites—makes its use as a target for directed test suite generation less likely at this point in time. Additional work demonstrating the cost effectiveness (and perhaps specific improvements for test generation) will be necessary before OMC/DC coverage can replace MC/DC coverage.

We see three key implications in our results. First, with regard to **RQ1**, we can weakly conclude that using any of these four structural coverage criteria as an addition to another non-structure-based testing method—in this case, random testing—

can potentially yield improvements in the testing process. These results are similar to those of other authors, for example, results indicating MC/DC is an effective coverage criterion when used to check the adequacy of manual, requirement-driven test generation [32] and results indicating that reducing randomly generated tests with respect to decision coverage yields improvements over pure random test generation [19]. These results, in conjunction with the results for **RQ2**, reinforce the advice that coverage criteria are best applied after test generation to find areas of the source code that have not been tested. In the case of MC/DC this advice is already explicitly stated in regulatory requirements and by experts on the use of the criterion [2], [15].

Second, the dichotomy between the *Docking_Approach*, *Alarms*, and *Infusion_Mgr* examples and the Rockwell Collins systems highlights that, while the current methods of determining test suite adequacy in avionics systems are themselves largely inadequate, some method of determining testing adequacy is needed. While current practice recommends that coverage criteria should be applied after test generation, in practice, this relies on the honesty of the tester (it is not required in the standard). Therefore, it seems inevitable that at least some practitioners will use automated test generation to reduce the cost of achieving the required coverage.

The lack of consensus in the results across these varied systems is concerning in light of this inevitability. While coverage-directed test generation *can* lead to effective testing, there is no evidence that it can do so consistently. With the importance given to coverage criteria in the avionics industry, the temptation exists to rely on coverage as an assurance of reliable and thorough testing. However, we stress that blind faith in the power of code coverage is a risky proposition at best in light of the inconsistency of the results in this study.

Finally, our results with OMC/DC coverage indicate that it is possible to extend existing coverage to overcome some of the issues we have highlighted above. The primary problem with existing structural coverage criteria is that all effort is expended on covering structures internal to the system, and no further consideration is paid to how the effect of covered structure reaches an observable point in the system. OMC/DC considers the observability aspect for Boolean expressions (using MC/DC) by appending a path condition onto each test obligation in MC/DC. Similar extensions could be applied to a variety of other existing coverage metrics, e.g., boundary value testing.

5 RECOMMENDATIONS

Assuming our results generalize, we believe that these studies raise serious concerns regarding the efficacy of coverage-directed automated testing. The tools are not at fault: we have asked these tools to produce test inputs satisfying some form of structural coverage, and they have done so admirably; for example, satisfying MC/DC for the *Docking_Approach* example, for which random testing achieves a mere 37.7% of the possible coverage. However, our results—along with the existing body of research on this topic—lead us to conclude that it is not sufficient to simply maximize a structural

coverage metric when automatically generating test inputs. In practice, the mechanism for achieving coverage is important. *How* tests are generated matters more than what is being maximized.

The key issues involve how structural coverage criteria are formulated and how automatic test generation tools operate. Traditional coverage criteria are formulated over specific elements in the source code. To cover an element, (1) execution must reach the element and (2) exercise the element in a specific way. However, commonly used structural coverage criteria typically leave a great deal of leeway to how the element is reached, and—more importantly, in our experience—place no constraints whatsoever on how the test should evolve after the element is exercised. Automatic test generation tools typically use this freedom to do just enough work to satisfy coverage criteria, without consideration of, for example, how the faults are to be detected by the test oracle.

First, let us consider the path to satisfy a coverage obligation, e.g., a branch of a complex conditional. Structural coverage criteria require only that the point of interest is reached and exercised. We have found that automatically generated tests often take a shortest-path approach to satisfying test obligations, and manipulate only a handful of input values, leaving other inputs at default values. This is a cost effective method of satisfying coverage obligations; why tinker with program values that do not impact the coverage achieved? However, we, and other authors, have observed that variations provided by (for example) simple random testing result in test suites that are nearly as effective in terms of coverage, and moreover produce more interesting behavior capable of detecting faults [25], [7]. As illustrated by our experiments, however, even with lower coverage achieved, randomly generated test inputs can outperform automatically generated test suites in terms of fault finding.

Second, for the most commonly used structural coverage criteria, there is no directive concerning how tests should evolve after satisfying the structural element. Given this lack of direction, automatic test generation tools typically do not consider the path from the covered element to an output/assertion/observable state when generating a test, and therefore test inputs may achieve high coverage but fail to demonstrate faults that exist within the code. One reason for this failure is *masking*, which occurs when expressions/computations in the system are prevented from influencing the observed output, i.e., do not reach a variable or assertion monitored by the test oracle. More generally, this is related to the distinction between incorrect program state and program output: just because a test triggers a fault, there is no guarantee this will manifest as a *detected* fault. Indeed, in our experience, care must be taken to ensure that this occurs.

These two high level issues result in the generation of test inputs that may indeed be effective at encountering faults, but may make actually observing them—that is, actually detecting the fault—very difficult or unlikely. This reduces the effectiveness of any testing process based on structural coverage criteria, as we can easily satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

Furthermore, even when the generated tests are more effective than the randomly-generated tests, these tests may still not actually yield *good* fault-finding performance. On the Alarms, Infusion_Mgr, and—in particular—Docking_Approach, the generated tests commonly found fewer than half of the seeded faults when using a common output-only oracle, and less than 80% of the faults with the prohibitively expensive maximum oracle.

We believe that these issues raise serious concerns about the efficacy of coverage-directed automated testing. A focus in automated test generation work has been on efficiently achieving coverage without carefully considering how achieving coverage impacts fault detection. We therefore run the risk of producing tools that are satisfying the letter of what is expected in testing, but not the spirit. Nevertheless, the central role of coverage criteria in testing is unlikely to fade, as demonstrated by the emphasis on coverage criteria for certification.

Hence, the key is to improve upon the base offered by these criteria and existing technology. We have come to the conclusion that the research goal in automated test generation should not be developing methods of maximizing structural code coverage, but rather determining how to *maximize fault-finding effectiveness*. We propose that algorithms built for test generation must evolve to take into account factors beyond the naive execution of individual elements of the code—factors such as masking, program structure, and the execution points monitored by the test oracle.

To that end, we recommend three approaches. First, we could improve, or replace, existing structural coverage criteria, extending them to account for factors that influence test quality. Automated test generation has improved greatly in the last decade, but the targets of such tools have not been updated to take advantage in this increase in power. Instead, we continue to rely on criteria that were originally formulated when manual test generation was the only practical method of ensuring 100% achievable coverage.

Second, automated test generation tools could be improved to avoid pitfalls when using structural coverage criteria. This could take many forms, but one straightforward approach would be to develop heuristics or rules that could operate alongside existing structural coverage criteria. For instance, tools could be encouraged to generate longer test cases, increasing the chances that a corrupted internal state would propagate to an observable output (or other monitored variable).

Third, important factors specific to individual domains, e.g., web testing v.s. embedded systems, could be empirically identified and formalized as heuristics within a test generation algorithm.

5.1 Use More Robust Coverage Criteria

In our own work, the issues of criteria formulation and masking motivated the development of the Observable MC/DC coverage criterion employed in this case study [12]. OMC/DC coverage explicitly avoids issues related to masking by requiring that its test obligations both demonstrate the independent impact of a condition on the outcome of the decision

statement, and follow a non-masking propagation path to some variable monitored by the test oracle. OMC/DC, by accounting for the program structure and the selection of the test oracle, can, in our experience, address some of the failings of traditional structural coverage criteria within the avionics domain, allowing for the generation of test suites achieving better fault detection than random test suites of equal size. Similarly, Vivanti et al. have demonstrated evidence that the use of data-flow coverage as a goal for test generation results in test suites with higher fault detection capabilities than suites generated to achieve branch coverage [52]. OMC/DC considers the observability aspect for Boolean expressions (using MC/DC) by appending a path condition onto each test obligation in MC/DC. Similar extensions could be applied to a variety of other existing coverage metrics, e.g., boundary value testing.

5.2 Algorithmically Improve Test Selection

Extensions to coverage criteria are not without downsides: for stronger metrics, programs will contain *unsatisfiable* obligations where there is no test that can be constructed to satisfy the obligation. Depending on the search strategy, the test generator may never terminate on such obligations. Further, the cost and difficulty of generating OMC/DC satisfying test suites—relative to generating the weaker MC/DC test suites—makes the use of strong coverage criteria as targets for test generation harder to universally recommend.

Instead, another possible method of ensuring test quality is to use a traditional structural coverage metric as the objective of test generation, and augment this by considering other factors empirically established to impact fault detection effectiveness. For example, in the context of a search-based test generation algorithm, this might mean adding additional objective functions to the search strategy, rather than adding additional constraints to the coverage criterion. For instance, an algorithm could both work to maximize an existing structural coverage criterion and minimize the propagation distance between the assignment of a value and its observation by the oracle (an algorithm that minimizes this distance for test prioritization purposes already exists [53]).

As an example, consider purely random testing. Random testing does not employ an objective function, but it is possible to use one to approximate how well the system’s state space is being covered. We have seen systems containing bottlenecks (pinch-points) in the state space where randomly generated tests perform very poorly. Such bottlenecks require certain specific input sequences to reach a large portion of the state space. However, approaches such as concolic testing [4]—combining random testing with symbolic execution—are able to direct the generation of tests around such bottlenecks. Similar approaches could be employed to direct test generation towards, for example, propagation paths from variable assignment to oracle-monitored portions of the system.

By pursuing and balancing multiple objectives, we could potentially offer stronger tests that both satisfy MC/DC obligations and offer short propagation paths, even when it would be impossible to generate a test that satisfies the corresponding—but stricter—OMC/DC obligation. Embedding aspects of test

selection into the objective function—or even into the search algorithm itself—may allow for improved efficiency. The search method can use, say, masking as a pruning mechanism on paths through the system, and the algorithm would not have to track as much symbolic information related to the objective metric itself.

5.3 Tailor an Approach to the Domain

It is important to emphasize that there is no “one size fits all” solution to test generation. The size and shape of the state space of a system varies dramatically between domains and programming paradigms, and, as a result, it is difficult to tailor universal testing strategies. Much of our work has focused on embedded systems that run as cyclic processes. In this area, a common issue is that the impact of exercising a code path on the system’s output is often delayed; only several cycles after a fault occurs can we observe it. If the goal of test generation is only to cause the code path to be executed, many of the tests will not cause a visible change in system behavior. In object-oriented systems, a central, but related issue is that a method call may change internal state that, again, is not visible externally until another method call produces output. As a result, choosing appropriate method sequences and their ordering becomes a major challenge.

Thus while general rules and heuristics for improving test generation are valuable, we believe there are large improvements to be found in tailoring the approach to the testing challenge at hand. For example, two often overlooked factors are the cost of generating tests and the cost of running tests. It is hard to outperform random testing in terms of the cost of generating tests, because doing so requires very little computation. If it is also cheap to run tests, then for many systems it is difficult to outperform straight random testing. On the other hand, if it is expensive to run tests, e.g., for embedded systems, this may require access to a shared hardware “rig.” In this case, using search-based techniques to generate tests for a specific strong coverage criterion (such as OMC/DC) may be sensible because the number of required tests can be dramatically smaller than the number of random tests required to achieve the same level of fault finding.

Another overlooked factor is the “reasonableness” of generated tests. Automated test generation methods should deliver tests that not only find faults, but are also meaningful to the domain of interest. Coverage-based techniques take the path of least resistance when generating tests, but can produce tests that make little sense in the context of the domain. Techniques that can generate inputs with meaning to the human testers are valuable in reducing the “human oracle cost” associated with checking failing test results [54].

Addressing factors such as these must be done on a per domain level, and indicate that code coverage should be one of several goals in test generation. We suspect that, in the long run, effective test generation tools will consist of both general techniques and heuristics, and additional *test generation profiles* for each domain. Determining the *correct* objective functions for test generation for each domain is an open research question, one requiring both technical advancements in search-based test generation and empirical studies.

6 THREATS TO VALIDITY

External Validity: Our study has focused on a relatively small number of systems but, nevertheless, we believe the systems are representative of the critical systems domain, and our results are generalizable to other systems in that domain.

We have used two methods for test generation (random generation and counterexample-based). There are many methods of generating tests and these methods may yield different results. Counterexample-based testing is used to produce coverage-directed test cases because it is a method used widely in testing safety-critical systems. Random testing is used as a contrasting baseline because it is one of the most simple test generation methods in existence. Because the length of test inputs and the values selected for the input variables are chosen at random, we believe that random generation is a fair baseline—it has not been tuned to systems in this domain and has no particular strengths or result guarantees.

For all coverage criteria, we have examined 50 test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [51].

Construct Validity: In our study, we primarily measure fault finding over seeded faults, rather than real faults encountered during development. However, Andrews et al. showed that seeded faults lead to similar conclusions to those obtained using real faults [55] for the purpose of measuring test effectiveness. We have assumed these conclusions hold true in our domain/language for several case examples. We have, however, made use of two systems containing real faults in order to widen our pool of observations.

Our generation of mutants was randomized to avoid bias in mutant selection. A large pool of mutants was used to avoid generating a set of mutants particularly skewed toward or against a coverage criteria. In our experience, mutants sets greater than 100 result in very similar fault finding; we generated 250 to further increase our confidence no bias was introduced. In addition, we have also used one case example which has an associated set of real faults, which yields results comparable to those found when using seeded faults.

We measure the cost of test suites in terms of the number of steps. Other measurements exist, e.g., the time required to generate and/or execute tests [56]. We chose size as a metric that favors directed test generation. Thus, conclusions concerning the inefficacy of directed test generation are reasonable.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. (Notably, we have avoided statistical inference across case examples.)

7 CONCLUSION

Our results indicate that the use of structural coverage as a supplement to an existing testing method—such as random testing—may result in more effective test suites than random testing alone. However, the results for the use of coverage

criteria as a *target* for directed, automatic test case generation are mixed. For three of the systems, automatic test case generation yielded effective tests. However, for the remaining systems, randomly generated tests often yielded similar—or more effective—fault-finding results. These results lead us to conclude that, while coverage criteria are potentially useful as test adequacy criteria, the use of coverage-directed test generation is more questionable as a means of creating tests within the domain of avionics systems. If simple random test generation can yield equivalently sized—but more effective test suites—for more traditional coverage criteria such as decision, condition or MC/DC coverage, then more research must be conducted before automated test generation can be recommended.

We do not wish to condemn a particular test generation method or recommend another. Instead, we want to shine a light on the risks of relying on structural coverage criteria as an assurance of effective testing. Given the important role of structural coverage criteria in the verification and validation of safety-critical avionics systems, we find these results quite troublesome. We believe that structural coverage criteria are, for the domain explored, potentially unreliable, and thus, unsuitable, as a target for determining the adequacy of automated test suite generation. Our observations indicate a need for methods of determining test adequacy that (1) provide a reliable measure of test quality and (2) are better suited as targets for automated techniques. At a minimum, such coverage criteria must, when satisfied, indicate that our test suites are better than simple random test suites of equal size. Such criteria must account for all of the factors influencing testing, including the program structure, the test oracle used, the nature of the state space of the system under test, and the method of test generation. Towards this goal, the OMC/DC criterion is an improvement in this regard, but we believe there is still much work to be done.

Until the challenges of determining the efficacy of generated test suites are overcome, we urge caution when automatically generating test suites: code coverage does not guarantee test quality. While automated test generation is an alluring possibility, savings of time and cost may not be worth the trade-off in the safety of the released software.

REFERENCES

- [1] H. Zhu and P. Hall, “Test data adequacy measurement,” *Software Engineering Journal*, vol. 8, no. 1, pp. 21–29, 1993.
- [2] RTCA, *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
- [3] S. Rayadurgam and M. Heimdahl, “Coverage based test-case generation using model checkers,” in *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pp. 83–91, IEEE Computer Society, April 2001.
- [4] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for C,” 2005.
- [5] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” *PLDI05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005.
- [6] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey on automated software test case generation,” *Journal of Systems and Software*, vol. 86, pp. 1978–2001, August 2013.
- [7] M. Staats, G. Gay, M. W. Whalen, and M. P. Heimdahl, “On the danger of coverage directed test case generation,” in *15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, April 2012.
- [8] M. Heimdahl, G. Devaraj, and R. Weber, “Specification test coverage adequacy criteria = specification test generation inadequacy criteria?,” in *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*, (Tampa, Florida), March 2004.
- [9] M. P. Heimdahl and G. Devaraj, “Test-suite reduction for model based tests: Effects on test quality and implications for testing,” in *Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering (ASE)*, (Linz, Austria), September 2004.
- [10] M. Staats, M. Whalen, and M. Heimdahl, “Better testing through oracle selection (nier track),” in *Proceedings of the 33rd Int'l Conf. on Software Engineering*, pp. 892–895, 2011.
- [11] M. Staats, G. Gay, and M. Heimdahl, “Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing,” in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, 2012.
- [12] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats, “Observable modified condition/decision coverage,” in *Proceedings of the 2013 Int'l Conf. on Software Engineering*, ACM, May 2013.
- [13] G. Hagen, *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [14] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, no. 99, p. 1, 2010.
- [15] J. J. Chilenski and S. P. Miller, “Applicability of Modified Condition/Decision Coverage to Software Testing,” *Software Engineering Journal*, pp. 193–200, September 1994.
- [16] N. Juristo, A. Moreno, and S. Vegas, “Reviewing 25 years of testing technique experiments,” *Empirical Software Engineering*, vol. 9, no. 1, pp. 7–44, 2004.
- [17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria,” 1994.
- [18] P. Frankl and S. N. Weiss, “An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria,” in *Proc. of the Symposium on Testing, Analysis, and Verification*, 1991.
- [19] A. Namin and J. Andrews, “The influence of size and coverage on test suite effectiveness,” 2009.
- [20] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 435–445, ACM, 2014.
- [21] E. Weyuker and B. Jeng, “Analyzing partition testing strategies,” *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 703–711, 1991.
- [22] T. Chen and Y. Yu, “On the expected number of failures detected by subdomain testing and random testing,” *IEEE Transactions on Software Engineering*, vol. 22, no. 2, 1996.
- [23] D. Hamlet and R. Taylor, “Partition testing does not inspire confidence,” *Software Engineering, IEEE Transactions on*, vol. 16, pp. 1402–1411, Dec 1990.
- [24] W. Gutjahr, “Partition testing vs. random testing: The influence of uncertainty,” *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, 1999.
- [25] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, “Formal analysis of the effectiveness and predictability of random testing,” in *ISSTA*, pp. 219–230, 2010.
- [26] A. Arcuri and L. C. Briand, “Adaptive random testing: An illusion of effectiveness?,” in *ISSTA*, 2011.
- [27] A. Gargantini and C. Heitmeyer, “Using model checking to generate tests from requirements specifications,” *Software Engineering Notes*, vol. 24, pp. 146–162, November 1999.
- [28] R. Gopinath, C. Jensen, and A. Groce, “Code coverage for suite evaluation by developers,” in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 72–82, ACM, 2014.
- [29] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *ICSE*, pp. 416–426, 2007.
- [30] Y. Yu and M. Lau, “A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions,” *Journal of Systems and Software*, vol. 79, no. 5, pp. 577–590, 2006.
- [31] S. Kandl and R. Kirner, “Error detection rate of MC/DC for a case study from the automotive domain,” *Software Technologies for Embedded and Ubiquitous Systems*, pp. 131–142, 2011.

- [32] A. Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the hete-2 satellite software," in *Proc. of the Digital Aviation Systems Conf. (DASC)*, (Philadelphia, USA), October 2000.
- [33] A. Gacek, "JKind - a Java implementation of the KIND model checker," <https://github.com/agacek>, 2015.
- [34] G. Fraser, F. Wotawa, and P. Ammann, "Issues in using model checkers for test case generation," *Journal of Systems and Software*, vol. 82, no. 9, pp. 1403–1418, 2009.
- [35] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*, Internet Society, 2008.
- [36] M. Veines, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with spec explorer," in *Formal Methods and Testing* (R. M. Hierons, J. P. Bowen, and M. Harman, eds.), vol. 4949 of *Lecture Notes in Computer Science*, pp. 39–76, Springer, 2008.
- [37] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, pp. 11:1–11:29, Feb. 2011.
- [38] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105–156, 2004.
- [39] "Reactive systems inc. Reactis Product Description." <http://www.reactive-systems.com/index.msp>.
- [40] RTCA/DO-178C, "Software considerations in airborne systems and equipment certification."
- [41] A. Murugesan, S. Rayadurgam, and M. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.
- [42] "Mathworks Inc. Simulink." <http://www.mathworks.com/products/simulink>, 2015.
- [43] "MathWorks Inc. Stateflow." <http://www.mathworks.com/stateflow>, 2015.
- [44] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [45] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," 2008.
- [46] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *Proc of the 27th Int'l Conf on Software Engineering (ICSE)* pp. 402–411, 2005.
- [47] C. Van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 814–819, 2002.
- [48] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.
- [49] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, IEEE Press, 2012.
- [50] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
- [51] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proc. of the 30th Int'l Conf. on Software engineering*, pp. 161–170, ACM, 2008.
- [52] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *ISSRE'13: Proceedings of the 24th IEEE Int'l Symposium on Software Reliability Engineering*, IEEE Press, Nov. 2013.
- [53] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *ISSRE*, pp. 311–320, 2012.
- [54] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation, STOV '10*, (New York, NY, USA), pp. 1–4, ACM, 2010.
- [55] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 608 –624, aug. 2006.
- [56] G. Devaraj, M. Heimdahl, and D. Liang, "Coverage-directed test generation with model checkers: Challenges and opportunities," *Computer Software and Applications Conf. Annual Int'l*, vol. 1, pp. 455–462, 2005.



Gregory Gay is an Assistant Professor of Computer Science & Engineering at the University of South Carolina. His research interests include automated testing and analysis—with an emphasis on test oracle construction—and search-based software engineering. Greg received his Ph.D. from the University of Minnesota, working with the Critical Systems research group, and an M.S. from West Virginia University.



Matt Staats has worked as a research associate at the Software Verification and Validation lab at the University of Luxembourg and at the Korean Advanced Institute of Science and Technology in Daejeon, South Korea. He received his Ph.D. from the University of Minnesota-Twin Cities. Matt Staats's research interests are realistic automated software testing and empirical software engineering. He is currently employed by Google, Inc.



Michael Whalen is a Program Director at the University of Minnesota Software Engineering Center. Dr. Whalen is interested in formal analysis, language translation, testing, and requirements engineering. He has developed simulation, translation, testing, and formal analysis tools for Model-Based Development languages including Simulink, Stateflow, SCADE, and *RSML^e*, and has published extensively on these topics. He has led successful formal verification projects on large industrial avionics models, including displays (Rockwell-Collins ADGS-2100 Window Manager), redundancy management and control allocation (AFRL CerTA FCS program) and autoland (AFRL CerTA CPD program). He has recently been researching tools and techniques for scalable compositional analysis of system architectures.



Mats P.E. Heimdahl is a Full Professor of Computer Science and Engineering at the University of Minnesota, the Director of the University of Minnesota Software Engineering Center (UM-SEC), and the Director of Graduate Studies for the Master of Science in Software Engineering program. He earned an M.S. in Computer Science and Engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden and a Ph.D. in Information and Computer Science from the University of California at Irvine.

His research interests are in software engineering, safety critical systems, software safety, testing, requirements engineering, formal specification languages, and automated analysis of specifications.

He is the recipient of the NSF CAREER award, a McKnight Land-Grant Professorship, the McKnight Presidential Fellow award, and the awards for Outstanding Contributions to Post-Baccalaureate, Graduate, and Professional Education at the University of Minnesota.

Efficient Observability-based Test Generation by Dynamic Symbolic Execution

Dongjiang You¹, Sanjai Rayadurgam¹, Michael Whalen¹, Mats P.E. Heimdahl¹, Gregory Gay²

¹Department of Computer Science and Engineering, University of Minnesota, USA

²Department of Computer Science and Engineering, University of South Carolina, USA

Email: [djyou, rsanjai, whalen, heimdahl]@cs.umn.edu, greg@greggay.com

Abstract—Structural coverage metrics have been widely used to measure test suite adequacy as well as to generate test cases. In previous investigations, we have found that the fault-finding effectiveness of tests satisfying structural coverage criteria is highly dependent on program syntax – even if the faulty code is exercised, its effect may not be observable at the output. To address these problems, observability-based coverage metrics have been defined. Specifically, Observable MC/DC (OMC/DC) is a criterion that appears to be both more effective at detecting faults and more robust to program restructuring than MC/DC. Traditional counterexample-based test generation for OMC/DC, however, can be infeasible on large systems. In this study, we propose an incremental test generation approach that combines the notion of observability with dynamic symbolic execution. We evaluated the efficiency and effectiveness of our approach using seven systems from the avionics and medical device domains. Our results show that the incremental approach requires much lower generation time, while achieving even higher fault finding effectiveness compared with regular OMC/DC generation.

I. INTRODUCTION

Test adequacy criteria defined over the structure of a program, such as branch coverage and modified condition/decision coverage (MC/DC) serve as useful benchmarks for assessing the thoroughness of testing software and for meeting regulatory requirements. Of particular interest to us are criteria used to evaluate testing effort for safety-critical systems, such as MC/DC [1], which is mandated by the U.S. standard DO-178C [2] for testing the most critical avionics software. In previous investigations, we have found that the effectiveness of structural coverage criteria is highly dependent on the structure of the program under test [3]. Simple syntactic transformations, such as inlining of variables, have a dramatic effect on the fault-finding efficacy of test suites designed to satisfy the MC/DC criterion. This effect is not surprising – structural criteria, as the name suggests, require that certain code structures, such as branches or Boolean expressions, be exercised to a certain level of thoroughness. However, the degree to which fault-finding was impacted by simple syntactic changes to the program is bothersome, especially given that such criteria are used to assess the testing effort for safety-critical software. These structural criteria specify that various structural parts of a program must be exercised, they do not, however, mandate that the effect of exercising that part must manifest itself at a subsequent *observable*

point in the program; a frequent problem is that the effect of a corrupted variable gets masked out through its use in subsequent operations.

To address this problem, *observability* has been proposed as a desirable attribute of testing in both hardware [4] and software domains [5]. Specifically, Whalen et al. proposed OMC/DC – a combination of traditional MC/DC with the notion of observability [5]. In practical terms, OMC/DC adds an additional path constraint to the coverage obligation. This constraint concretizes the idea that the *effect* of satisfying the MC/DC obligation must propagate to an observable output. The OMC/DC criterion defines propagation (optimistically) by specifying what constitutes *masking* and mandating a non-masking path from the point of exercising the code structure to some monitored output variable. This greatly increases the likelihood of faults triggered during testing to be observed as failures. It was demonstrated that OMC/DC significantly outperforms MC/DC with respect to fault finding and robustness to syntactic transformations on industrial models from the avionics and medical device domains [5], [6].

While OMC/DC offers a significant improvement over MC/DC, it makes generating tests to satisfy the criterion more difficult; in particular, to give a corrupted state the opportunity to propagate to an output in a stateful system (common in the critical systems domain), a test case may need to execute for a large number of test steps. In our work, we have generated tests by formulating test obligations – properties that must be satisfied by the paths that are executed – and using model checkers to find the concrete tests by asserting that such obligations cannot be satisfied. The counterexamples generated are test cases – paths that satisfy those obligations. The path constraint added by OMC/DC for effect propagation makes this search harder and often prohibitively expensive even for relatively small systems.

In this paper, we propose an incremental test generation strategy that addresses this scalability problem using *dynamic symbolic execution*, in which test inputs are generated incrementally by combining concrete and symbolic executions of the program under test [7], [8]. Starting with some concrete values for inputs to obtain a concrete execution path, path constraints are then generated for that execution path by treating (some of the) variables as symbolic. These constraints are then systematically modified and solved to force the program to take different (but feasible) execution paths. There

This work has been partially supported by NSF grant CNS-1035715.

has been a large volume of research on dynamic symbolic execution that demonstrates encouraging improvement over classic symbolic execution [9], [10], [11], [12]. Efficiency is gained by modifying the path constraints along a known concrete path, which localizes the search for newer feasible paths. This reduces the high computational cost associated with symbolic search and, in practice, leads to better coverage of the program paths and higher likelihood of revealing faults.

In the test generation approach presented in this paper, we adopt the dynamic symbolic execution idea in a novel approach to generate tests that, first, satisfy a condition based coverage criterion to exercise a particular part of the code (MC/DC) and, second, satisfy a dataflow criterion propagating the possibly corrupted state to a point where it is used by an observable output. To this effect, we employ a tagging semantics similar to the one defined by Whalen et al. [5] where a tag is assigned to each condition and the propagation of tags to outputs approximates observability. Instead of invoking a model checker to generate a complete OMC/DC test, which can be prohibitively expensive due to the test case length, the incremental test generation starts with concrete test inputs that satisfy an obligation and then invokes the model checker at each test step repeatedly to solve path conditions in an attempt to propagate tags through non-masking paths towards outputs. A test case satisfying an obligation incrementally grows and eventually tags associated with the condition of interest (the condition the MC/DC obligation exercised) reach output variables (to form an OMC/DC test).

While this approach introduces some incompleteness – a tag associated with a condition may not be able to propagate to outputs on the particular concrete path being extended – this approach worked remarkably well in our experiments. Our results indicate that the incremental approach consumes significantly less time than regular test generation, while achieving universally better fault finding effectiveness.

II. BACKGROUND

Coverage metrics can often be used to measure test suite adequacy as well as to generate test cases. Modified condition/decision coverage (MC/DC) is a structural coverage metric used in some safety-critical systems domains. In order to satisfy MC/DC, (1) each condition in a decision has to take on every possible outcome, and (2) each condition in a decision has to be shown to independently affect the outcome of the decision [1]. For example, in order for a to independently affect the outcome of $(a \text{ and } b)$, b has to be *true*, otherwise, $(a \text{ and } b)$ would evaluate to *false* no matter what value a has. In this paper, we use the masking form of MC/DC [13].

It has been shown that structural coverage metrics are sensitive to program structures [3]. Simple syntactic transformations can have a dramatic effect on the generated test suite as well as its fault finding effectiveness. Furthermore, even if the faulty code is exercised, the corrupted program state may not be propagated to observable outputs. Observability-based coverage metrics have been proposed to address these problems [14], [5]. Observable modified condition/decision

TABLE I
ENHANCED TAGGING SEMANTICS

$E ::=$	$Val Id E \text{ op } E \text{not } E $
	$E ? E : E \text{tag}(E, T) (Val, TS) \text{addTags}(E, TS)$
$Context ::=$	$\square Context \text{ op } E E \text{ op } Context \text{not } Context $
	$Context ? E : E \text{addTags}(Context, TS) $
	$\langle K : Context, \mathcal{E} : Env, \dots \rangle$
lit	$n \Rightarrow (n, \emptyset)$
var	$\langle \mathcal{E} : \sigma \rangle[x] \Rightarrow \langle \mathcal{E} : \sigma \rangle[(\sigma x)] \text{ if } x \in \text{dom}(\sigma)$
op	$(n_0, l_0) \oplus (n_1, l_1) \Rightarrow (n_0 \oplus n_1, l_0 \cup l_1)$
and₁	$(tt, l_0) \text{ and } (tt, l_1) \Rightarrow (tt, l_0 \cup l_1)$
and₂	$(tt, l_0) \text{ and } (ff, l_1) \Rightarrow (ff, l_1)$
and₃	$(ff, l_0) \text{ and } _ \Rightarrow (ff, l_0)$
ite₁	$(tt, l_0) ? e_t : e_e \Rightarrow \text{addTags}(e_t, (e_t =_v e_e) ? \emptyset : l_0)$
ite₂	$(ff, l_0) ? e_t : e_e \Rightarrow \text{addTags}(e_e, (e_t =_v e_e) ? \emptyset : l_0)$
tag	$\text{tag}(t, (v, l)) \Rightarrow (v, l \cup \{(t, v)\})$
addt	$\text{addTags}((v, l_0), l_1) \Rightarrow (v, l_0 \cup l_1)$

coverage (OMC/DC) is an improved criterion over MC/DC, in which a tag is associated with each condition and the propagation of tags is used to approximate observability.

Although OMC/DC outperforms MC/DC in terms of fault finding effectiveness and sensitivity to program structure, it suffers from scalability issues. This is not surprising – with the notion of observability, the generated test case is much longer than the ones generated to satisfy MC/DC, and the time cost of generation increases exponentially. Furthermore, the original tagging semantics has optimistic inaccuracy. In the following code fragment, the condition c is reported to be observable, but c cannot affect the outcome of this code fragment.

```
if (c) then out := 0 else out := 0 ;
```

In this paper, we extended the tagging semantics defined by Whalen et al. by removing the optimistic inaccuracy in *if-then-else* expressions. The extended tagging semantics is shown in Table I. The rules with gray backgrounds are enhancements of the original definitions, where $=_v$ represents value equality of two expressions.

A. The Dataflow Language Lustre

Dataflow languages, which assign values to a set of equations in response to periodic inputs, are popular in model-based development using tools such as Simulink and SCADE. The dataflow language we are using – Lustre – is a synchronous dataflow language for programming reactive systems [15]. It is typically used as an intermediate representation between behavioral models and source code. Lustre programs can be automatically translated from models in notations such as Simulink, and can be automatically translated further to implementations in languages such as C/C++ and also as input models to verification tools such as model checkers.

A Lustre program consists of assignments to *combinatorial* and *delay* variables, and evaluates in cycles (i.e., computational *steps*). Combinatorial variables are used to update program state within one computational step. Delay variables are used to store program state ($\frac{1}{z}$ blocks in Simulink), such that an expression can refer to a variable's value from the *previous* step (i.e., values of delay variables).

During a cycle, variables are assigned values based on their defining equations – a combinatorial computation involving values at the current step for combinatorial variables and values from the previous step for delay variables. Within a computational step, Lustre does not impose a sequential order on evaluation of equations, but a partial order based on data dependencies. An equation can be evaluated as long as values of all variables it uses have been computed.

Suppose we have the following code fragment, in which $in1$, $in2$, and $in3$ are input variables, $v1$, $v2$, $v3$, and $v4$ are internal state variables, and out is an output variable. All variables are of type Boolean. Variable $v1$ at the initial step will have the value *false* followed by (shown as an arrow), at each subsequent step, $in1$'s value from the previous step.

```
v1 = (false -> (pre in1));
v2 = (in2 and v1);
v3 = (false -> (pre v2));
v4 = (if in3 then v2 else v3);
out = (false -> (pre v4));
```

When testing such reactive systems, a test case typically contains multiple steps, each of which specifies values for input variables. Internal and output variables are then computed by the program as described above. Values stored in delay variables (i.e., $in1$, $v2$, and $v4$) will be used by other variables (i.e., $v1$, $v3$, and out , respectively) in the next computational step. Table II shows an example of a test case of 4 steps together with values of internal and output variables.

TABLE II
LUSTRE PROGRAM EVALUATION

Step	Input Vars. ($in1$, $in2$, $in3$)	Internal Vars. ($v1$, $v2$, $v3$, $v4$)	Output Vars. (out)
1	(T, F, F)	(F, F, F, F)	(F)
2	(F, T, F)	(T, T, F, F)	(F)
3	(F, F, F)	(F, F, T, T)	(F)
4	(F, F, F)	(F, F, F, F)	(T)

B. Dynamic Symbolic Execution

In the regular counterexample-based test generation, test obligations are first formulated and instrumented in the original program. Test obligations represent path constraints for the program to take certain paths (e.g., satisfying a coverage criterion). Trap properties are simply negations of test obligations that must be fulfilled. Asserting these properties on the program to a model checker leads to the generation of counterexamples, which can be seen as test cases satisfying the obligations. For the purpose of generating OMC/DC tests, the path constraints describe *non-masking paths* starting from the initial program state to a state exercising the condition of interest in a specific way and then to a program state where the effect is observable at some output variable. This kind of generation can be expensive because the model checker has to search through a complete non-masking path which may involve many computational steps.

Dynamic symbolic execution provides improved scalability over classic symbolic execution by combining concrete and symbolic executions. By using concrete values, dynamic symbolic execution is able to simplify constraints, which helps it

generate test inputs for execution paths that classic symbolic execution is infeasible to analyze.

The notion of observability can be captured naturally using dynamic symbolic execution. That is, in order to propagate a tag tag_c from a condition c to an output out , which traverses a sequence of variables $(c, v_1, v_2, \dots, v_n, out)$, we can propagate tag_c to any intermediate variable v_i and from v_i further propagate tag_c along the path. This process terminates when tag_c reaches out or there are no feasible paths.

Therefore, our incremental approach invokes the model checker at each computational step, and forms and solves path conditions *locally* based on the existing program state resulting from executing a concrete input. Specifically, at the initial step, our approach is similar to traditional counterexample-based test generation. Once we have a concrete input, it is immediately executed on the current program state and then concatenated with the existing test. If tags – which are associated with conditions and used to approximate observability – are propagated to outputs, then we have a complete test; otherwise, we record the set of variables that these tags can propagate to. In the next step, we start generating tests based on the program state and recorded set of variables. To enable test generation from a specific program state, the original trap property is combined with additional constraints that imply a desired initial state, which is essentially the program state at the end of the execution of the test case constructed thus far.

III. INCREMENTAL TEST GENERATION

We define *observation points* as the set of variables where (internal) program state can be observed, i.e., an effect propagated to any of the observation points is *observable*. In an observability-based coverage criterion, observation points are usually the set of output variables. We use \mathcal{O} to represent *output observation points*. In our incremental test generation approach, we also define *delayed observation points* (represented by \mathcal{D}) as the set of delay variables, i.e., an effect propagated to delay variables is stored and may be further propagated. Therefore, when a tag is propagated to some output observation point, we have a complete test. Alternatively, when a tag is propagated to some delayed observation point, it will be further propagated in the next step.

A. Non-Masking Path Conditions

A condition/variable is *observable* if it is not masked along some path as described in the tagging semantics in Table I. We call the path a *non-masking path* and the constraints for the program to take the path as *non-masking path conditions*. An *immediate* non-masking path is a dataflow path from a variable to an observation point that is entirely within one computational step (i.e., no delays) where the value of the variable is not masked. Immediate non-masking paths can be defined inductively by examining define-use relationships among variables. In our incremental test generation approach, all non-masking paths are immediate. Therefore, we use the term *non-masking paths* in the remaining of the paper.

Suppose y is one of the variables that uses x , then tags associated with x can be propagated if x is not masked in the definition of y (i.e., x affects y) and y is observable. We track these relationships by instrumenting the original program with additional path condition variables. Specifically, we use x_AFFECT_y to indicate the path condition that x is not masked in the definition of y (i.e., tags with x can be propagated to y). We use $var_observed$ to represent the constraint of a non-masking path that the variable var can be propagated to (one or more) observation points. We also create additional input variables $var_observable$ for all delay variables to represent whether a delay variable var is one of the observation points. Output variables are always observable.

Suppose we have the same code fragment from Section II, we can then generate the following non-masking path conditions.

```

in2_AFFECT_v2 = v1;
v1_AFFECT_v2 = in2;

in3_AFFECT_v4 = (v2 <> v3);
v2_AFFECT_v4 = in3;
v3_AFFECT_v4 = (not in3);

in1_observed = in1_observable;
in2_observed = (in2_AFFECT_v2 and v2_observed);
in3_observed = (in3_AFFECT_v4 and v4_observed);
v1_observed = (v1_AFFECT_v2 and v2_observed);
v2_observed = ((v2_AFFECT_v4 and v4_observed)
               or v2_observable);
v3_observed = (v3_AFFECT_v4 and v4_observed);
v4_observed = v4_observable;
out_observed = true;

```

In Whalen et al.'s work, tags associated with the condition $in3$ can always be propagated to the variable $v4$, which creates optimistic inaccuracy: when $v2 = v3$, $in3$ does not play any role in determining the value of $v4$. In the present work we remove this inaccuracy by strengthening the path condition for tag propagation in such cases. In the above example, in order to propagate tags from $in3$ to $v4$, we also require the inequality of $v2$ and $v3$. More precisely, we require the value inequality of expressions from two branches.

The variable $v2$ is used in two equations and therefore has two non-masking paths to observability: one through $v4$ and the other through itself. Each of the other variables is used once and thus has one non-masking path.

B. Test Generation

We then define *propagation* as a structure that contains the following fields:

- 1) *tag* labels a condition and its Boolean value.
- 2) *state* represents the current program state.
- 3) *locations* represents the set of variables that *tag* has propagated to. Initially, for example, tag_c is at the location of condition c .
- 4) *visited* represents the set of visited observation points.
- 5) *test* represents an incrementally generated test case.

Algorithm 1 shows the main algorithm of our incremental test generation approach. The MAIN procedure takes a condition c and returns a test case that can propagate the effect

of c to some output observation points, or *UNSAT*. During each iteration of the *while* loop, the algorithm first attempts to generate a test that can propagate tag_c to \mathcal{O} (line 7 – 10). If *SAT*, the newly generated test is concatenated with existing ones and we have a complete test. Otherwise, the algorithm attempts to generate a test that can propagate tag_c to \mathcal{D} (line 11 – 14). If *UNSAT*, there is no feasible path that can propagate tag_c further and thus *UNSAT* is returned. Otherwise, tag_c is propagated to delayed observation points and will be further propagated in the next iteration.

Algorithm 1 Incremental Test Generation

```

1:  $\mathcal{O} \leftarrow$  output observation points
2:  $\mathcal{D} \leftarrow$  delayed observation points
3:
4: procedure MAIN( $c$ )
5:    $p \leftarrow (tag_c, \emptyset, \{c\}, \emptyset, \emptyset)$ 
6:   while true do
7:      $test \leftarrow$  GENERATE( $p, \mathcal{O}$ )
8:     if  $test = SAT$  then
9:        $p.test \leftarrow p.test ++ test$  ▷ Concatenate
10:      return  $p.test$ 
11:      $test \leftarrow$  GENERATE( $p, \mathcal{D}$ )
12:     if  $test = UNSAT$  then
13:       return UNSAT
14:      $p \leftarrow$  PROPAGATE( $p, test$ )

```

Algorithm 2 shows the sub-procedure GENERATE. The GENERATE procedure takes a propagation structure p and a given set of variables as observation points op , and returns a test that can propagate $p.tag$ to any variable(s) in op , or *UNSAT*. The *path constraint* is first constructed by a disjunction of path conditions from the set of variables in $p.locations$ (line 2 – 5). The disjunction ensures that the tag can be propagated when *any* of the variables can be observable. Then the constraint on observation points is added (line 6 – 10). For each delay variable, if it is in the given set of observation points and not visited, it will be a candidate observation point that $p.tag$ can propagate to. Finally, the program state $p.state$ is added such that the generation starts from a concrete state (line 11).

Algorithm 2 Test Generation

```

1: procedure GENERATE( $p, op$ )
2:    $pc \leftarrow false$  ▷ Path constraint
3:   for each  $var$  in  $p.locations$  do
4:      $pc \leftarrow pc \vee var\_observed$ 
5:    $pc \leftarrow (pc)$ 
6:   for each  $var$  in  $\mathcal{D}$  do
7:     if  $var \in (op \setminus p.visited)$  then
8:        $pc \leftarrow pc \wedge (var\_observable)$ 
9:     else
10:       $pc \leftarrow pc \wedge (not var\_observable)$ 
11:    $pc \leftarrow pc \wedge p.state$ 
12:   return SOLVE( $pc$ )

```

Essentially, in an observability-based coverage for dataflow languages, tags are always located at delay variables, unless they are propagated to output variables. In the incremental test generation approach, however, it is possible that a tag is repeatedly propagated to the same set of delay variables, forming a cyclic propagation. Cyclic propagation is a result

of concretizing program state and generating new tests locally. From a concrete program state, test generation (and thus tag propagation) is deterministic. The locally optimal path found by the model checker may be in a subspace that will repeatedly propagate the tag inside the subspace – the particular path chosen need not necessarily be the best candidate for eventually reaching an output – making test generation not terminate.

In our present approach, we favor *incrementality*. To account for the problem of cyclic propagation, we include a simple heuristic (i.e., recording all visited variables in $p.visited$ and not propagating $p.tag$ to them) that avoids repeatedly visiting the same delayed observation point. This heuristic forces the test case to take a different path – which intuitively would increase fault finding effectiveness – and guarantees that the generation process will terminate. Otherwise, as in the traditional generation, a model checker typically generates the shortest possible path that satisfies an obligation. This may not be desirable from a testing perspective: for it may lead to fewer program states being visited which can negatively impact fault finding [6]. Given all the benefits, however, the heuristic may potentially miss some feasible test cases that necessarily require passing through cyclic propagation to reach some observable output. These test cases could have been found (if feasible) by attempting to generate a complete test case to satisfy the OMC/DC obligation.

Algorithm 3 shows the sub-procedure PROPAGATE. The PROPAGATE procedure takes a propagation structure p and a newly generated $test$, and returns the updated p .

Algorithm 3 Tag Propagation

```

1: procedure PROPAGATE( $p, test$ )
2:    $p.state \leftarrow EXECUTE(p.state, test)$ 
3:    $locations \leftarrow \emptyset$ 
4:   while  $p.locations \neq \emptyset$  do
5:     Select and remove  $var$  from  $p.locations$ 
6:      $useSet \leftarrow DFG$  retrieval variables that use  $var$ 
7:     for each  $use$  in  $useSet$  do
8:       if  $var\_AFFECT\_use$  then
9:         if  $use \in (\mathcal{D} \setminus p.visited)$  then
10:           Add  $use$  to  $locations$ 
11:         else
12:           Add  $use$  to  $p.locations$ 
13:    $p.locations \leftarrow locations$ 
14:    $p.visited \leftarrow p.visited \cup p.locations$ 
15:    $p.test \leftarrow p.test ++ test$                                  $\triangleright$  Concatenate
16:   return  $p$ 
```

The newly generated test is first executed from the current program state to obtain an updated program state (line 2). Then $p.tag$ is propagated from current $p.locations$ to new $locations$ (line 3 – 13). Specifically, we first select and remove a location var from $p.locations$ and retrieve the set of variables $useSet$ that use var from the dataflow graph (line 5 – 6). For each variable use in $useSet$, the path condition var_AFFECT_use is checked (line 8). If var affects use , $p.tag$ is propagated from var to use . Then if use is an unvisited observation point, it is added to new $locations$; otherwise, it is added to $p.locations$ for further propagation. Eventually, new locations are recorded as visited (line 14) and the new test is concatenated with existing ones (line 15).

C. Test Generation Example

Suppose our goal is to cover the MC/DC obligation of $in2$ with *true* value (as MC/DC requires, the condition $in2$ has to evaluate to both *true* and *false* values, and be shown to independently affect the outcome of the decision) and propagate its effect not only to $v2$ (as MC/DC requires), but also to out (as OMC/DC requires). In the following illustration, the program state is not expanded in order to simplify the representation. We will only show feasible path constraints and generated tests, as well as the propagation structure before and after each iteration.

1) Iteration 1: Propagate tag_{in2_true} from $\{in2\}$.

At the first iteration, the path constraint is a conjunction of an MC/DC obligation over a single atomic condition plus a path condition representing the variable's observability at one of the observation points. Propagating tag_{in2_true} to output observation points is infeasible, so it will be propagated to $v2$ as shown in the following.

Before: $p = (tag_{in2_true}, state, \{in2\}, \emptyset, \emptyset)$	$pc = (in2 \text{ and } in2_AFFECT_v2 \text{ and } v2_observed)$
	and
	$(v2_observable \text{ and } v4_observable)$
After: $p = (tag_{in2_true}, state, \{v2\}, \{v2\},$	
	$\{(T, F, \bar{F}), (\bar{F}, T, F)\}\}$

2) Iteration 2: Propagate tag_{in2_true} from $\{v3\}$.

At the second iteration, tag_{in2_true} automatically propagates from $v2$ to $v3$, since $v3$ uses the delay variable $v2$. Propagating tag_{in2_true} to output observation points is still infeasible, so it will be propagated to $v4$ as shown in the following. Note that since $v2$ has been visited, it is shown as *not observable* in the path constraint.

Before: $p = (tag_{in2_true}, state, \{v3\}, \{v2\},$	$\{(T, F, F), (F, T, F)\}\}$
	$pc = (v3_observed)$
	and
	$(not v2_observable \text{ and } v4_observable)$
	and
	$(p.state)$
After: $p = (tag_{in2_true}, state, \{v4\}, \{v2, v4\},$	
	$\{(T, F, F), (F, T, F), (F, F, F)\}\}$

3) Iteration 3: Propagate tag_{in2_true} from $\{out\}$.

At the third iteration, tag_{in2_true} automatically propagates from $v4$ to out , since out uses the delay variable $v4$. Propagating tag_{in2_true} to output observation points is feasible, so a complete test is generated and the incremental test generation algorithm terminates. Note that since both $v2$ and $v4$ have been visited, they are shown as *not observable* in the path constraint.

Before: $p = (tag_{in2_true}, state, \{out\}, \{v2, v4\},$	$\{(T, F, F), (F, T, F), (F, F, F)\}\}$
	$pc = (out_observed)$
	and
	$(not v2_observable \text{ and } not v4_observable)$
	and
	$(p.state)$
After: $p = (tag_{in2_true}, state, \{out\}, \{v2, v4, out\},$	
	$\{(T, F, F), (F, T, F), (F, F, F), (F, F, F)\}\}$

Table III shows a summary of the generated test case as well as the locations of tag_{in2_true} at each step.

TABLE III
TEST CASE EXAMPLE

	Locations of tag_{in2_true}	in1	in2	in3
Step 1	<i>in2</i>	<i>True</i>	<i>False</i>	<i>False</i>
Step 2	<i>v2</i>	<i>False</i>	<i>True</i>	<i>False</i>
Step 3	<i>v4</i>	<i>False</i>	<i>False</i>	<i>False</i>
Step 4	<i>out</i>	<i>False</i>	<i>False</i>	<i>False</i>

IV. EVALUATION

The quality in terms of fault finding of the test suites generated to satisfy OMC/DC and MC/DC has been evaluated [5]. In this paper, we are interested in comparing the two approaches – incremental and regular test generation satisfying OMC/DC – in terms of efficiency and effectiveness. Therefore, we have the following research questions:

- 1) Is incremental test generation more efficient than regular test generation? What is the percentage of time needed to generate a test suite in the incremental approach compared with regular test generation?
- 2) What is the generated test suite size (i.e., the number of satisfied obligations) and how does it affect test suite effectiveness in the two approaches?
- 3) How well does the test suite generated by the incremental approach perform in terms of fault finding effectiveness compared with regular test generation?

A. Case Example Systems

In this study, we used four industrial systems (i.e., *DWM1*, *DWM2*, *Vertmax*, and *Latctl*) developed by Rockwell Collins Inc., two subsystems (i.e., *Alarm* and *Infusion Manager*) of a Generic Patient Controlled Analgesia (GPCA) infusion system [16], and a last system (i.e., *Docking Approach*) created as a case example at NASA.

The Rockwell Collins systems were modeled using Simulink [17]. Two of the systems, *DWM1* and *DWM2*, represent distinct portions of a Display Window Manager for a commercial display system. The other two systems, *Vertmax* and *Latctl*, represent the vertical and lateral mode logic for a Flight Guidance System.

The remaining three systems are modeled using Stateflow [18]. Two of the systems, *Alarm* and *Infusion Manager*, represent the alarm-induced behavior and the prescription management of an infusion pump device. The NASA system, *Docking Approach*, describes the behavior of a space shuttle as it docks with the International Space Station.

All the systems were translated to the Lustre programming language [15] to take advantages of existing automation. Lustre code generation from Simulink/Stateflow models is analogous to the automated code generation offered by Mathworks Simulink Coder [19]. In practice, Lustre programs will be automatically translated to C code. Therefore, results of this study would be identical, if applied to their C implementations.

Table IV shows basic information of the systems measured on the original Simulink and Stateflow models.

TABLE IV
SIMULINK AND STATEFLOW CASE EXAMPLE INFORMATION

(a) Simulink models

	Subsystems	Blocks
DWM1	3109	11,439
DWM2	128	429
Vertmax	396	1,453
Latctl	120	718

(b) Stateflow models

	States	Transitions	Variables
Alarm	78	107	60
Infusion	27	50	36
Docking	64	104	51

B. Test Suite Generation

We used the counterexample-based approach to generate test cases satisfying obligations/path conditions [20], [21], specifically, we used the JKind model checker [22] in our experiments. This approach is guaranteed to generate a test suite that achieves the maximum possible coverage of the system under test. Note that, however, in regular test generation, this is guaranteed as long as the model checker is given enough computing resource to terminate; in incremental test generation, the maximum propagation is only guaranteed at each step, which can have both advantages and disadvantages and will be discussed in detail in Section V.

To account for variance in the generation time, we generated 10 test suites for each of the two approaches (i.e., incremental and regular test generation) for each of the seven systems under test. We measured time using Linux *time* utility. Specifically, we measured *user time* in order to account for the variance of parallel computing inside the model checker in a multi-core environment.

In this study, we used the full generated test suite to perform fault finding analysis, even if there can be redundancy in the generated test cases [5], [3]. There are two reasons for this setting: (1) when we compare the generation time of a test suite, the comparison is based on the full test suite rather than a reduced test suite; (2) it is debatable whether test suite reduction will decrease fault finding effectiveness [23], [24], and the choice of reduction algorithms may also have an effect on fault finding and reduced test suite size. Performing coverage measurement and test suite reduction have a cost (e.g., at least we need to execute all tests to measure how they satisfy obligations) and the cost of measuring coverage of the full test suite would be much more than just executing the test suite on the original program. Because of the above reasons, we decided to use the full generated test suite in this study for the purpose of fault finding analysis.

C. Mutant Generation

Mutation testing has been shown to be an adequate proxy for real faults for the purpose of investigating fault finding effectiveness [25]. In our experiment, mutant generation is done by automatically introducing a single fault into the correct implementation. The mutation operators used in this study are typical and were discussed in detail in our previous paper [26]. The generated set of mutants has roughly the

same distribution of fault types across the system occurring naturally.

For each of our systems, we created 750 mutants¹ and randomly grouped them into 10 sets, each of which contains 75 mutants. We removed mutants that cause compile-time or run-time errors (e.g., divide-by-zero), but we left possibly equivalent mutants. Although checking and removing equivalent mutants can be feasible on small systems [5], the cost of checking equivalent mutants for our last three systems is prohibitive. Therefore, we did not attempt to remove equivalent mutants in our experiments in order to keep our experimental configuration consistent across different case example systems.

D. Test Oracles

For the purpose of this study, we used output-only expected value oracles – the ones people would most likely use in practice [27]. When using an output-only expected value test oracle, for each test input, concrete values are specified that the system is expected to produce for output variables.

For each case example, we ran the generated test suites against each mutant and the original version of the system (i.e., the correct version). For each test suite, we recorded the value of all output variables at every test step of the execution of every test case using an in-house Lustre simulator.

To determine the fault finding effectiveness of the generated test suites, we simply compared output values produced by a faulty version against expected output values produced by the original version. The fault finding effectiveness of a test suite is computed as the percentage of mutants killed.

Although we generated 10 test suites for each test generation approach to account for the variance in generation time, for the purpose of evaluating fault finding effectiveness, we used one of them, since both the incremental and regular test generation approaches are deterministic in producing test cases.

Due to proprietary reasons, experiments on the first four systems were performed on an encrypted laptop with an Intel quad-core processor @ 2.4 GHz, 4 GB memory, and Ubuntu Linux. Experiments on the last three systems were performed on a server with four AMD eight-core processors @ 3.0 GHz, 192 GB memory, and Ubuntu Linux.

V. RESULTS & DISCUSSION

Recall that in Section IV we outlined three research questions related to the possible performance gains with the incremental test generation approach, the nature of the generated test suites (i.e., the number of tests generated), and the fault finding effectiveness of tests generated incrementally. Below we will present our experimental results and discuss the three questions in order.

A. RQ1: Test Generation Time

We would first like to determine whether the incremental approach performs better than regular test generation in terms of test generation time.

¹750 is slightly smaller than the maximum number of possible mutants in our smallest case example system Latctl.

TABLE V
TEST GENERATION TIME (IN SECONDS) FOR EACH SYSTEM, AVERAGE OVER 10 TEST SUITES

System	Incremental Generation	Regular Generation	Time Ratio	p-value
DWM1	141.2	107.4	131.5%	< 0.01
DWM2	30.4	39.8	76.4%	< 0.01
Vertmax	119.8	279.8	42.8%	< 0.01
Latctl	19.1	28.4	67.3%	< 0.01
Alarm	363.3	2958.9	12.3%	< 0.01
Infusion	278.5	3623.6	7.7%	< 0.01
Docking	40413.2	176226.3*	22.9%	< 0.01

Table V shows the average test generation time for each case example system and each test generation approach. The *Time Ratio* column is the percentage of time needed to incrementally generate a test suite as opposed to the regular generation of the suite. *p-value* is computed using a two-sided permutation test using *mean* as the test statistic, under the null hypothesis that test generation time from incremental and regular approaches are drawn from the same distribution. Note that regular test generation is infeasible on *Docking*. We set a timeout of 48 hours for the model checker to terminate and measured user time (as starred in Table V). It is both time and memory prohibitive to obtain the actual time from regular test generation on *Docking*. Figure 1 further illustrates the distribution of test generation time for each system.

The incremental approach is significantly more efficient than regular test generation for all systems except *DWM1*. We investigated this outlier and found that *DWM1* is purely combinatorial, i.e., there is no state maintained in the system and thus no delay operations. Therefore, all computation is within one step and finding an immediate non-masking path is generally cheap since the tag propagation is trivial, while the incremental approach adds more overhead to produce path constraints and track tag propagations. On all other systems, the incremental approach requires much less time as opposed to regular test generation with a trend that the larger/more complex the system is, the more is the improvement achieved. For smaller systems, the overhead associated with the incremental approach dominates the generation time. Additionally, our approach calls the model checker multiple times and JVM overhead can be significant on small systems with an execution time of tens of seconds.

The observed improvement matches our expectation. As we generate new test inputs at each step, we concretize the inputs we have already computed and solve a much simpler path condition to further propagate tags based on the execution results of existing test inputs.

B. RQ2: Test Suite Size

Since we used the full test suite generated from each approach, the test suite size indicates the number of satisfied obligations. Table VI shows the total number of obligations and generated test suite sizes for each approach for each case example system.

Given the enhanced tagging semantics used in our incremental approach, the test suites generated incrementally will be of equal or smaller size than test suites generated regularly,

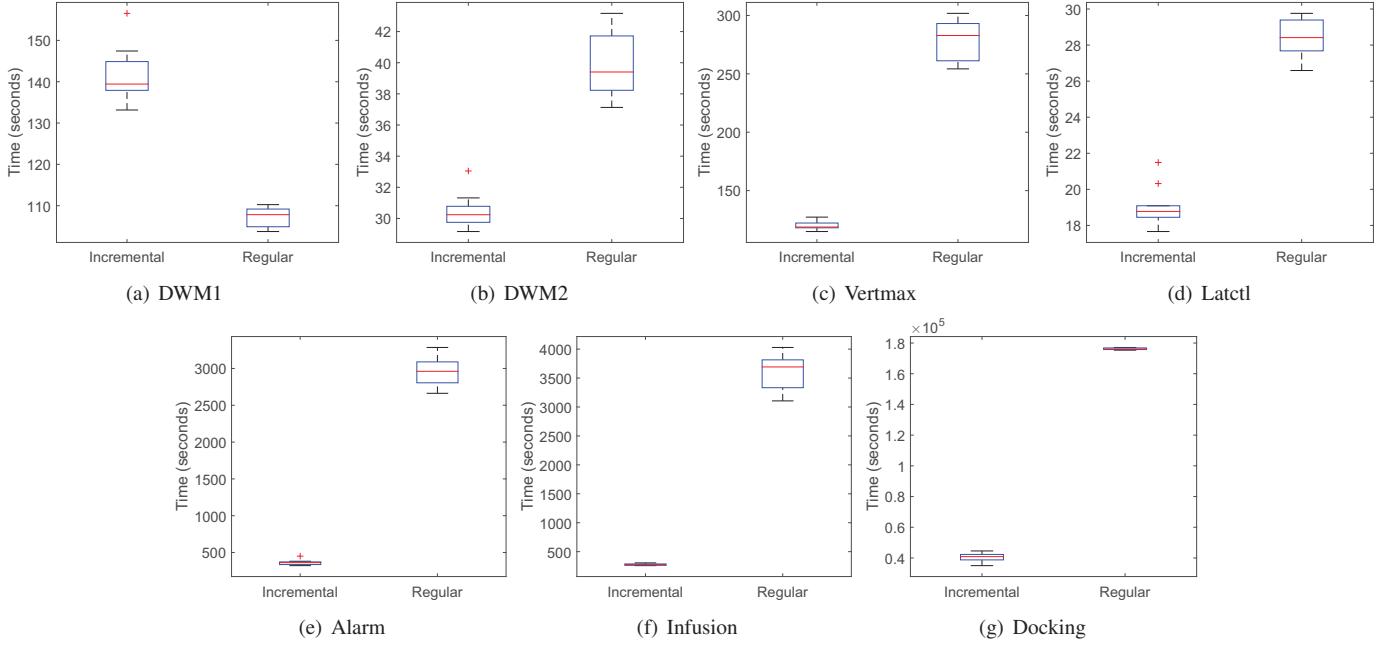


Fig. 1. Test generation time

TABLE VI
TEST SUITE SIZE FOR EACH SYSTEM

System	Total Obligations	Incremental Generation	Regular Generation
DWM1	2038	1833	2036
DWM2	530	511	511
Vertmax	1732	1705	1728
Latctl	380	371	371
Alarm	1406	991	1098
Infusion	1666	719	934
Docking	4900	961	1954

since more obligations would become unsatisfiable and tests that in fact cannot propagate tags (i.e., optimistic inaccuracy in the original OMC/DC) would be eliminated. In addition, since the incremental approach concretizes all input values at each step, it may fail to propagate tags in later steps, because future paths may have been rendered infeasible based on the previous concretization. As a result, incrementally generated test suites are expected to have fewer tests than test suites generated regularly.

Previous investigations on test suite size and fault finding effectiveness have shown that there is a moderate to high correlation between them [28], [29]. In this study, the incrementally generated test suite is, however, smaller than the test suite generated regularly. This could be a concern since we do not wish to achieve better efficiency by losing fault finding effectiveness. This leads to our third research question on comparing fault finding effectiveness between incremental and regular test generation approaches.

C. RQ3: Fault Finding Effectiveness

In the work presented in this paper we have a stronger tagging semantics than the one defined by Whalen et al. [5]. Given this more restrictive propagation requirement, it is harder (or impossible) to satisfy propagation obligations

TABLE VII
PERCENTAGE OF MUTANTS KILLED FOR EACH SYSTEM, AVERAGE OVER 10 SETS OF MUTANTS

System	Incremental Generation	Regular Generation	% Change	p-value
DWM1	93.1%	86.1%	7.0%	< 0.01
DWM2	97.5%	96.6%	0.9%	0.26
Vertmax	94.3%	89.3%	5.0%	< 0.01
Latctl	94.4%	89.9%	4.5%	0.02
Alarm	58.0%	56.5%	1.5%	0.56
Infusion	41.3%	33.7%	7.6%	< 0.01
Docking	29.8%	25.7%	4.1%	0.14

that would have been satisfiable. Consequently, we run the risk of generating fewer test cases and potentially having worse fault finding effectiveness. On the other hand, stronger path conditions can lead to better tests – with guaranteed propagation – that may increase fault finding.

Table VII shows the average percentage of mutants killed for each case example system and each test generation approach. The *% Change* column indicates the improvement in fault finding effectiveness from generating test suites incrementally compared with generating test suites regularly. *p-value* is computed using a two-sided permutation test using *mean* as the test statistic, under the null hypothesis that fault finding effectiveness from incremental and regular approaches are drawn from the same distribution. Figure 2 further illustrates the distribution of the percentage of mutants killed for each system.

Fault finding effectiveness of incrementally generated tests is for our case example systems universally better than (or equal to) regularly generated tests. Several reasons contribute to this result. Given the stronger tag propagation requirement, the optimistic inaccuracy presented in the original definition of OMC/DC is eliminated. Furthermore, JKInd is an SMT-based model checker and will always generate the shortest

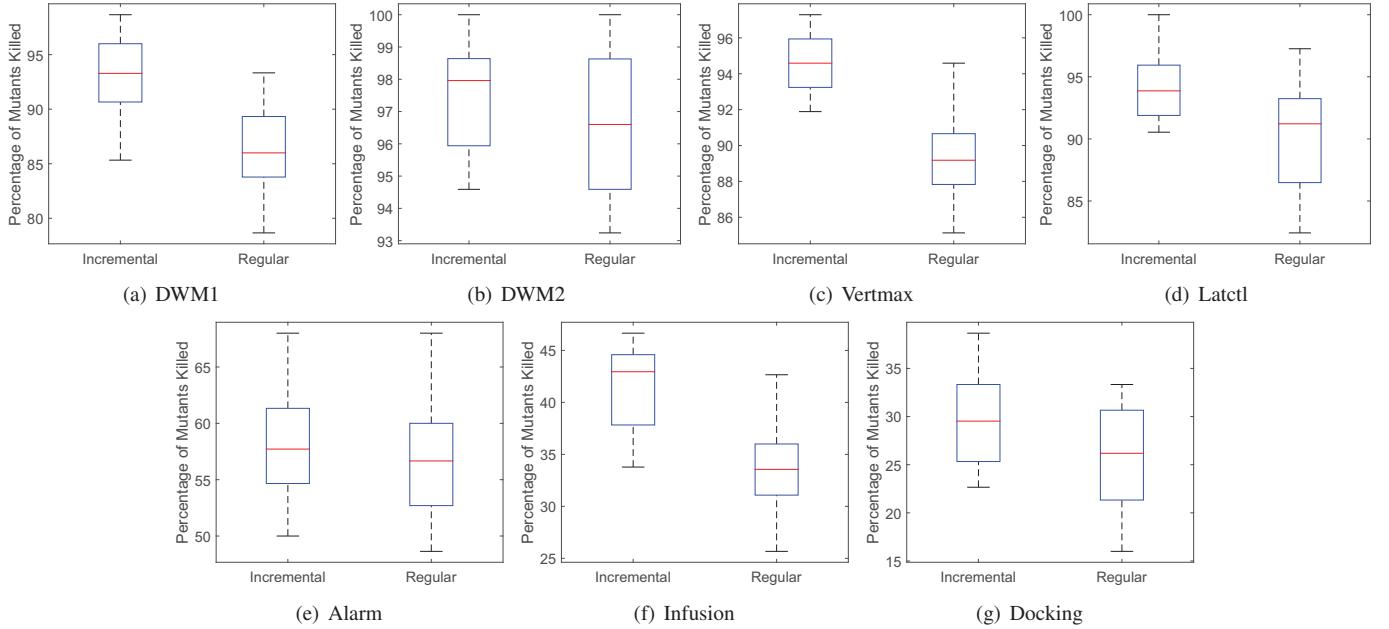


Fig. 2. Fault finding effectiveness

test cases possible to satisfy an obligation or a path condition. In our experience, such tests technically satisfy the coverage criterion, but they are (1) susceptible to the masking effect discussed earlier and (2) likely to only vary the few input variables needed to cover the obligation and leave all other input variables at the model checker’s default values (typically zero and false) [6]. Both issues are mitigated in our incremental approach. First, the incremental search for a propagation path allows the possibility to pursue a path longer than the shortest propagation path. Second, since we are restarting the search for a path in each incremental step, there is an opportunity for more variables to change values (i.e., we are no longer searching for the shortest path with the fewest variable changes). From our experimental results, we see that a combination of these two aspects of the incremental approach leads to longer and more diverse tests with better fault finding effectiveness.

On a related note, as introduced in Section IV, we did not remove equivalent mutants in this study in order to keep our experimental configuration consistent across different case example systems. As a result, fault finding effectiveness for the first four systems would be in general slightly lower than our previous results [5].

VI. THREATS TO VALIDITY

External Validity: We have used seven synchronous critical systems of different sizes from different domains (i.e., industrial avionics systems and medical device systems). We believe that these systems are representative of avionics and medical device domains. Thus, our results are generalizable to other systems in these domains.

We have used the dataflow language Lustre [15] as the implementation language. Although Lustre is a less common language than C/C++ or Java, these Lustre programs are

similar in structure to systems written in C/C++ in these domains. In practice, Lustre programs will be automatically translated to C code. Therefore, we believe that our results are applicable to systems written in more traditional imperative languages.

We have generated approximately 750 mutants for each case example system. These mutants were further divided into 10 sets. The total number of mutants and the number of mutants in each set were chosen to provide enough data points and yield a reasonable cost for evaluating fault finding effectiveness. Based on past experience, results using these numbers of mutants are representative [27]. The mutants were generated using the same tool and configuration as in previous work [5], in which OMC/DC was shown to achieve much better fault finding effectiveness than MC/DC.

Internal Validity: We have used the model checker JKInd [22] to generate test cases. The counterexample-based approach provides the shortest test cases possible to satisfy an obligation or a path condition. These automatically generated test cases may behave differently from test cases obtained by other means [6], and thus, it is possible that test cases derived by hand or random generation, may provide different results.

Construct Validity: The fault finding effectiveness is measured over mutants, i.e., seeded faults, rather than real faults encountered during development. It is possible that using real faults would lead to different results. However, Andrews et al. showed that the use of mutants leads to conclusions similar to those obtained using real faults [25].

VII. RELATED WORK

A. Observability-based Coverage Criteria

The study of observability is not new in testing of hardware logic circuits [4]. Observability-based code coverage metric (OCCOM) is a criterion in which tags are attached to internal

program states and the propagation of tags is used to predict the actual propagation of errors [14]. A variable is tagged when there is a change in the value of the variable due to an error. The observability-based coverage can be used to determine whether erroneous effects that are triggered by the inputs can be observed at the outputs.

Observable modified condition/decision coverage was recently defined as an improvement over MC/DC [5]. OMC/DC also uses a counterexample-based test generation approach and is complete in theory. That is, it is guaranteed to find non-masking paths if they exist, as long as enough time and memory are given to the model checker. In practice, however, such test generation can be infeasible on large systems and thus lose completeness. On the other hand, OMC/DC has optimistic inaccuracy in certain program structures (i.e., if-then-else statements). As a result, it may generate tests to propagate a condition to outputs, but the condition is actually not observable, leading to wasted testing effort.

Both OCCOM and OMC/DC are observability-based coverage metrics, as well as using tag propagation to approximate observability. But both approaches can be infeasible on large systems when generating tests.

In this study, we extended the tagging semantics defined by Whalen et al. [5] to generate path conditions. The differences between our work and the original OMC/DC are: (1) Our approach is a test generation approach rather than a test adequacy measurement approach defining new coverage metrics. (2) We removed optimistic inaccuracy in the original definition of OMC/DC by requiring value inequality of expressions from two branches when propagating if conditions. This added constraint is feasible due to our improvement over efficiency. (3) Our approach will explicitly terminate when there is no feasible paths, while for the regular test generation, a timeout is usually estimated and manually set in order to terminate the generation process.

A similar tagging approach, dynamic taint analysis [30], has been used in security as well as software testing and debugging. However, taint analysis is usually defined over define-use relations and masking is generally not considered. Thus, taint analysis is far more optimistically inaccurate than an observability-based approach.

B. Dynamic Symbolic Execution

The notion of dynamic symbolic execution has been used in many applications such as CUTE and jCUTE [8], [31], DART [7], KLEE [9], and Pathcrawler [10], as well as combined with fuzz testing to detect exploitable security issues [12].

Concolic testing [8] is an approach that incrementally generates test inputs by combining concrete and symbolic executions. Specifically, it starts with a random input, symbolic constraints are generated in the execution of a concrete input. These constraints are then modified and solved to force the program to take different (but feasible) execution paths. Feedback-directed random test generation [32] is another approach that also builds test inputs incrementally. That is, as soon as a new test input is created, it is executed. The

execution results are used to guide further generation. New test inputs would extend previous ones, such that it can avoid redundant or illegal test inputs and towards the execution of new program states.

We refer the reader to recent surveys [33], [34] for more details on symbolic execution systems.

Most of these approaches, however, are applied for a program to take more and different paths, as well as to cover simple coverage metrics such as branch coverage. Our incremental test generation approach, together with the extended tagging semantics for observability, explicitly propagates faulty program state to observable outputs, and thus, fault finding effectiveness can be improved dramatically.

C. Test Concatenation

Concatenating test cases have been investigated in dataflow languages [35], [36]. In a common approach to generating tests satisfying a coverage metric, each obligation is solved at the initial program state. In Hamon et al.'s approach [35], after creating a test case for an obligation, the final state of the test case serves as the initial state for the next test case. That is, the next test case can be interpreted as an extension of the previous test case. Fraser et al. [36] further investigated how test case length affects fault finding effectiveness by concatenating test cases to create test suites with different test case length, while achieving similar coverage.

VIII. CONCLUSIONS

In this paper, we described an incremental test generation approach that combines the notion of observability and dynamic symbolic execution. This approach is proposed to address optimistic inaccuracy and feasibility issues in regular counterexample-based test generation satisfying observability-based coverage criteria. Our empirical study on seven case example systems from avionics and medical device domains indicates that compared with traditional test generation approach: (1) the incremental approach is far more efficient and feasible on large systems; (2) the generated test suite size is generally smaller; (3) fault finding effectiveness of test suites generated by the incremental approach is generally better.

While our results are encouraging, there are the following areas open for exploration in future research.

(1) Reducing pessimistic inaccuracy. An obligation that is satisfiable may not have a feasible path to outputs by extending certain concrete test inputs. To address this issue, we may leave certain variables in previous tests symbolic and have a backtracking mechanism, which also have to be carefully tuned, since having symbolic values will increase overhead and frequent backtracking may also slow down incremental test generation.

(2) Comparison to other coverage metrics. We have specifically compared incremental and regular test generation satisfying OMC/DC. It would be interesting to see how OMC/DC tests perform against tests generated from other means, such as test suites satisfying requirements-based coverage, which may also address the propagation from inputs to outputs.

REFERENCES

- [1] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.
- [2] RTCA, "DO-178C, software considerations in airborne systems and equipment certification," 2011.
- [3] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 161–170.
- [4] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation," in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, 1997, pp. 418–425.
- [5] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 102–111.
- [6] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.
- [7] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.
- [8] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.
- [9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.
- [10] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proceedings of the 5th European Dependable Computing Conference*, 2005, pp. 281–292.
- [11] N. Tillmann and J. De Halleux, "Pex—white box test generation for .NET," in *Tests and Proofs*, 2008, pp. 134–153.
- [12] P. Godefroid, M. Y. Levin, D. A. Molnar et al., "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, vol. 8, 2008, pp. 151–166.
- [13] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, *A practical tutorial on modified condition/decision coverage*. NASA Langley Research Center, 2001.
- [14] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM-efficient computation of observability-based code coverage metrics for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 1003–1015, 2001.
- [15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.
- [16] A. Murugesan, S. Rayadurgam, and M. P. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, 2013, pp. 13–17.
- [17] "MathWorks Simulink," <http://www.mathworks.com/products/simulink>, August 2015.
- [18] "MathWorks Stateflow," <http://www.mathworks.com/products/stateflow>, August 2015.
- [19] "MathWorks Matlab Coder," <http://www.mathworks.com/products/matlab-coder>, August 2015.
- [20] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proceedings of 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999, pp. 146–162.
- [21] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," in *Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2001, pp. 83–91.
- [22] "JKind," <https://github.com/agacek/jkind>, August 2015.
- [23] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th International Conference on Software Engineering*, 1995, pp. 41–41.
- [24] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.
- [25] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.
- [26] A. Rajan, M. Whalen, M. Staats, and M. P. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," in *Formal Methods and Software Engineering*, 2008, pp. 86–104.
- [27] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "Automated oracle data selection support," *IEEE Transactions on Software Engineering*, 2015.
- [28] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 435–445.
- [29] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 57–68.
- [30] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007, pp. 196–206.
- [31] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, 2006, pp. 419–423.
- [32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 75–84.
- [33] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.
- [34] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.
- [35] G. Hamon, L. De Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, 2004, pp. 261–270.
- [36] G. Fraser and A. Gargantini, "Experiments on the test case length in specification based test case generation," in *Proceedings of the 2009 ICSE Workshop on Automation of Software Test*, 2009, pp. 18–26.

Automated Oracle Data Selection Support

Gregory Gay, Matt Staats, Michael Whalen, *Senior Member, IEEE*, and
Mats P.E. Heimdahl, *Senior Member, IEEE*

Abstract—The choice of test oracle—the artifact that determines whether an application under test executes correctly—can significantly impact the effectiveness of the testing process. However, despite the prevalence of tools that support test input selection, little work exists for supporting oracle creation. We propose a method of supporting test oracle creation that automatically selects the *oracle data*—the set of variables monitored during testing—for expected value test oracles. This approach is based on the use of mutation analysis to rank variables in terms of fault-finding effectiveness, thus automating the selection of the oracle data. Experimental results obtained by employing our method over six industrial systems (while varying test input types and the number of generated mutants) indicate that our method—when paired with test inputs generated either at random or to satisfy specific structural coverage criteria—may be a cost-effective approach for producing small, effective oracle data sets, with fault finding improvements over current industrial best practice of up to 1,435% observed (with typical improvements of up to 50%).

Index Terms—Testing, Test Oracles, Oracle Data, Oracle Selection, Verification

1 INTRODUCTION

There are two key artifacts to consider when testing software: the *test inputs* and the *test oracle*, which determines if the system executes correctly. Substantial research has focused on supporting the creation of test inputs but little attention has been paid to the creation of oracles [1]. However, existing research indicates that the choice of oracle has a significant impact on the effectiveness of testing [2], [3], [4], [5]. Therefore, we are interested in the development of automated tools that support the creation of a test oracle.

Consider the following testing process: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgement, among others; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. This type of oracle is known as an *expected value test oracle* [3]. Experience with industrial practitioners indicates that such test oracles are commonly used in testing critical systems, such as avionics or medical device systems.

Our goals here are twofold. First, the current practice when constructing expected value test oracles is to define expected values for only the outputs, a practice that can be suboptimal when faults that occur inside the system fail to propagate to these observed variables. Second, manually defining expected values is a time-consuming and, consequently, expensive pro-

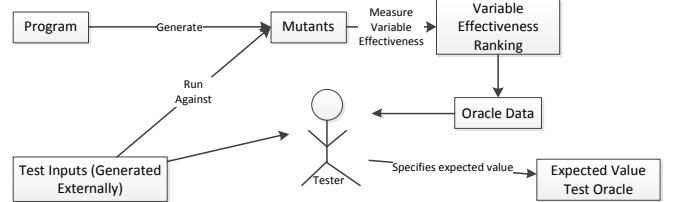


Fig. 1. Supporting Expected Value Test Oracle Creation process. It is simply not feasible to monitor everything¹. Even in situations where an executable software specification can be used as an oracle—for instance, in some model-based development scenarios—limited visibility into embedded systems or the high cost of logging often make it highly desirable to have the oracle observe only a small subset of all variables.

To address these goals, we present and evaluate an approach for automatically selecting *oracle data*—the set of variables for which expected values are defined [4]—that aims at maximizing the fault finding potential of the testing process relative to cost. This oracle data selection process, illustrated in Figure 1, is completely automated. First, we generate a collection of mutants from the system under test. Second, an externally generated test suite is run against the mutants using the original system as the oracle and logs of the values of a candidate set of variables are recorded after certain points in execution (i.e., at a certain timing granularity or after pre-defined execution points). Third, we use these logs to measure how often each variable in the candidate set reveals a fault in a mutant and, based on this information, we rank variable effectiveness. Finally, based on this ranking, we estimate which variables to include in the oracle data. The underlying hypothesis is that, as with mutation-based test data selection, oracle data that is likely to reveal faults in the mutants will also be likely to reveal faults in the actual system under test. Once

G. Gay is with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: greg@greggay.com

M. Whalen and M. Heimdahl are with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: [mwhalen,heimdahl]@cs.umn.edu

M. Staats is with Google, Inc. E-Mail: staatsm@gmail.com

This work has been supported by NASA Ames Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583, CNS-0931931, and CNS-1035715, an NSF graduate fellowship, and the Fonds National de la Recherche, Luxembourg (FNR/P10/03). We would additionally like to thank Rockwell Collins Inc. for their support.

1. In some scenarios, such as regression testing, we can automate definition of expected values. Comparing large numbers of expected values is still potentially expensive, and our approach is still of value in such a scenario.

this oracle data is selected, the tester defines expected values for each element of the oracle data. Testing then commences with a small, and potentially highly effective, oracle.

In previous work, we proposed this approach and applied it to a fixed number of mutants and test inputs generated to satisfy two specific structural coverage criteria [6]. Although the results were promising, the initial study was limited, and did not yield enough evidence to identify the specific conditions in which our approach would be useful. Specifically, we did not know how the use of structural coverage criteria—as compared to, for example, tests generated to assess compliance of a program to its specification—or the number of mutants used during training impacted our results.

To better evaluate our hypothesis and find answers to these additional questions, we have evaluated our approach using four commercial sub-systems from the civil avionics domain and two medical device systems, and using four different types of test input data—tests generated to satisfy two structural coverage metrics, tests generated to satisfy coverage of the requirements, and tests generated purely at random. We perform a comparison against two common *baseline approaches*: (1) **current practice**, favoring the outputs of the system under test as oracle data, and (2) **random selection** of the oracle data set. We also compare to an idealized scenario where the seeded faults in the test suites and mutants used for training are identical to the test suites and mutants in the final evaluation set; thus, providing an estimate of the maximum fault finding effectiveness we could hope to achieve with a mutation-based oracle data selection support method. We repeat the experiment using varying numbers of mutants in order to determine the amount of data needed to construct powerful, stable sets of oracle data.

We draw from our results three key conclusions:

- Our approach generally produces more effective test oracles than the alternative methods explored—in particular, outperforming output-based test oracles with observed improvements in fault finding of up to 1,435%, and consistent improvements of up to 50%. Even in cases where our approach is not effective, the results achieved are similar to using output-based test oracles.
- Our approach is the most effective when paired with test suites generated to exercise the internal structure of the program under test. When our approach is applied to randomly generated test inputs, improvements are more subdued (3%-43%), and improvements are nearly non-existent when applied to requirements-based tests.
- Finally, the choice of test inputs impacts the number of mutants required to derive effective sets of oracle data. In our study, for a given system, the oracle data for structural test suites is improved by the use of a large number of mutants (up to 125 mutants). For requirements based tests suites, no improvements are generally observed after the number of mutants exceeds 50.

We therefore conclude that our approach may be a cost effective method of supporting the creation of an oracle data set—particularly in scenarios where test suites are generated to satisfy structural coverage criteria.

2 BACKGROUND & RELATED WORK

In software testing, a *test oracle* is the artifact used to determine whether the software is working correctly [7]. There are many types of test oracles, ranging from program invariants to “no crash oracles” [1]. In our experience with industrial partners developing critical software systems, one commonly used class of test oracles are *expected value test oracles*—test oracles that, for each test input, specify concrete values the system is expected to produce for one or more variables (internal state and/or output). Thus, in designing an expected value test oracle, two questions must be answered: (1) “*what variables should be monitored?*” and (2), “*how often should the values of those variables be checked?*”.

During testing, the oracle compares the actual values produced by the program against the expected values at the selected points in execution. In current practice, testers generally must manually select these expected values without the aid of automation; this process is naturally dependent on the skill of the tester. Our goal is therefore to develop tools and techniques to support and optimize the selection of the oracle data. While others have focused on the latter question—how often values should be checked [8]—we focus on the former question: what variables should be monitored for faults?

In Richardson et al.’s definition of a test oracle, an oracle is composed of two parts—the oracle *information* and oracle *procedure* [7]. The oracle procedure renders a pass/fail verdict on an executed test at selected points in execution using the data collected in the oracle information. For expected value oracles, the oracle information must contain two distinct sets of information: the oracle *value set* and the oracle *data set*. The oracle data set is the subset of variables (internal state and outputs) for which expected values are specified; i.e., what variables the oracle must monitor. The oracle value set is then the set of expected values for those variables [3], [6].

For example, an oracle may specify expected values for all of the outputs; we term this an *output-only* oracle. This type of oracle appears to be the most common expected value oracle used in testing critical systems. Other types of test oracles include, for example, *output-base* oracles, whose oracle data set contains all the outputs, followed by some number of internal state variables, and *maximum* oracles, whose oracle data set contains *all* of the outputs and internal state variables.

It has been empirically shown that larger oracle data sets are generally more powerful than smaller oracle data sets [3], [5], [2]. (In the remainder of this paper the *size* of an oracle is the number of variables used in the oracle data set.) This is due to *fault propagation*—faults leading to incorrect states usually propagate and manifest themselves as failures for only a subset of program variables; larger oracle data sets increase the likelihood that such a variable will be monitored by the test oracle. This limited fault propagation is particularly a concern in the avionics community, where complex Boolean expressions can mask out faults and prevent incorrect values from affecting output variables. Common practice dictates focusing on the output variables, but the effects of this masking can delay or prevent the propagation of faults to these variables. It would be desirable to identify the variables whose definition

creates the potential for masking and check the behavior of these internal “bottleneck” points.

Naturally, one solution is to use the maximum oracle, an oracle that observes and checks the behavior of all variables in the system. This is always the most effective expected value test oracle, but using it is often prohibitively expensive. This is particularly the case when (1) expected values must be manually specified—a highly labor intensive process, especially when results are checked at multiple execution points or time steps during a single test—or (2) when the cost of monitoring a large oracle data set is prohibitive, e.g., when testing embedded software on the target platform.

One might ask at this point if a “fault” is really a fault if it fails to propagate to an output variable? Frequently the system under test may be capable of *absorbing* the fault, as the corrupt program state fails to impact the final behavior of the software. Indeed, in some cases these “faults” cannot ever propagate to an output. However, in practice (as we will see later in Section 5) such faults typically do propagate to the output under some circumstance, and thus these faults should still be identified and corrected. This is particularly true for safety-critical systems, such as avionic or medical devices, where undiscovered faults can lead to loss of equipment or harm to human operators. This motivates the use of internal variables, which allow testers to correct more issues in the system without incurring the cost of the maximum oracle.

2.1 Related Work

Work on test oracles often focuses on methods of constructing oracles from other software engineering artifacts, such as formal software requirements [1]. In our work, we are not constructing the entire oracle; rather, we are identifying effective oracle data sets, from which effective expected value oracles can be built. We are not aware of any work proposing or evaluating alternative methods of selecting the oracle data.

Voas and Miller have proposed the PIE approach, that—like our work—relies on a form of mutation analysis [9]. Their approach could be used to select internal variables for monitoring, though evaluation of this idea is lacking. More recent work has demonstrated how test oracle selection can impact the effectiveness of testing, indicating a need for effective oracle selection techniques [4], [3].

Memon and Xie have applied mutation testing in order to optimize the oracle procedure in an expected value test oracle for event-driven graphical user interfaces [8]. The authors’ goal was to construct cheaper—but still effective—test oracles by considering *how often* to invoke the potentially expensive oracle procedure (i.e., to compare expected and actual values). The authors found that (1) transient faults are as common as persistent ones, and (2) the majority of transient faults were linked to two event types. Thus, by comparing values only after those events, the oracle can catch the majority of faults while remaining computationally affordable.

Memon and Xie’s goals are similar to ours—they are supporting the construction of efficient expected value test oracles. However, our respective domains require different approaches to this task. In the avionics systems that we have

studied, complexity stems from the hundreds or thousands of variables that can potentially be monitored, and there are not necessarily discrete events that we can choose to ignore (especially when considering critical systems). Thus, even if we could execute the oracle procedure less often, we are still left with the question of which variables to monitor.

Several tools exist for automatically generating invariant-based test oracles for use in regression testing, including Eclat [10], DiffGen [11], and work by Evans and Savoy [12]. However, Briand et al. demonstrate for object-oriented systems that expected value oracles outperform state-based invariants, with the former detecting faults missed by the latter [2].

Fraser and Zeller use mutation testing to generate both test inputs and test oracles [13] for Java programs. The test inputs are generated first, followed by generation of post-conditions capable of distinguishing the mutants from the program with respect to the test inputs. Unlike our work, the end result of their approach is a *complete* test case, with inputs paired with expected results (in this case, assertions). Such tests, being generated from the program under test, are guaranteed to pass (except when the program crashes). Accordingly, the role of the user in their approach is to decide, for each input and assertion pair, if the program is working correctly. Thus, in some sense their approach is more akin to invariant generation than traditional software testing. The most recent version of this work attempts to generalize the result of their approach to simplify the user’s task [14]. However, this creates the possibility of producing *false positives*, where a resulting parameterized input/assertion can indicate faults when none exist—further changing the user’s task.

With respect to evaluation, no comparisons against baseline methods of automated oracle selection are performed; Generated tests and assertions are compared against developer-produced tests and assertions, but the cost—i.e., number of developer tests/assertions—is not controlled. Thus, relative cost-effectiveness cannot accurately be assessed. Additionally, their approach selects enough tests and assertions to detect *all* generated mutations, and thus allows no method of controlling for the human cost.

Our work chiefly differs from other approaches in that we are trying to *support* creation of a test oracle, rather than automate it. Oracle creation support can be considered an approach to addressing the *human oracle cost problem*—that is, the problem of alleviating the burden on the human tester when no means exist to completely automate the generation of a test oracle [1]. Other approaches to addressing the human oracle problem include the automated generation of human-readable test inputs [15], test suite reduction aimed at reducing human workload [16], and incorporating knowledge from project artifacts in order to generate test cases that human testers can easily understand [17].

This report is an extension of previously published work [6], and improves upon it in two key ways. First, we explore the effectiveness of our approach with respect to a variety of test input types. Second, we investigate how the number of mutants generated impacts the process. These analyses help us to better understand the effectiveness of our approach in different testing contexts—in particular, how structural coverage may

be helped by good oracle data selection—and the cost and scalability of our approach.

3 ORACLE DATA SELECTION

Our approach for selecting the oracle data set is based on the use of mutation testing [18]. In mutation testing, a large set of programs—termed *mutants*—are created by seeding faults (either automatically or by hand) into a system. Test input capable of distinguishing the mutant from the original is said to *kill* the mutant. In our work, we adopt this approach for oracle creation support. Rather than generate test inputs that kill the mutants, however, we use mutants to automatically generate an oracle data set that—when used in an expected value oracle, and with a fixed set of test inputs—kills the mutants. To construct this data set, we perform the following:

- 1) Generate several mutants, called the *training set*, from our system under test.
- 2) Run test inputs over the training set and the original system, collecting logs of the values of a set of candidate variables at pre-defined observation points.
- 3) Use these logs to determine which variables distinguish each mutant from the original system.
- 4) Process this information to create a list of variables ordered in terms of apparent fault finding effectiveness, the *variable ranking*.
- 5) Examine this ranking, along with the mutants and test inputs, to estimate (as x) how large the oracle data set should be. Alternatively, the tester can specify x based on the testing budget.
- 6) Select the top x variables in the ranking for use in the final oracle data set.

While conceptually simple, there are several relevant parameters to be considered for each step. The following subsections will outline these parameters, as well as the rationale for the decisions that we have made.

3.1 Mutant Generation and Test Input Source

During mutation testing, *mutants* are created from an implementation of a system by introducing a single fault into the program. Each fault results from either inserting a new operator into the system or by replacing an operator or variable with a different operator or variable. This mutation generation is designed such that no mutant will “crash” the system under test. The mutation testing operators used in this experiment include changing an arithmetic operator, changing a relational operator, changing a Boolean operator, introducing the Boolean \neg operator, using the value of a variable from the previous computation cycle, changing a constant expression by adding or subtracting 1 from int and real constants (or by negating Boolean constants), and substituting a variable reference with another variable of the same type.

The type of faults used to create mutants may impact the effectiveness of the selected oracle data when used to test the actual system under test. Note that the type of mutants used in the evaluation in this report are similar to those used by Andrews et al., where the authors found that generated

mutants are a reasonable substitute for actual failures in testing experiments [19]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [20]. This offers evidence that mutation-based techniques will be useful for supporting the creation of oracles for real-world systems.

Our approach can be used with any set of test inputs. In this work, we assume the tester is equipped with an existing set of test inputs and wishes to determine what oracle data is likely to be effective with said test inputs. This assumption allows the numerous existing methods of test input selection to be paired with our approach for oracle data selection. This scenario is likely within our domain of interest.

3.2 Variable Ranking

Once we have generated mutants, we then run suites of test inputs over both the mutants and the original program (with the execution over the original program serving as a golden run [21]). The user chooses a set of internal and output variables to serve as candidates for oracle data selection. Then, during execution of these inputs, we collect the value of every variable in that candidate set at various points during execution. A variable has detected a fault when the variable value in the original “correct” system differs from the variable value produced by a mutant, for some test. We track the mutants killed by each variable.

In order to form an oracle data set, snapshots of the state of the candidate variables must be collected. *When and how often* those snapshots are logged will help determine what data is available to our variable ranking approach. Options can generally be divided into *event-driven* logging or *time-driven* logging. In the event-driven case, the state of the candidate variables might be logged at certain pre-defined points in execution. This could include, for example, after discrete computational cycles, when new input is passed to the system, when new output is issued from the system, or when certain types of events occur. In a time-driven process, snapshots could be taken at a certain timing granularity—say, every X seconds.

The more often snapshots are taken, the more data the variable ranking algorithm has to work with. However, this logging also incurs a computational overhead on the execution of the system. Therefore, the selection of logging frequency depends on the project that an oracle is being produced for.

Once we have collected these traces, we can produce a set of variables ranked according to effectiveness. One possible method of producing this ranking is simply to order variables by the number of mutants killed. However, the effectiveness of individual variables can be highly correlated. For example, when a variable v_a is computed using the value of a variable v_b : if v_b is incorrect for some test input, it is highly probable that v_a is also incorrect. Thus, while v_a or v_b may be highly effective when used in the oracle data set, the combination of both is likely to be only marginally more effective than the use of either alone.

To avoid selecting a set of dependent variables that are individually effective—but duplicative as a group—we make

use of a greedy algorithm for solving the *set covering problem* [22] to produce a ranked set of variables. In the set covering problem, we are given several sets with some elements potentially shared between the sets. The goal is then to select the minimum set of elements such that one element from each set has been selected. In this problem, each set represents a mutant, and each element of the set is a variable capable of detecting the mutant for at least one of the test inputs. Calculating the smallest possible set covering is an NP-complete problem [23]. Thus, we employ a well-known effective greedy algorithm to solve the problem [24]: (1) select the element covering the largest number of sets, (2) remove from consideration all sets covered by said element, and (3) repeat until all sets are covered.

In our case, each element removed corresponds to a variable. These variables are placed in a ranking in the order they were removed (with the most effective variables being removed first). The resulting ranking can then be used to produce an oracle data set of size n by simply selecting the top n variables from the list.

In the case examples studied, all variables are *scalar* and cannot be a heap object or pointers. Thus, comparison is straightforward. As our approach requires only that expected and actual values differ (rather than *how* they differ), mutation-based oracle optimization should be effective when making comparisons of any data structure, as long as an accurate and efficient method of comparing for equality exists.

Additionally, the systems explored in this work contain no nested loops. For a “step” of the system, every variable is declared and assigned exactly once. Thus conceptually, there exists no difference between output variables and internal variables in terms of how expected values should be defined.

3.3 Estimating Useful Oracle Data Size

If the testers would—by default—use the output variables of the system as their oracle data set, then a natural use of this mutation-based technique would be to select a new oracle data set of the same size. That is, if the testers would have used n output variables in their oracle data set, then they could use our technique to select n variables from the full set of internal and output variables.

However, one potential strength of our technique is that it can produce oracle data sets of any size, granting freedom to testers to choose an oracle data set to fit their budget, schedule, monitoring limitations, or other testing concerns. Once we have calculated the ranked list of variables, we can select an oracle data set of size 1, 2, 3, etc. up to the maximum number of variables in the system.

In some scenarios, the tester may have little guidance as to the appropriate size of the oracle data. In such a scenario, it would be ideal to offer a recommendation to the tester. One would like to select an oracle data set such that the size of the set balances cost and effectiveness—that is, not so small that potentially useful variables are omitted, and not so large that a significant number of variables contribute little to performance.

To accomplish this, testers could examine the fault finding effectiveness of oracle data sets of size 1, 2, 3, etc. The

	# Subsystems	# Blocks	# Output Variables	# Internal Variables
DWM_1	3109	11,439	7	569
DWM_2	128	429	9	115
Vertmax	396	1,453	2	415
Latefl	120	718	1	128

	# States	# Transitions	# Output Variables	# Internal Variables
Infusion_Mgr	27	50	5	107
Alarms	78	107	5	182
Infusion_Mgr (faulty)	30	47	5	86
Alarms (faulty)	81	101	5	155

TABLE 1
Case Example Information

effectiveness of these oracles will increase with the oracle’s size, but the increases will likely diminish as the oracle size increases. As a result, it is generally possible to define a natural cutoff point for recommending an oracle size; if the fault finding improvement between an oracle of size n and size $n + 1$ is less than some threshold, we recommend an oracle of size n .

In practice, establishing a threshold will depend on factors specific to the testing process. In our evaluation, we examine oracle sizes up to $\max(10, (2 * \# \text{ output variables}))$ and explore two potential thresholds: 5% and 2.5%.

4 EVALUATION

We wish to evaluate whether our approach yields effective oracle data sets. While it would be preferable to directly compare against existing algorithms for selecting oracle data, to the best of our knowledge, no such methods exist. We therefore compare our technique against two *baseline approaches* for oracle data set selection, detailed later, as well as against an idealized *best case* application of our own approach.

We also would like to determine the impact of the choice of test input generation criteria on our approach. In particular, we are interested if the effectiveness varies when moving from tests generated to satisfy structural coverage criteria—which tend to be short and targeted at specific code constructs—to requirements-based test inputs, which tend to be longer and are directly concerned with showing the relationship between the inputs and outputs (i.e., they attempt to cause values to propagate through the system). Finally, we are interested in cost and scalability, specifically how the number of mutants used to select the oracle data impacts the effectiveness of the resulting oracle.

We have explored the following research questions:

Research Question 1 (RQ1): *Is our approach more effective in practice than baseline approaches to oracle data selection?*

Research Question 2 (RQ2): *What is the maximum potential effectiveness of the mutation-based approach, and how effective is the realistic application of our approach in comparison?*

Research Question 3 (RQ3): *How does the choice of test input data impact the effectiveness of our approach?*

Research Question 4 (RQ4): *What impact does the number of training mutants have on the effectiveness of our approach?*

Research Question 5 (RQ5): *What is the ratio of output to internal variables in the generated oracle data sets?*

4.1 Experimental Setup Overview

We have used four industrial systems developed by Rockwell Collins Inc. engineers and two additional subsystems of an infusion pump created for medical device research [25]. The Rockwell Collins systems were modeled using the Simulink notation from Mathworks Inc. [26] and the remaining systems using Stateflow [26], [27]. The systems were automatically translated into the Lustre programming language [28] to take advantage of existing automation. In practice, Lustre would then be automatically translated to C code. This translation is a simple transformation, and if applied to C, the case study results would be identical.

The four Rockwell Collins systems represent sizable, operational modules of industrial avionics systems. Two systems, *DWM1* and *DWM2*, represent distinct subsystems of a Display Window Manager (DWM) for a commercial cockpit display system. Two other systems, *Vertmax_Batch* and *Latctl_Batch*, describe the vertical and lateral mode logic for a Flight Guidance System. The remaining two systems, *Infusion_Mgr* and *Alarms*, represent the prescription management and alarm-induced behavior of an infusion pump. Both systems come with a set of real faults that we can use to assess real-world fault-finding.

Information related to these systems is provided in Table 1. Subsystems indicates the number of Simulink subsystems presents, while blocks represents the number of blocks used. Outputs and internals indicates the number of output and internal variables present. For the examples developed in Stateflow, we list the number of Stateflow states, transitions, and internal and output variables. As we have both faulty and corrected versions of *Infusion_Mgr* and *Alarms*, we list information for both.

For the purposes of the study conducted in this work, we automatically generate tests—both randomly and with a coverage-directed search algorithm—that are effectively *unit tests* for modules of synchronous reactive systems. Computation for synchronous reactive systems takes place over a number of execution “cycles.” That is, when input is fed to the system, there is a corresponding calculation of the internal variables and outputs. A single cycle can be considered a sequence of assignments of values to variables. A loop would be considered as a series of computational cycles. This naturally answers the question of “when” to log variable values for oracle generation—after each computational cycle, we can log or check the current value of the candidate variables.

For each case example, we performed the following steps:

- 1) **Generated test input suites:** We created 10 test suites satisfying decision coverage, and 10 test suites satisfying MC/DC coverage, and—for systems with known requirements—10 test suites satisfying UFC (requirements) coverage using automatic counterexample-based test generation. We also produced randomly constructed test suites of increasing size. (Section 4.2.)
- 2) **Generated training sets:** We randomly generated 10 sets of 125 mutants to be used to construct oracle data sets, each containing a single fault. (Section 4.3.)
- 3) **Generated evaluation sets:** For each training set, we

randomly generated a corresponding evaluation set of 125 mutants, each containing a single fault. Each mutant in a evaluation set is guaranteed to *not* be in the training set. (Section 4.3.)

- 4) **Ran test suite on mutants:** We ran each mutant (from both training and evaluation sets) and the original case example using every test suite and collected the internal state and output variable values produced after each computation cycle. This yields raw data used for the remaining steps of our study. (Section 4.5.)
- 5) **Generated oracle data sets:** We used the information gathered to generate oracle data sets using the algorithm detailed in Section 3. Data sets were generated for each training set *and* for each evaluation set (in order to calculate an idealized ceiling performance). We also generated random and output-based baseline rankings. These rankings are used to generate oracles of various sizes. (Section 4.5.)
- 6) **Assessed fault finding ability of each oracle and test suite combination:** We determined how many mutants were detected by every oracle, using each test suite. For oracles generated using a training set, the corresponding evaluation set was used; for oracles generated using an evaluation set, the same evaluation set was used. For the *Infusion_Mgr* and *Alarms* systems, we also assess the performance of each oracle and test suite combination on the set of real faults (Section 4.6.)

4.2 Test Suite Generation

As noted previously, we assume the tester has an existing set of test inputs. Consequently, our approach can be used with any method of test input selection. As we are studying the effectiveness using avionics systems, two structural coverage criteria are likely to be employed: decision coverage and Modified Condition/Decision Coverage (MC/DC) [29].

Decision coverage is a criterion concerned with exercising the different outcomes of the Boolean decisions within a program. Given the expression, $((a \text{ and } b) \text{ and } (\text{not } c \text{ or } d))$, tests would need to be produced where the expression evaluates to true and the statement evaluated to false, causing program execution to traverse both outcomes of the decision point. Decision coverage is similar to the commonly-used branch coverage. Branch coverage is only applicable to Boolean decisions that cause program execution to branch, such as that in “if” or “case” statements, whereas decision coverage requires coverage of all Boolean decisions, whether or not execution diverges. Improving branch coverage is a common goal in automated test generation.

Modified Condition/Decision Coverage further strengthens condition coverage by requiring that each decision evaluate to all possible outcomes (such as in the expression used above), each condition take on all possible outcomes (the conditions shown in the description of condition coverage), and that each condition within a decision be shown to independently impact the outcome of the decision. Independent effect is defined in terms of *masking*, which means that the condition has no effect on the value of the decision as a whole;

for example, given a decision of the form x and y , the truth value of x is irrelevant if y is false, so we state that x is masked out. A condition that is not masked out has *independent effect* for the decision.

Suppose we examine the independent affect of d in the example; if (a and b) evaluates to false, than the decision will evaluate to false, masking the effect of d ; Similarly, if c evaluates to false, then ($\text{not } c$ or d) evaluates to true regardless of the value of d . Only if we assign a , b , and c true does the value of d affect the outcome of the decision.

MC/DC coverage is often mandated when testing critical avionics systems. Accordingly, we view MC/DC as likely to be effective criteria, particularly for the class of systems studied in this report. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [30].

We are also interested in test suites designed to satisfy criteria that are not focused on the internal structure of the system under test, such as **Unique First Cause (UFC)**—a black-box criterion that measures coverage of a set of requirements encoded as temporal logic properties [31]. Adapted from MC/DC a test suite satisfies UFC coverage over a set of requirements—encoded as LTL formulas—if executing the test cases in the test suite guarantees that every basic condition in each formula has taken on all possible outcomes at least once, and each basic condition in each expression has been shown to independently affect the outcome of the expression. As requirement were not available for the *Infusion_Mgr* and *Alarms* systems, we only produce UFC-satisfying tests for the four Rockwell Collins systems.

We used counterexample-based test generation to generate tests satisfying the three coverage criteria [32], [33]. In this approach, each coverage obligation is encoded as a temporal logic formula and the model checker can be used to detect a counterexample (test case) illustrating how the coverage obligation can be covered. By repeating this process for each property of the system, we can use the model checker to automatically derive test sequences that are guaranteed to achieve the maximum possible coverage of the model.

This coverage guarantee is why we have elected to use counterexample-based test generation, as other directed approaches (such as DSE/SAT-based approaches) do not offer such a straightforward guarantee. In the context of avionics systems, the guarantee is highly desirable, as achieving maximum coverage is required [29]. We have used the JKind model checker [34], [35] in our experiments because we have found that it is efficient and produces tests that are easy to understand [36].

Counterexample-based test generation results in a separate test for each coverage obligation. This results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. Such an unnecessarily large test suite is unlikely to be used in practice. We therefore reduce each generated test suite while maintaining coverage. We use a simple randomized greedy algorithm. It begins by determining the coverage obligations satisfied by each test generated, and initializing an empty test set reduced . The algorithm then randomly selects a test input from the full set of tests; if

it satisfies obligations not satisfied by test input already in reduced , it is added to the set. The algorithm continues until all tests have been removed from the full set of tests.

We produce 10 test suites for each combination of case example and coverage criterion to control for the impact of randomization. We also produced 10 suites of random tests—increasing in size from 10 to 100 tests—in order to determine the effectiveness of our approach when applied to test suites that were not designed to fulfill a coverage criterion.

For the systems with real faults, we generate coverage-satisfying tests twice. When calculating fault-finding effectiveness on generated mutants, we generate tests using the corrected version of the system (as the Rockwell Collins systems are free of known faults). However, when assessing the ability of the test suites to find the real faults, we generate the tests using the faulty version of the system. This reflects real-world practice, where—if faults have not yet been discovered—tests have obviously been generated to provide coverage over the code as it currently exists.

4.3 Mutant Generation

For each case example, 250 mutants are created by introducing a single fault into the correct implementation (using the approach discussed in Section 3.1). We then produce 10 *training sets* by randomly selecting 10 subsets of 125 mutants. For each training set, the 125 mutants *not* selected for the training set are used to construct an evaluation set.

Mutants can be divided into *weak mutants*—mutations that infect the program state, but where the result of that corruption does not propagate to a variable checked by the oracle—and *strong mutants*—mutants where state is corrupted and that corruption does propagate. The mutants as they are generated are weak mutants, because there is no a-priori way without analysis of determining whether a mutant is functionally equivalent to the original program. However, we perform a post-processing analysis (described below) to remove functionally equivalent mutants, so the mutants used for testing are *strong mutants*.

We remove *functionally equivalent* mutants from the evaluation set using the JKind model checker [34], [35]. This is possible due to the nature of the systems in our study—each system is finite; thus, determining equivalence is decidable and fast.² This removal is done for the evaluation sets because equivalent mutants represent a potential threat to validity in our evaluation. No mutants are removed from the training sets.

In practice, one would *only* generate a training set of mutants for use in building an oracle data set. We generate both training and evaluation sets in order to measure the performance of the proposed generation approach for research purposes. Thus, while it is possible we select the oracle data based partly on equivalent mutants which cannot affect the output, our evaluation measures the ability of the approach to detect provably faulty systems.

To address Question 4, we also produced four subsets of each training set. These subsets, respectively, contained 10%,

2. Equivalence checking is fairly routine in the hardware domain; a good introduction can be found in [37].

Infusion_Mgr	
1	When entering therapy mode for the first time, infusion can begin if there is an empty drug reservoir.
2	The system has no way to handle a concurrent infusion initiation and cancellation request.
3	If the alarm level is ≥ 2 , no bolus should occur. However, intermittent bolus mode triggers on alarm < 2 .
4	Each time step is assumed to be one second.
5	When patient bolus is in progress and infusion is stopped, the system does not enter the patient lockout. Upon restart, the patient can immediately request an additional dosage.
6	If the time step is not exactly one second, actions that occur at specific intervals might be missed.
7	The system has no way to handle a concurrent infusion initiation and pause request.

Alarms	
1	If an alarm condition occurs during the initialization step, it will not be detected.
2	The Alarms system does not check that the pump is in therapy before issuing therapy-related alarms.
3	Each time step is assumed to be one second.

TABLE 2

Real faults for infusion pump systems

25%, 50%, and 75% of the training mutants. These subsets allow us to determine the effectiveness of oracle data generated with fewer mutants.

4.4 Real Faults

For both of the infusion pump systems—*Infusion_Mgr* and *Alarms*—we have two versions of each case example. One is an untested—but feature-complete—version with several faults, the second is a newer version of the system where those faults have been corrected. We can use the faulty version of each system to assist in determining the effectiveness of each test suite. As with the seeded mutants, effective tests should be able to surface and alert the tester to the residing faults.

For the *Infusion_Mgr* case example, the older version of the system contains seven faults. For the *Alarms* system, there are three faults. Although there are a relatively small number of faults for both systems, several of these are faults that required code changes in several locations to fix. Most are non-trivial faults—these were not mere typos or operand mistakes, they require specific conditions to trigger, and extensive verification efforts were required to identify these faults.

A brief description of the faults can be seen in Table 2.

4.5 Oracle Data Set Generation

For each given case example, we ran the test suites against each mutant and the original version of the program. For each execution of the test suite, we recorded the value of every internal variable and output at each step of every test using an in-house Lustre interpreter. This raw trace data is then used by our algorithm and our evaluation.

For each combination of set of mutants (training sets and evaluation sets) and test suite, we generated an oracle ranking using the approach described in Section 3. The rankings produced from training sets reflect how our approach would be used in practice; these sets are used in evaluating our research questions. The rankings produced from evaluation sets represent an idealized testing scenario, one in which we already know the faults we are attempting to detect. Rankings generated from the evaluations sets, termed *idealized rankings*,

hint at the maximum potential effectiveness of our approach and are used to address Question 2.

Each ranking was limited to m variables (where m is 10 or twice the number of output variables, whichever was larger) since oracles significantly larger than output-only oracles were deemed unlikely to be used in practice. Note that the time required to produce a single ranking—generate mutants, run tests, and apply the greedy set cover algorithm—is less than one hour for each pairing of case example and test suite.

To answer Questions 1 and 3, we compare against two baseline rankings. First, to provide an unbiased ranking for comparison, the *random approach* creates completely random oracle rankings. The resulting rankings are simply a random ordering of the output and internal variables. Second, the *output-base* approach creates rankings by first selecting output variables—ordered at random—and then randomly selecting internal state variables until the size of the oracle matches the pre-determined threshold. Thus, the output-base rankings always lists the outputs first (i.e., more highly ranked) followed by the randomly-selected internal state variables. The output-based ranking reflects a common industrial approach to oracle data selection: focus on the output variables. However, there are two differences between the output-base oracles used in our evaluation and common practice: (1) we randomly vary the order of the output variables to avoid any particular biasing of the results (in the real world, no one order would typically be favored), and (2), we add randomly chosen internal variables to the oracle data set after prioritizing the outputs so that we can compare larger oracle sizes than would typically be employed.

4.6 Oracle Evaluation

To determine the fault finding effectiveness of a test suite t and oracle o on a case example, we simply compare the values produced by the original case example against every mutant using test suite t and the subset of variables corresponding to the oracle data for oracle o (the original “correct” system fulfills the role of the user in our approach, specifying expected values for the oracle data).

For all six case examples, we measure the fault finding effectiveness of oracles generated from the training sets using the corresponding evaluation sets, and we measure the effectiveness of the idealized oracles generated from evaluation sets using the same evaluation sets. The fault finding effectiveness of an oracle is computed as the percentage of mutants killed versus the total number of mutants in the evaluation set. We perform this analysis for each oracle and test suite for every case example, and use the produced results to evaluate our research questions.

For *Infusion_Mgr* and *Alarms*, we also assess the fault-finding effectiveness of each test suite and oracle combination against the version of the model with real faults by measuring the ratio of the number of tests that fail to the total number of tests for each test suite. We use the number of tests rather than number of real faults because all of the real faults are in a single model, and we do not know which specific fault led to a test failure. However, we hypothesize that the test failure ratio is a similar measure of the *sensitivity* of a test suite to

Oracle Size	Decision					
	DWM _1	DWM _2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	66.67%*	109.17%	34.48%	4.89%	1307.14%	11.62%
2	26.79%*	14.09%	29.76%	24.10%	222.22%	41.38%
3	12.50%*	11.11%	31.82%	28.73%	88.49%	44.28%
4	0.00%*	7.50%	35.90%	27.47%	41.94%	42.35%
5	2.94%*	9.45%	43.39%	31.91%	44.99%	27.84%
6	0.00%*	8.21%	45.79%	35.98%	48.49%	28.06%
7	0.00%*	6.55%	46.51%	38.84%	52.10%	20.78%
8	0.00%*	6.49%	49.44%	37.73%	45.95%	19.19%
9	8.01%*	7.96%	46.94%	37.15%	44.59%	21.90%
10	10.82%*	9.03%	52.33%	36.60%	45.17%	23.96%
11	15.83%	9.42%				
12	21.24%	9.46%				
13	23.13%	10.60%				
14	23.53%	10.53%				
15		12.19%				
16		13.46%				
17		13.46%				
18		14.08%				
MC/DC						
Oracle Size	DWM _1	DWM _2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	0.00%*	168.85%	37.50%	0.00%	1435.00%	21.11%
2	9.72%*	7.95%	37.33%	3.12%	305.00%	38.27%
3	-12.92%	7.59%	43.42%	3.16%	125.83%	42.11%
4	-13.96%	7.70%	44.00%	3.96%	35.29%	40.73%
5	-20.00%	7.23%	46.05%	3.96%	45.20%	21.78%
6	-19.23%	5.88%	44.45%	3.98%	46.34%	24.00%
7	-15.69%	4.40%	43.30%	4.40%	48.78%	20.79%
8	-12.00%	2.15%	43.30%	3.96%	51.22%	20.19%
9	-4.81%*	1.02%*	38.27%	3.96%	43.90%	21.83%
10	0.00%*	1.02%*	37.13%	3.64%	38.54%	23.47%
11	3.85%	1.01%*				
12	10.62%	2.00%				
13	12.02%	2.93%				
14	16.95%	3.02%				
15		3.88%				
16		4.77%				
17		4.17%				
18		4.17%				

TABLE 3

Median relative improvement using mutation-based selection over output-base selection for structural coverage-satisfying tests

the mutant kill ratio. Note that we do not generate separate “idealized” oracles when evaluating on real faults, as these would simply be oracles generated from additional mutants, and do not represent the same performance ceiling.

5 RESULTS & DISCUSSION

In this section, we discuss our results in the context of our four research questions. We begin by plotting the median fault finding effectiveness of the produced test oracles for increasing oracle sizes in Figures 2-5.³ Four ranking methods are plotted: both baseline rankings, our mutation-based approach, and an idealized mutation-based approach. For each subfigure, we plot the number of outputs as a dashed, vertical line. This line represents the size of an output-only oracle; this is the oracle size that would generally be used in practice. We also plot the 5% and 2.5% thresholds for recommending oracle sizes as solid lines (see Section 3.3). Note that the 2.5% threshold is not always met for the oracle sizes explored.

In Tables 3-5, we list the median relative improvement in fault finding effectiveness using our proposed oracle data

3. For readability, we do not state “median” relative improvement, “median” fault finding, etc. in the text, though this is what we are referring to.

Oracle Size	Random					
	DWM _1	DWM _2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	24.27%	10.07%	15.50%	0.00%*	60.0%	70.33%
2	15.52%	7.95%	13.07%	0.84%	85.71%	80.00%
3	8.79%	8.29%	15.09%	0.85%*	83.33%	80.91%
4	8.11%	8.71%	16.56%	0.85%*	55.55%	53.14%
5	6.73%	9.54%	17.54%	0.91%*	33.33%	35.14%
6	1.46%*	7.48%	17.20%	0.87%*	33.33%	29.51%
7	-1.11%	4.61%	17.03%	1.64%	37.09%	28.57%
8	0.00%*	0.88%*	16.04%	0.86%	38.46%	27.33%
9	0.00%*	-1.72%	15.47%	0.86%*	28.57%	31.58%
10	1.03%*	-0.93%	15.47%	0.84%*	28.57%	32.58%
11	2.02%	-0.84%				
12	2.39%	0.00%*				
13	4.10%	0.00%*				
14	4.19%	0.85%*				
15		0.88%*				
16		0.88%*				
17		0.95%*				
18		0.97%*				
UFC						
Oracle Size	DWM _1	DWM _2	Vertmax_Batch	Latctl_Batch		
1	0.00%*	8.59%	-0.81%	0.00%*		
2	0.00%*	6.98%	-0.81%	0.00%*		
3	0.00%*	9.53%	-0.81%	0.00%*		
4	-10.56%	9.52%	-0.81%	0.00%*		
5	-11.76%	9.55%	0.00%*	0.83%		
6	-14.29%	6.42%	0.00%*	0.83%		
7	-17.32%	2.73%	0.00%*	0.83%		
8	-13.96%	-0.87%	0.00%*	0.83%		
9	-12.02%	-3.36%	0.00%*	0.83%		
10	-11.11%	-3.36%	0.00%*	0.82%		
11	-10.91%	-2.59%				
12	-6.48%*	-2.56%				
13	-6.56%	-2.52%				
14	-5.96%*	-2.46%				
15		-1.65%				
16		-0.86%				
17		-0.83%				
18		-0.82%				

TABLE 4

Median relative improvement using mutation-based selection over output-base selection for randomly-generated and UFC-satisfying tests

creation approach versus the output-base ranking. In Tables 6-7, we list the median relative improvement in fault finding effectiveness using the idealized mutation-based approach (an oracle data set built and evaluated on the same mutants) versus our mutation-based approach. As shown in Figures 2-5, random oracle data performs poorly; thus, detailed comparisons were deemed uninteresting and are omitted.

5.1 Statistical Analysis

Before discussing the implications of our results, we would like to first determine which differences observed are statistically significant. With regard to *RQ1* and *RQ2*, we would like to determine with significance at what oracle sizes, and for which case examples, (1) the idealized performance of a mutation-based approach outperforms the actual performance of the mutation-based approach, and (2) the mutation-based approach outperforms the baseline ranking approaches. We evaluated the statistical significance of our results using a two-tailed bootstrap permutation test. We begin by formulating the following statistical hypotheses:⁴

4. As we evaluate each hypothesis for each case example and oracle size, we are essentially evaluating a set of statistical hypotheses.

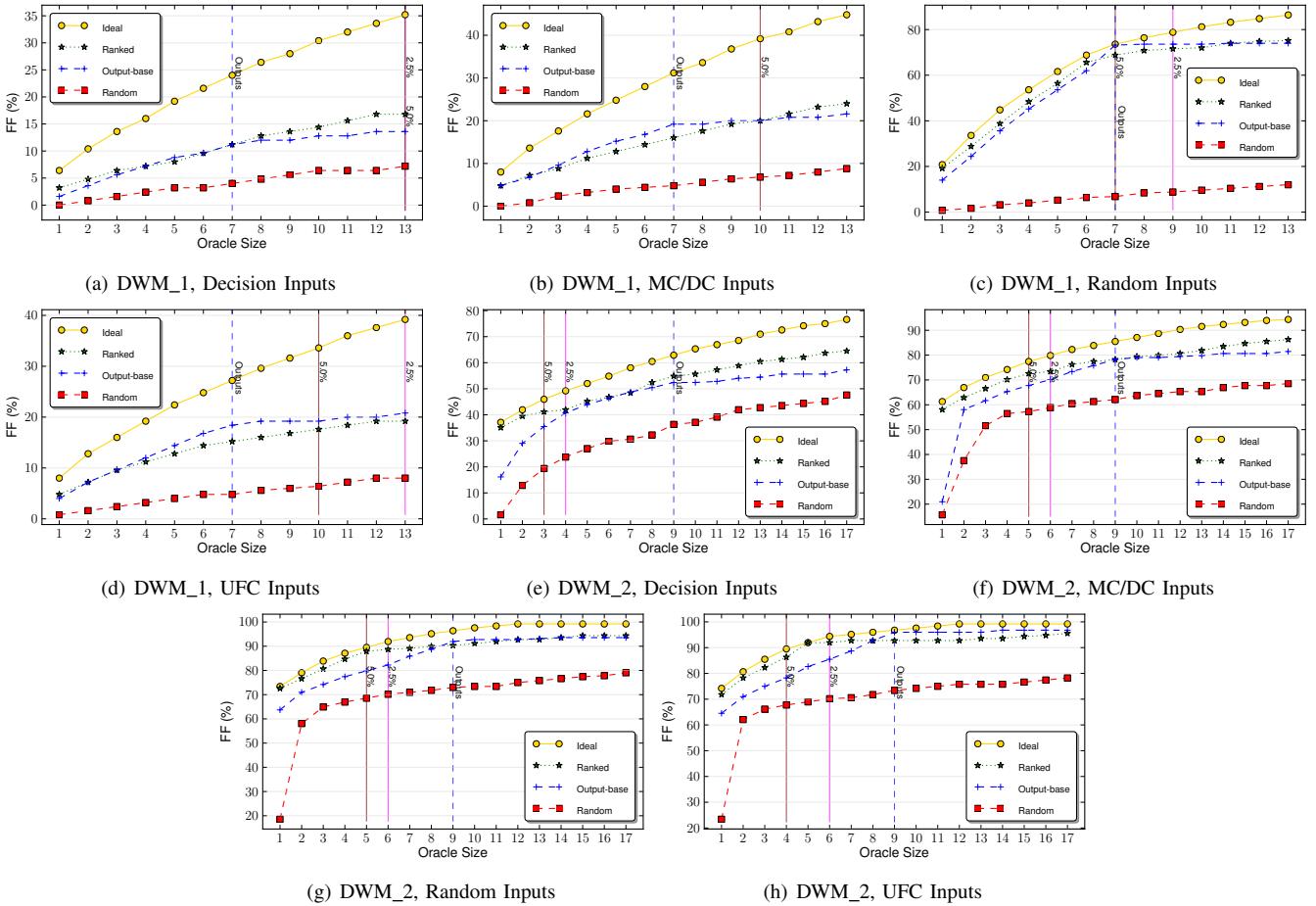


Fig. 2. Median Effectiveness of Various Approaches to Oracle Data Selection for *DWM_1* and *DWM_2*. Yellow Circles = Ideal; Green * = Ranked; Blue + = Output-base; Red Squares = Random.

- H_1 : For a given oracle size m , the standard mutation-based approach outperforms the output-base approach.
- H_2 : For a given oracle size m , the standard mutation-based approach outperforms the random approach.
- H_3 : For a given oracle size m , the idealized approach outperforms the standard mutation-based approach.

Towards *RQ4*, we would also like to quantify—with significance—the number of mutants needed to train the oracle data. We repeated the same experiment, varying the number of mutants used to train the oracle—making use of training sets containing 10%, 25%, 50%, and 75% of the mutants used to train oracles in the initial experiment. We have generated fault finding results using the same evaluation sets, and formulated the following hypothesis.

- H_{4-7} : For a given oracle size m , the standard mutation-based approach, generated using a full training set, outperforms the standard mutation-based approach, generated with $N\%$ of the same training set.

The null hypothesis H_{0X} for each hypothesis H_X above is that each set of values are drawn from the same distribution. To evaluate our hypotheses without any assumptions on the distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution

assumptions [38]), with median as the test statistic. Per our experimental design, each evaluation set has a paired training set, and each training set has paired baseline rankings (output-base and random). Thus, for each combination of case example and coverage criterion, we can pair each test suite T + training set ranking with T + random or output-base ranking (H_{01}, H_{02}), each test suite T + idealized ranking with T + training set ranking (for H_{03}), and finally each test suite T + training set with T + training set subset ranking ($H_{04} - H_{07}$). We then apply and evaluate our null hypotheses for each case example, coverage criteria, and oracle size with $\alpha = 0.05$.⁵ We discuss the results of our statistical tests below in the context of the questions they address.

5.2 Evaluation of Practical Effectiveness (Q1)

When designing a method of supporting oracle creation, the obvious question to ask is, “*Is this better than current best practice?*” In Tables 3 and 4, we list the median relative improvement in fault finding effectiveness on seeded faults using

⁵ Note that we do not generalize across case examples or coverage criteria as the appropriate statistical assumption—random selection from the population of case examples and coverage criteria—is not met. Furthermore, we do not generalize across oracle sizes as it is possible our approach is statistically significant for some sizes, but not others.

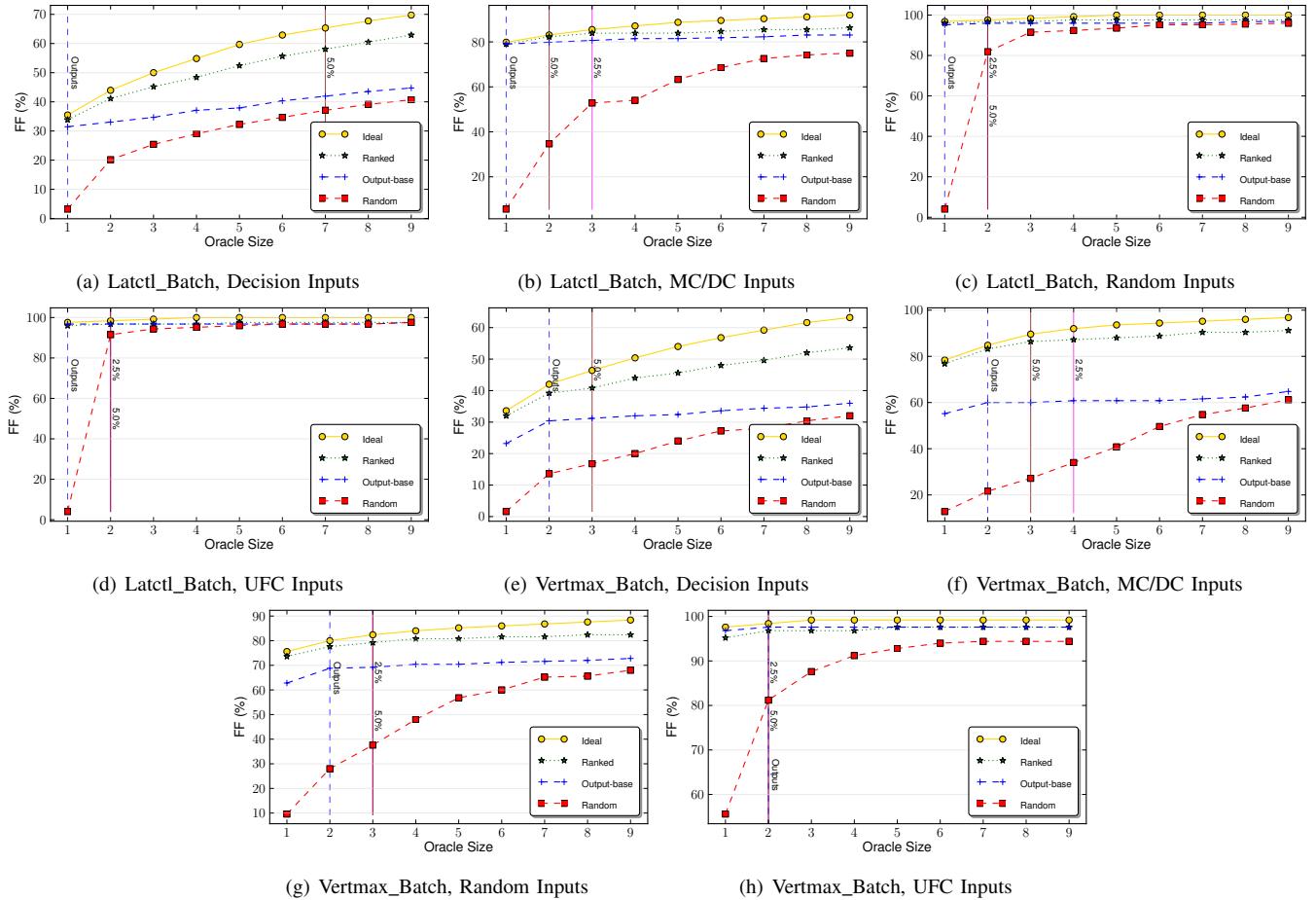


Fig. 3. Median Effectiveness of Various Approaches to Oracle Data Selection for *Latctl_Batch* and *Vertmax_Batch*. Yellow Circles = Ideal; Green * = Ranked; Blue + = Output-base; Red Squares = Random.

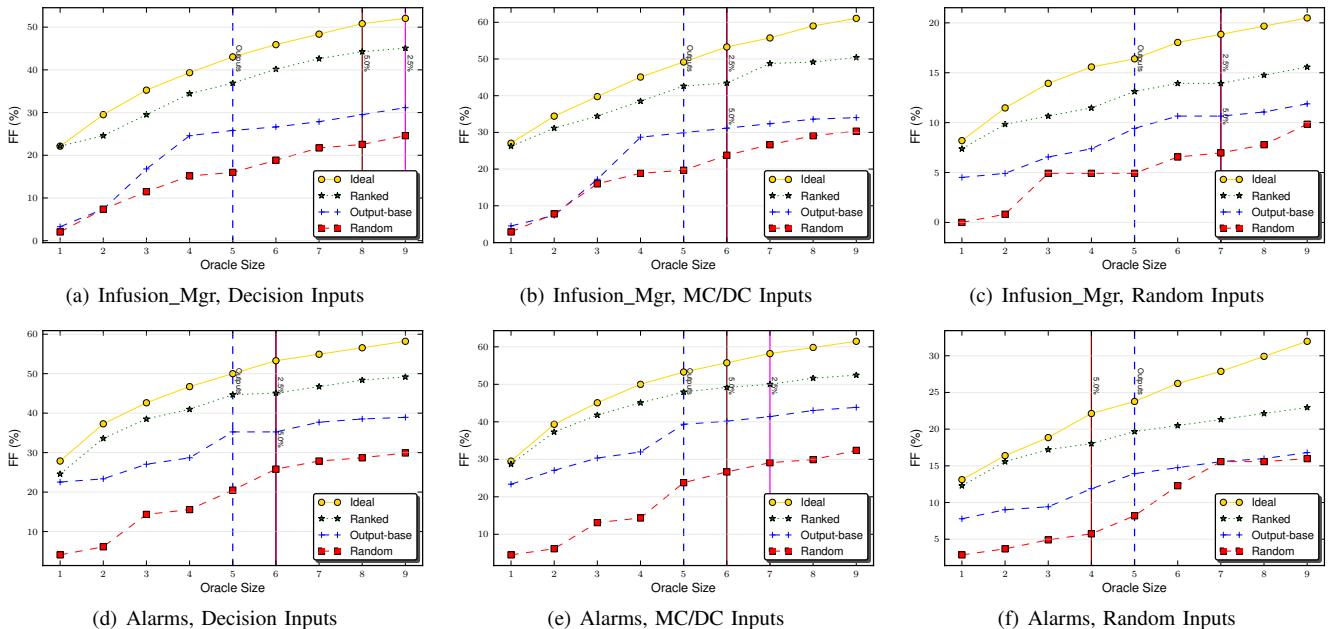


Fig. 4. Median Effectiveness of Various Approaches to Oracle Data Selection for *Infusion_Mgr* and *Alarms*, evaluated on mutants. Yellow Circles = Ideal; Green * = Ranked; Blue + = Output-base; Red Squares = Random.

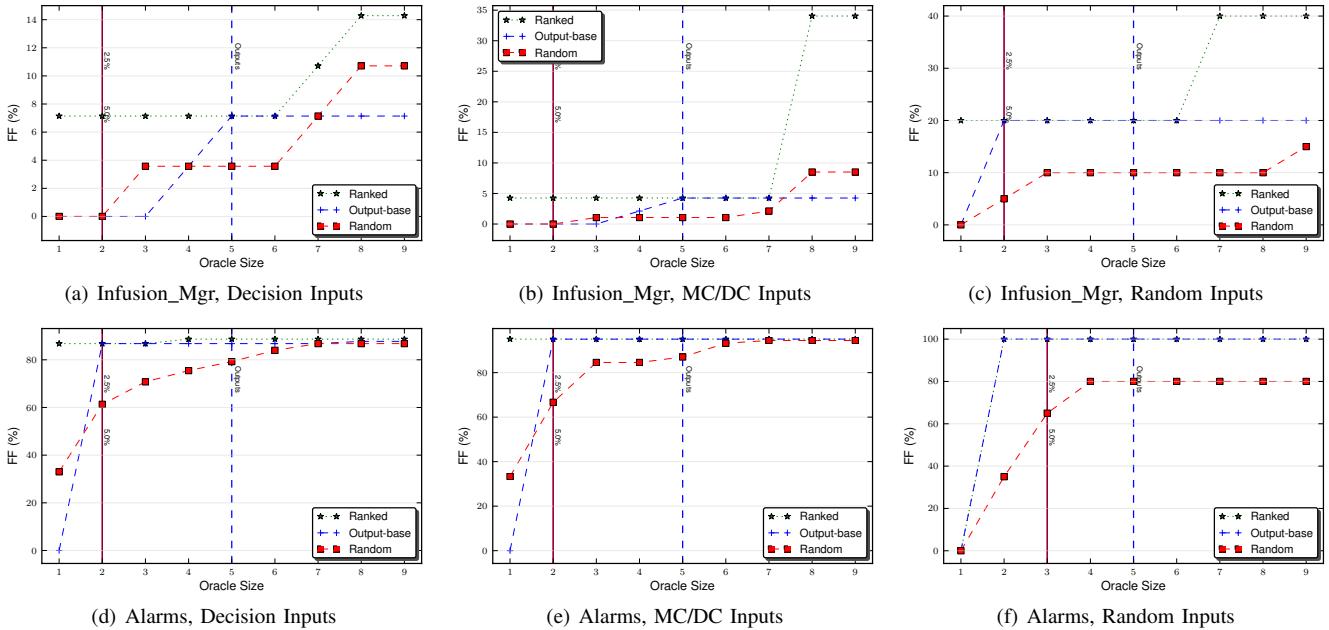


Fig. 5. Median Effectiveness of Various Approaches to Oracle Data Selection for *Infusion_Mgr* and *Alarms*, evaluated on real faults. Yellow Circles = Ideal; Green * = Ranked; Blue + = Output-base; Red Squares = Random.

Oracle Size	Decision	
	Infusion_Mgr	Alarms
1	Inf (0.00% → 7.14%)	Inf (0.00% → 86.78%)
2-3	Inf (0.00% → 7.14%)	0.00%*
4	100%	2.17%
5-6	0.00%*	2.17%
7	0.00%	2.17%
8	9.09%	0.00%*
9-10	99.99%	0.00%*
Oracle Size	MC/DC	
	Infusion_Mgr	Alarms
1	Inf (0.00% → 4.25%)	Inf (0.00% → 86.78%)
2-3	Inf (0.00% → 4.25%)	0.00%*
4	700.0%	0.00%*
5-7	0.00%*	0.00%*
8	700.0%	0.00%*
9-10	349.99%	0.00%*
Oracle Size	Random	
	Infusion_Mgr	Alarms
1	Inf (0.00% → 20.00%)	0.00%*
2-6	0.00%*	0.00%*
7-10	100.00%	0.00%*

TABLE 5

Median relative improvement using mutation-based selection over output-base selection over real faults. “Inf” = output-base oracle failed no tests—we note the median percentage of tests failed by the generated oracle.

our proposed oracle data creation approach versus the output-base ranking—the standard practice of checking the values of the output variables, with additional random variables added to ensure the same number of variables across all oracle types. Relative improvements not statistically significant at $\alpha = 0.05$ level are marked with a *. Almost all of the oracles generated outperform the random approach with statistical significance, often by a wide margin⁶.

From these two tables, we can see that for both struc-

6. Exceptions being output-base vs random on *Infusion_Mgr* with MC/DC and decision inputs at size 2 and *Alarms* with random inputs at size 7

tural coverage criteria, nearly every oracle generated for five of six systems (*Latctl_Batch*, *Vertmax_Batch*, *DWM_2*, *Infusion_Mgr*, and *Alarms*) outperforms the output-base approach with statistical significance. A common pattern can be seen: for oracles smaller than the output-only oracle, our approach tends to perform well compared to output-base, with improvements of up to 1,435%. This reflects the strength of prioritizing variables: we generally select more effective variables for inclusion earlier than the output-base approach. Even in cases where output variables are the most effective, our approach is able to order them in terms of effectiveness. As the test oracle size grows closer in size to the output-only oracle, the relative improvement decreases, but our approach often still outperforms the output-only oracle, up to 45.2%. Finally, as the test oracle grows in size beyond the output-only oracle incorporating (by necessity) internal variables, our relative improvement when using our approach again grows, with improvements of 3.64-52.33% for the larger oracles.

A similar trend can be seen when random tests are used to train the oracle, although the actual improvements are more subdued. For oracles smaller than the output-base approach, improvements of up to 85.71% can be seen. As middle sizes are approached, our approach performs between an identical performance (plus or minus roughly 2%—our method does demonstrate a higher level of variance, as it depends on both a set of training mutants and a particular test suite) and improvements of up to 35.14%. For the largest oracle sizes, modest improvements can be seen—up to 38.46% for the *Infusion_Mgr* system.

In particular, the *Infusion_Mgr* and *Alarms* systems, we see a large improvement in fault-finding effectiveness from using our oracle creation method. This is explained by the structure of these systems. Both systems work to determine the behavior

of an infusion pump by checking sensor readings against a series of complex Boolean conditions. The state spaces of these models are both *deep* and *narrow*, meaning that to reach large portions of the state space, a specific sequence of input values that meet certain combinations of those conditions must occur. Thus, output-based oracles may not detect faults due to *masking*—some expressions in the systems can easily be prevented from influencing the outputs. If masking prevents the effect of a fault from reaching an output variable, then the effectiveness of an output-based oracle will naturally decrease. Our oracle creation approach can select the important output variables and certain internal bottlenecks to observe.

For the structural criteria and random test suites, one key exception can be seen when examining the *DWM_1* system. For this case example, mutation-based oracles tend to be roughly equivalent in effectiveness to output-base oracles (we generally cannot reject $H0_1$ at $\alpha = 0.05$), and at times (particularly for oracles generated using MC/DC satisfying test suites) produce results that are up to 20% worse. It is only for small or large oracles (approximately +6 variables from the output-only oracle) that our approach does well, with up to a 66.67% improvement at smallest sizes and 4.19%-23.53% improvement at the largest recorded oracle size. Examining the composition of these oracles reveals why: the ranking generated using our approach for this case example begins mostly with output variables, and thus oracles generated using our approach are very similar to those generated using the output-base approach. The performance gain at small sizes suggest that certain output variables are far more important than others, but what is crucially important at all levels up to the total number of outputs is *to choose output variables*.

As noted previously, in a small number of instances, our approach in fact does worse than the output-base approach. The issue appears to be that the greedy set-coverage algorithm is overfitting to the training data. If the trace data indicates that there is a highly effective internal state variable, the algorithm will prevent a computationally-related output variable from being selected later in the process. However, it is possible that overall, faults occur more prevalently in that output variable, and that the internal variable is only more fault-prone for the selected set of training data. Given a more optimal set cover algorithm or additional overfitting avoidance improvements, this issue would likely be circumvented. However, for larger oracle sizes, this issue is generally corrected, with statistically significant improvements again being demonstrated.

Very different results are observed when our technique is applied with test inputs generated to satisfy the UFC requirements coverage criterion. When these test suites are used, our technique is, on occasion, modestly successful (up to 9.55% improvement), but more often demonstrates either no improvement (*Vertmax_Batch* and *Latctl_Batch*) or worse results (for oracle sizes surrounding the number of output variables on the *DWM_1* system). We will elaborate on reasons for this difference shortly.

In Table 5, we list the median improvement in fault finding effectiveness on the set of real faults for the *Infusion_Mgr* and *Alarms* systems. On the *Infusion_Mgr* system, we observe the same trends that we saw when evaluating against seeded

mutations. At small oracle sizes, we see an improvement in the percentage of the test suite that fails from 0% of the tests with an output-base oracle up to 20% with a generated oracle. As we approach the number of output variables, the two approaches converge. Finally, at large oracle sizes, our approach improves on the output-base oracle by up to 349%.

Examining the plots for *Infusion_Mgr* in Figure 5 offers insight into the importance of the outputs in finding the real faults embedded into the system—and shows why focusing on the output variables is not always a wise idea. Only two of the five output variables are relevant for finding any of the known faults. This can be seen in the plots, as the median percentage of tests that fail rises exactly twice for the output-base oracle for sizes 1-5. As a result, for the structure-based test inputs, choosing oracle *completely at random* sometimes results in a more effective oracle. If some care is taken in selecting an oracle, time may not be wasted in specifying the expected behavior of outputs that rarely, if ever, are useful for finding faults. Our approach is able to select the important output variables early and automatically suggest additional bottlenecks in the state space.

The results on the version of *Alarms* with real faults are more subdued. At size one, our approach typically outperforms the output-base approach by a large margin—from no failing tests to failure results in a median of 86.78% of the tests. However, from that point, the largest improvement from our approach is by an additional 2.17%. The reason for this can be clearly seen in Figure 5—no matter what variables are chosen, by an oracle of size ten, 90-100% of the tests will have failed. This can be explained by examining the faults listed in Table 2. In particular, the first fault—that if an alarm condition occurs during the first initialization step of execution, it will not be detected in the faulty version of the system—explains the observed results. The majority of the system outputs deal explicitly with signaling alarms in the presence of particular input conditions. If a test triggers that particular fault, then the results of that fault will be obvious at the output level. Our approach is able to select the most important output first, while an unordered approach may not. However, unlike on the *Infusion_Mgr* system, additional gains from observing internal variables are rare.

The major observations made when evaluating on seeded mutations are confirmed by the evaluation against the real faults, indicating the applicability of our approach in practice. However, the performance benefits were more subdued. A likely reason for this effect is due to the potentially substantial differences between the faulty and corrected models (or faulty system and any source of expected values)—in the case of *Alarms*, fixing the model involved adding 24 new transitions and changing the guard conditions on time-related transitions, meaning that any expected value oracle is *very likely* to detect a fault. Examining models that are less substantially different would likely yield different results—for instance, the fixes to the *Infusion_Mgr* model only required two new transitions and changes to a small number of guard conditions.

In addition, the changes imposed by the mutation operators may not be similar to the real mistakes made by the system developers. In particular, mutations do not replicate errors

of omission—leaving out functionality is very different from making a mistake in the implemented functionality. While examining the traces of mutated systems may help us discover important internal variables for identifying faults caused by incorrect implementation, those bottlenecks may not assist in observing errors that stem from missing execution paths (as several of the real faults, listed in Table 2, are). This indicates that—while the core tenants of our mutation-based approach seem correct—there is room for improvement in the sources of the traces used to generate an oracle with our approach. For instance, in addition to seeded faults, it may be possible to train oracles using past revisions of a system (as long as there is enough overlap in the internal structure of the system). The combination of seeded mutations and corrected faults may yield even more effective oracles.

While our approach is of little use when paired with UFC-satisfying test inputs, it does seem clear that our approach can be effective in practice when paired with either randomly-generated test inputs or test suites generated to satisfy a structural coverage criterion. For those test suites, we can consistently generate oracle data sets that are effective over *different* faults, generally outperforming existing ranking approaches. Our approach is able to highlight the most important variables, allowing developers to craft smaller, more effective oracle data sets.

5.3 Potential Effectiveness of Oracle Selection (Q2)

In Tables 6 and 7, we list the median improvement in fault finding effectiveness between the idealized performance (oracle data set built and evaluated on the same mutants) and the real-world performance of our approach. The results which are not statistically significant at $\alpha = 0.05$ are marked with a *.

As noted, there is limited empirical work on test oracle effectiveness. Consequently, it is difficult to determine what constitutes effective oracle data selection—clearly performing well relative to a baseline approach indicates our approach is effective, but it is hard to argue the approach is effective in the absolute sense. We therefore posed *Q2*: what is the *maximum* potential effectiveness of a mutation-based approach? To answer this question, we applied our approach to the same mutants used to evaluate the oracles in *Q1* (as opposed to generating oracles from a disjoint training set). This represents an idealized testing scenario in which we already know what faults we are attempting to find; thus, this scenario is used to estimate the maximum potential of our approach.

The results can be seen in Figures 2-4 and Tables 6-7. We can observe from these results that while the *potential* performance of a mutation-based oracle is (naturally) almost always higher than the actual performance of our method, the gap between the actual implementation of our approach and the ideal scenario is often quite small. In some cases, such as when using UFC-satisfying test inputs with the *Vertmax_Batch* system or at small oracle sizes on *Infusion_Mgr*, the difference in results is statistically insignificant. Thus we can conclude that while there is clearly room for improvement in oracle data selection methods, our approach appears to often be quite effective in terms of *absolute* performance.

Oracle Size	Decision					
	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	100.00%	3.54%	5.00%	0.00%*	0.00%*	13.79%
2	133.33%	8.70%	5.28%	7.55%	13.79%	3.35%
3	133.33%	11.24%	13.73%	12.50%	21.62%	9.00%
4	143.65%	14.29%	16.03%	16.67%	12.50%	11.59%
5	124.04%	15.76%	15.25%	15.38%	16.67%	12.61%
6	123.21%	17.24%	17.24%	10.71%	14.58%	15.69%
7	116.67%	18.64%	20.34%	11.27%	14.00%	18.80%
8	112.50%	17.24%	17.46%	13.89%	16.67%	21.67%
9	109.82%	17.95%	19.40%	14.86%	18.18%	21.55%
10	106.46%	18.87%	21.54%	14.81%	14.29%	21.43%
11	105.41%	17.65%				
12	109.26%	19.05%				
13	109.31%	19.18%				
14	115.00%	20.00%				
15		19.72%				
16		17.72%				
17		18.18%				
18		17.33%				

Oracle Size	MC/DC					
	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	66.67%	5.48%	2.08%	0.97%	0.00%*	3.23%
2	82.86%	6.41%	2.80%	0.99%	10.53%	7.14%
3	91.29%	6.69%	4.55%	2.04%*	15.91%	11.11%
4	85.71%	7.06%	3.67%	3.77%	16.00%	4.51%
5	100.00%	7.74%	4.07%	4.85%	12.96%	8.27%
6	97.37%	8.84%	5.41%	5.66%	14.29%	11.57%
7	94.87%	8.74%	6.14%	5.63%	15.52%	13.59%
8	100.00%	10.05%	6.19%	6.60%	16.13%	14.71%
9	95.65%	8.91%	6.55%	7.55%	20.00%	15.50%
10	92.15%	10.21%	7.02%	8.37%	21.67%	16.91%
11	88.89%	11.00%				
12	87.10%	11.00%				
13	87.10%	11.00%				
14	90.31%	11.59%				
15		11.43%				
16		10.90%				
17		11.21%				
18		11.21%				

TABLE 6
Median relative improvement in idealized performance over standard performance of mutation-based selection for structural coverage-satisfying tests

5.4 Impact of Coverage Criteria (Q3)

Our technique relies on pre-existing suites of test inputs. It follows that we would like to investigate the impact of varying the *type* of test suite on the effectiveness of oracle selection.

Intuitively, when examining the results of the oracle data sets generated using the two structural coverage criteria, using test suites satisfying the stronger criterion (MC/DC) should have a lower potential for improving the testing process via oracle selection, as the test inputs should do a better job of exercising the code. However, as shown in Figures 2-5 for each case example, the gap between the output-base and generated oracles for both decision and MC/DC test suites seems to be roughly the same. For example, for the *DWM_2* system, we can see that despite overall higher levels of fault finding when using the MC/DC test suites, the general relationships between the output-base baseline approach, our approach, and the idealized approach remain similar. We see a rapid increase in effectiveness for small oracles, followed by a decrease in the improvement of our approach versus the output-base baseline as we approach oracles of size 10 (corresponding to an output-only oracle), followed by a gradual increase in the improvement. In some cases, relative improvements are higher for decision coverage (*Latctl_Batch*) and in others they

Oracle Size	Random					
	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	8.89%	1.12%	2.06%	0.82%	12.50%	7.14%
2	20.26%	3.23%	1.96%	1.65%	10.00%	11.11%
3	12.61%	3.86%	3.33%	1.67%	26.97%	16.23%
4	9.74%	2.87%	3.90%	2.48%	28.57%	22.73%
5	9.67%	0.98%*	4.68%	2.48%	30.77%	25.00%
6	5.55%	2.78%	5.10%	2.48%	31.25%	31.91%
7	4.18%	4.41%	5.91%	2.48%	29.41%	38.39%
8	5.82%	5.31%	6.12%	2.48%	27.78%	41.38%
9	7.96%	6.28%	6.90%	2.48%	31.41%	42.86%
10	8.75%	7.11%	7.33%	2.48%	30.00%	36.36%
11	10.64%	6.96%				
12	12.56%	6.60%				
13	12.25%	6.06%				
14	12.37%	5.98%				
15		5.19%				
16		5.08%				
17		5.08%				
18		4.27%				

Oracle Size	UFC			
	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch
1	60.00%	3.37%	0.85%	0.83%*
2	72.08%	4.17%	1.64%	1.67%
3	76.92%	3.96%	1.65%	2.50%*
4	81.25%	3.88%	1.65%	3.33%*
5	79.47%	0.85%	1.64%	2.48%
6	82.29%	2.54%	1.64%	2.48%
7	89.18%	3.45%	0.82%	2.48%*
8	88.24%	4.31%	0.82%	2.48%*
9	94.12%	4.46%	0.82%	2.48%*
10	100.00%	5.22%	0.82%	2.48%*
11	100.00%	6.03%		
12	100.00%	6.03%		
13	103.85%	5.24%		
14	103.94%	5.19%		
15		5.13%		
16		4.68%		
17		4.22%		
19		3.36%		

TABLE 7

Median relative improvement in idealized performance over standard performance of mutation-based selection for random and UFC-satisfying tests

	Decision	MC/DC	UFC	Random
DWM_1	2.0	2.0	4.0	6.0
DWM_2	1.5	2.0	7.0	6.0
Vertmax_Batch	1.0	2.0	5.0	6.0
Latctl_Batch	1.0	2.0	5.0	6.0
Infusion_Mgr	3.0	3.0		6.0
Infusion_Mgr (real faults)	3.0	3.5		6.0
Alarms	2.0	2.0		6.0
Alarms (real faults)	2.0	3.0		6.0

TABLE 8

Median number of steps per test

are higher for MC/DC (*Vertmax_Batch*). Relative improvements even vary between oracle sizes—as can be seen on *Infusion_Mgr* and *Alarms*, where MC/DC-satisfying tests tend to lead to larger improvements than decision-satisfying tests at small oracle sizes, but decision-satisfying tests lead to larger improvements at larger sizes.

The results in Tables 3-5 and Figures 2-5—particularly those for DWM_2—reveal three key observations about the oracles generated using random test inputs. First, the oracles largely exhibit the same trends as the oracles generated for the structure-based test suites—a sharp rise at small sizes, performance comparable to the output-base oracle around middle sizes, and further gains at the end. Second—however, the improvements from using our method over an output-base oracle are more modest. Finally, On the Rockwell Collins

systems, *all oracle selection methods* achieve higher levels of fault finding over random tests than they do when applied to test inputs generated to satisfy structural coverage criteria.

Observations 2 and 3 can be partially explained by examining the *individual tests*. As seen in Table 8, the median length of each randomly-generated test—as measured in number of recorded test steps—is significantly longer than the length of the tests in the coverage satisfying tests. In fact, the random tests tend to be three to six times longer than their structure-based counterparts. This addresses the third observation in particular, because longer tests allow more time for corrupted internal states to propagate to the output variables. Tests generated to satisfy structural coverage criteria tend to focus on exercising particular syntactic elements of the source code, and thus, tend to be short—just long enough to exercise that particular obligation. As a result, tests generated to satisfy structural coverage obligations may be very effective at locating faults, but may not be long enough to propagate the fault to the output level.

The third observation is not true for *Infusion_Mgr* and *Alarms*—coverage-satisfying tests outperform randomly-generated tests. This is due to the complex structure of these systems. Deep exploration of their state spaces requires particular combinations of Boolean conditions, and random tests are less likely to hit some of these combinations. However, the second observation—that the gains are more modest—is still true. Even if the state space is less thoroughly explored, the longer test lengths still allow more time for the effects of the triggered faults to propagate to the output variables.

As shown in Figures 2 and 3—despite different fault finding numbers—the plots for test oracles that make use of test suites generated to satisfy the UFC requirements coverage criterion look very similar to the plots for the oracles that use other test input types. However, as with random tests, (1) *all oracle selection methods* tend to do very well, and (2), the improvement from using our method tends to be small.

There is some wisdom in the common approach of monitoring the output variables, as some faults will always be caught by monitoring them. Ultimately, the recommendation of whether or not to make use of our oracle data selection method is biased by the choice of test input—more specifically, *the probability of a fault propagating to the output variables*. As UFC coverage obligations tend to take the form of explicit relationships between inputs and outputs, it is unsurprising that our approach often fails to outperform the output-base oracle. While faults can still be masked in UFC tests, they are far more likely to propagate to an output variable than when running tests providing structural coverage. Therefore, it is unlikely that much improvement will be seen when working with test inputs that satisfy a requirements coverage criterion. On the other hand, when working with tests that explicitly exercise the internal structure of the software (as is mandated for certification for the avionics domain), masking of faults is common and our approach can significantly improve the effectiveness of the testing process. Regardless of the differing level of fault finding, the consistent trends across case examples exhibited for generated oracles indicates that, perhaps more that the test inputs used, characteristics of the system under test are the

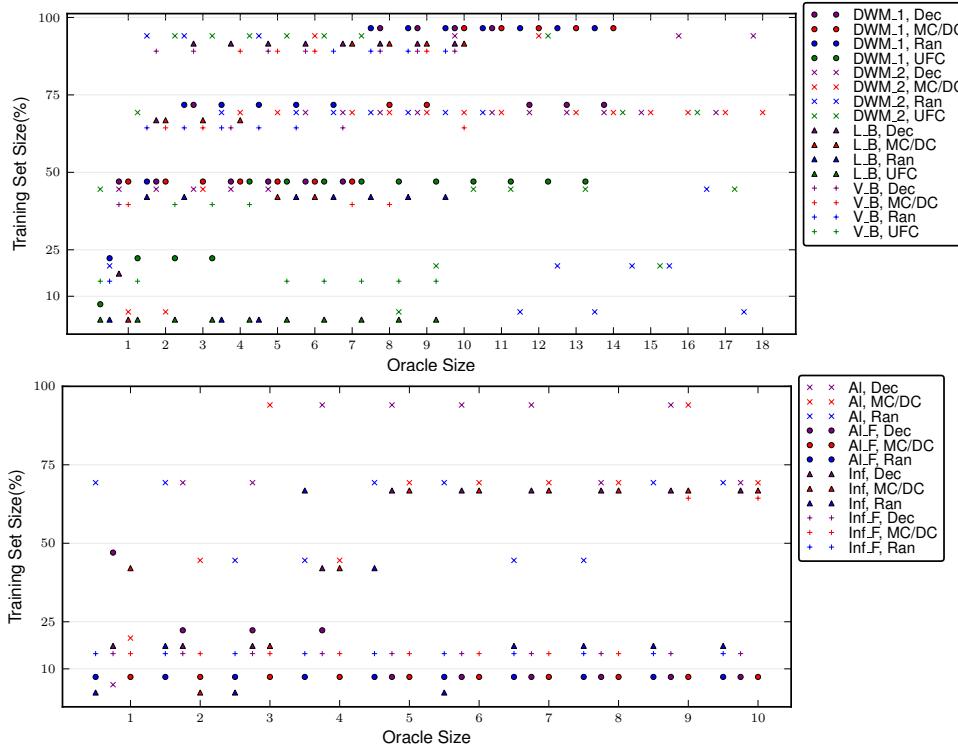


Fig. 6. Smallest Effective Training Sets ($V_B = Vertmax_Batch$, $L_B = Latctl_Batch$, $AI = Alarms$, $Inf = Infusion_Mgr$, $F = \text{version with real faults}$)

primary determinant of the effectiveness of our approach.

5.5 Impact of Number of Training Mutants (Q4)

For our initial experiment, we chose to use 125 mutants for training the oracle data. This number was chosen because earlier studies yielded evidence that results tend to stabilize after 100 mutants are used. That said, it may be possible to train sufficiently powerful oracles with fewer mutants—in fact, this would be an ideal situation, as examining fewer mutants will save time in the oracle generation process.

In Figure 6, we have summarized the results for statistical tests $H4 - H7$. We have omitted the full set of values, instead opting to determine the smallest training set size where we *cannot* reject the null hypothesis—the smallest training set that produces test oracles of effectiveness not statistically different from the full training set. The selected training set for each combination of case example, coverage criteria, and oracle size is plotted in the figure. (Note small vertical and horizontal offsets are present for readability. Points above $X\%$ and below $Y\%$ should be interpreted as training set size of exactly $Y\%$. Point above oracle size X and Y should be interpreted as oracle sizes of Y .) For example, for the DWM_1 system, training set sizes of 50% of those typically used provided statistically equivalent performance for oracle sizes between one to seven, while the full training set is required for oracles of sizes eight and larger.

Each point represents the plateau at which we cease to observe significant improvements from adding additional mutants to the training process. Based on the figure, we conclude that, for the most part, our initial estimate of 125 mutants was reasonable. For several of the combinations of case

example and test input, we fail to observe this plateau for several oracle sizes—we can say with statistical certainty that oracle effectiveness will diminish with fewer training mutants. These points typically correspond to areas where the ideal approach has a moderate (7–15%) and statistically significant difference over our approach, for example the DWM_1 system with random, MC/DC, and decision coverage for oracle sizes between 7–13. For these systems, our approach constructs test oracles that are effective, but can clearly be improved.

Often, however, smaller training set sizes are sufficient. We frequently observe a plateau in fault finding results after 75% or 50% of the training mutants are used to train the oracle (though *some* improvement is often seen when using the full training set). Thus, for the majority of combinations of test input type and case example, we can conclude that 62 to 100 mutants are needed to produce effective test oracles.

When using real faults as an evaluation criterion, diminishing returns are seen at smaller training set sizes—often at 25% for *Infusion_Mgr* and 10% for *Alarms*. As shown by our earlier results, ranking variables using information learned from seeded faults leads to improvements in effectiveness. However, when the mutation operators are dissimilar to the real mistakes made by the developers, we quickly exhaust what we can learn from the seeded faults. This again indicates that—while our mutation-based approach yields benefits—there is a need to expand the sources of the traces used to generate the oracle data set.

Although results vary between case example, it seems that one of the largest determinations of how many mutants are needed is the choice of test input type. Test suites generated to satisfy a structural coverage metric tend to require a large

	Test Type	Oracle Size	# Chosen Outputs	Total Outputs	# Chosen Internals	Total Internals
DWM_1	Decision	14	0.37	7	13.63	569
	MC/DC		0.43		13.57	
	UFC		0.84		13.16	
	Random		2.28		11.72	
DWM_2	Decision	18	3.10	9	14.90	115
	MC/DC		4.33		13.67	
	UFC		7.84		10.16	
	Random		8.30		9.70	
Vertmax_Batch	Decision	10	1.50	2	8.50	415
	MC/DC		1.51		8.49	
	UFC		2.00		8.00	
	Random		0.96		9.04	
Latctl_Batch	Decision	10	0.00	1	10.00	128
	MC/DC		0.00		10.00	
	UFC		1.00		9.00	
	Random		0.97		9.03	
Infusion_Mgr	Decision	10	1.88	5	8.12	107
	MC/DC		1.73		8.27	
	Random		2.72		7.28	
Infusion_Mgr(RF)	Decision	10	2.40	5	7.60	86
	MC/DC		2.29		7.71	
	Random		3.34		6.66	
Alarms	Decision	10	2.02	5	7.98	182
	MC/DC		2.27		7.73	
	Random		1.03		8.97	
Alarms(RF)	Decision	10	1.71	5	8.29	155
	MC/DC		1.54		8.46	
	Random		0.88		9.12	

TABLE 9

Average composition of generated oracles. RF = real faults.

number of mutants. In fact, when using suites designed to satisfy decision coverage, it may be possible to improve results further by adding *more mutants* than we did. Decision and MC/DC coverage obligations exercise pieces of the internal structure of a system, and thus, it is hard to predict exactly where a corrupt internal state will propagate to. The more mutants we examine, the more evidence there is for which specific internal variables we can observe to catch faults.

In contrast, UFC test obligations are expressed in terms of the relationship between inputs and outputs to the system under test, and thus, the tests are very likely to propagate faults to the output variables. As a side effect, our oracle selection method reaches a plateau very quickly. For two case examples, *Vertmax_Batch* and *Latctl_Batch*, as few as 10% of the training mutants used are necessary to reach stable conclusions. In fact, using more training mutants with test suites that satisfy UFC coverage can occasionally be slightly detrimental, as overfitting at larger training set sizes leads to worse median fault finding results at certain oracle sizes.

5.6 Oracle Composition (Q5)

In addition to performance, we are also interested in the *composition* of the generated data sets. Are they similar in structure to the current industrial practice of favoring the output variables, or, are they more heavily constructed from the many internal variables of the examined systems?

The average composition of the generated oracle data sets are listed in Table 9. For each system and test type, we list the size of the oracle data, the average number of chosen output variables, the total number of output variables, the average number of chosen internal variables, and the total number of internal variables. Recall that the oracle size was chosen to be the larger of twice the number of output variables or ten variables. Therefore, while a generated oracle data set could be

entirely composed of internal variables, none of the generated data sets will be composed entirely of output variables.

As with the performance of the generated oracle data, the composition seems to be largely determined by the type of test suite used to generate the data. The oracles created from structure-based test suites saw a large efficacy improvement from the oracle generation process because faults did not tend to propagate to the output variables in these tests. It follows, that the generated oracle data sets favor internal variables and largely ignore the output variables—for every system except *Vertmax_Batch*, fewer than half of the output variables are used in the oracles generated from structure-based tests. The UFC tests, on the other hand, are written specifically to propagate certain behaviors to the output variables, and the oracle data generated from the UFC tests tends to make use of more of those output variables. The composition of the oracle data sets generated from random tests varies depending on the case example—for the *Vertmax_Batch* system, oracle data generated from random tests only uses one of the two output variables. Similarly, for the *Alarms* system, the oracles generated from random tests made use of fewer output variables than those generated from structure-based tests. However, for the other systems, the oracle data sets generated from the random tests did more commonly choose output variables.

If the majority of the selected variables are internal variables, the effort cost of producing expected values for those variables must be considered. Not all variables can have their values specified with equal difficulty. Two key factors must be considered—*how often the value of the variable is checked* and *how the variable is used within the system*.

When and how often correctness is checked will help determine how effective the test oracle will be at catching faults. It is common to check behaviors after particular events or at regular points in time—for example, following the completion of a discrete computational cycle, as is the case with the systems used in our evaluation. A reasonable hypothesis might be that the more frequently values are checked, the easier it will be to spot problems. However, if expected values must be specified manually, there will be an increase in the required effort to produce the expected values for these result checks. Balancing the effort-to-volume ratio is important.

How a variable is used within the structure of the system may make specifying the value of those variables quite complicated. For example, if a variable is assigned a value in a triple-nested loop, unrolling the loops and selecting the value may be difficult. Similarly, an assignment requiring a complicated calculation might also require specifying the values of other dependent variables.

Therefore, even if our approach suggests a particular internal variable, testers may ignore the advice if specifying values for that variable is too difficult. In such situations, we recommend two courses of action—either remove difficult internal variables from consideration or weight those variables by the difficulty of specification.

Our approach starts with a candidate set of oracle variables and prunes it down into a recommended subset. While we have used all of the internal variables in our candidate set, that is not required. Testers could simply remove certain variables

from consideration before generating an oracle. This is the easiest solution to the problem of specifying expected values for more difficult internal variables.

However, that solution does carry a risk of causing the tester to miss out on valuable oracle information. Variables that are difficult to specify expected values for might be worth considering if they also correspond to valuable monitoring points in the system. Therefore, another option is that, before generating the oracle data set, the tester could go through the list of variables in the system and apply weights to some or all of them to represent the cost of producing expected output for that variable. Weights can easily be incorporated into the set-covering algorithm used to generate the oracle data—the weight can be factored against the expected fault-finding improvement from the use of that variable. Unless the improvement from using that variable is quite high, an expensive to test internal variable will not be chosen.

6 THREATS TO VALIDITY

External Validity: Our study is limited to six synchronous reactive critical systems. Nevertheless, we believe these systems are representative of the avionics systems in which we are interested and our results are therefore generalizable to other systems in the domain.

We have used Lustre [28] as our implementation language rather than a more common language such as C or C++. However, systems written in Lustre are similar to traditional imperative code produced in embedded systems development. A simple syntactic transformation suffices to translate Lustre code into C code.

We have generated approximately 250 mutants for each case example, with 125 mutants used for training sets and up to 125 mutants used for evaluation. These values are chosen to yield a reasonable cost for the study. It is possible the number of mutants is too low. Nevertheless, based on past experience, we have found results using less than 250 mutants to be representative [39].

Internal Validity: We have used the JKind model checker to generate test cases. This generation approach provides the shortest test cases that provide the desired coverage. It is possible that test cases produced through some other method would yield different oracle data sets.

Construct Validity: We measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software, for four of the examined systems. Given that our approach to selecting oracle data is also based on the mutation testing, it is possible that using real faults would lead to different results. As mentioned earlier, Andrews et al. and Just et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults in similar fault finding experiments [19], [20]. For the systems with real faults, our general results held.

Yao et al. have found that certain mutation operators can result in more equivalent mutants than other operators, thus skewing the results of testing experiments [40]. While we did remove equivalent mutants from the evaluation set (about 3% of the mutants for each system), in practice, these did not

disproportionately result from particular mutation operators. Therefore, we do not feel that our results could have been impacted by biasing in the mutation operators.

7 CONCLUSION

In this study, we have explored a mutation-based method for supporting oracle creation. Our approach automates the selection of oracle data, the set of variables monitored by the test oracle—a key component of expected value test oracles.

Experimental results indicate that our approach, when paired with test suites generated to satisfy structural coverage criteria or random tests, is successful with respect to alternative approaches for selecting oracle data, with improvements up to 1,435% over output-base oracle data selection and improvements of up to 50% relatively common. Even in cases where our approach is not more effective, it appears to be comparable to the common practice of monitoring the output variables. We have also found that our approach performs within an acceptable range from the expected maximum performance.

However, we also found that our approach was not effective when paired with test inputs generated to satisfy requirements-based metrics. These tests, expressed in terms of the relationships between inputs and outputs, are highly likely to propagate faults to the output variables, reducing the potential gains from selecting key internal states to monitor.

Thus, we recommend the use of our approach when test suites that exercise structures internal to the system under test are employed in the testing process (such test suites are required by standards in the avionics domain) [29].

REFERENCES

- [1] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “A comprehensive survey of trends in oracles for software testing,” Tech. Rep. CS-13-01, University of Sheffield, Department of Computer Science, 2013.
- [2] L. Briand, M. DiPenta, and Y. Labiche, “Assessing and improving state-based class testing: A series of experiments,” *IEEE Trans. on Software Engineering*, vol. 30 (11), 2004.
- [3] M. Staats, M. Whalen, and M. Heimdahl, “Better testing through oracle selection (nier track),” in *Proceedings of the 33rd Int'l Conf. on Software Engineering*, pp. 892–895, 2011.
- [4] M. Staats, M. Whalen, and M. Heimdahl, “Programs, testing, and oracles: The foundations of testing revisited,” in *Proceedings of the 33rd Int'l Conf. on Software Engineering*, pp. 391–400, 2011.
- [5] Q. Xie and A. Memon, “Designing and comparing automated test oracles for gui-based software applications,” *ACM Trans. on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, p. 4, 2007.
- [6] M. Staats, G. Gay, and M. Heimdahl, “Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing,” in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, 2012.
- [7] D. J. Richardson, S. L. Aha, and T. O’Malley, “Specification-based test oracles for reactive systems,” in *Proc. of the 14th Int'l Conf. on Software Engineering*, pp. 105–118, Springer, May 1992.
- [8] A. Memon and Q. Xie, “Using transient/persistent errors to develop automated test oracles for event-driven software,” in *Automated Software Engineering, 2004. Proceedings. 19th Int'l Conf. on*, pp. 186 –195, sept. 2004.
- [9] J. Voas and K. Miller, “Putting assertions in their place,” 1994.
- [10] C. Pacheco and M. Ernst, “Eclat: Automatic generation and classification of test inputs,” *ECOOP 2005-Object-Oriented Programming*, pp. 504–527, 2005.
- [11] K. Taneja and T. Xie, “Diffgen: Automated regression unit-test generation,” 2008.
- [12] R. Evans and A. Savoia, “Differential testing: a new approach to change detection,” 2007.

- [13] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," 2010.
- [14] G. Fraser and A. Zeller, "Generating parameterized unit tests," 2011.
- [15] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2013.
- [16] M. Harman, K. Sung, 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 *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third Int'l Conf. on*, pp. 182–191, 2010.
- [17] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, (New York, NY, USA), pp. 1–4, ACM, 2010.
- [18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, no. 99, p. 1, 2010.
- [19] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 608 –624, aug. 2006.
- [20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering*, 2014.
- [21] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *Proceedings of the 2001 Int'l Conf. on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, (Washington, DC, USA), pp. 161–172, IEEE Computer Society, 2001.
- [22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.
- [23] M. Garey and M. Johnson, *Computers and Intractability*. New York: Freeman, 1979.
- [24] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.
- [25] A. Murugesan, S. Rayadurgam, and M. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.
- [26] "Mathworks Inc. Simulink." <http://www.mathworks.com/products/simulink>, 2015.
- [27] "MathWorks Inc. Stateflow." <http://www.mathworks.com/stateflow>, 2015.
- [28] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [29] RTCA, *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
- [30] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.
- [31] M. Whalen, A. Rajan, M. Heimdahl, and S. Miller, "Coverage metrics for requirements-based testing," in *Proceedings of the Int'l Symposium on Software Testing and Analysis*, pp. 25–36, 2006.
- [32] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *Software Engineering Notes*, vol. 24, pp. 146–162, November 1999.
- [33] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pp. 83–91, IEEE Computer Society, April 2001.
- [34] G. Hagen, *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [35] A. Gacek, "JKind - a Java implementation of the KIND model checker." <https://github.com/agacek>, 2015.
- [36] M. Heimdahl, G. Devaraj, and R. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?," in *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*, (Tampa, Florida), March 2004.
- [37] C. Van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 814–819, 2002.
- [38] R. Fisher, *The Design of Experiment*. New York: Hafner, 1935.
- [39] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proc. of the 30th Int'l Conf. on Software engineering*, pp. 161–170, ACM, 2008.
- [40] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 919–930, ACM, 2014.



Gregory Gay is an Assistant Professor of Computer Science & Engineering at the University of South Carolina. His research interests include automated testing and analysis—with an emphasis on test oracle construction—and search-based software engineering. Greg received his Ph.D. from the University of Minnesota, working with the Critical Systems research group, and an M.S. from West Virginia University.



Matt Staats has worked as a research associate at the Software Verification and Validation lab at the University of Luxembourg and at the Korean Advanced Institute of Science and Technology in Daejeon, South Korea. He received his Ph.D. from the University of Minnesota-Twin Cities. Matt Staats' research interests are realistic automated software testing and empirical software engineering. He is currently employed by Google, Inc.



Michael Whalen is a Program Director at the University of Minnesota Software Engineering Center. Dr. Whalen is interested in formal analysis, language translation, testing, and requirements engineering. He has developed simulation, translation, testing, and formal analysis tools for Model-Based Development languages including Simulink, Stateflow, SCADE, and *RSML*–e, and has published more than 40 papers on these topics. He has led successful formal verification projects on large industrial avionics models, including displays (Rockwell-Collins ADGS-2100 Window Manager), redundancy management and control allocation (AFRL CerTA FCS program) and autoland (AFRL CerTA CPD program). He has recently been researching tools and techniques for scalable compositional analysis of system architectures.



Mats P.E. Heimdahl is a Full Professor of Computer Science and Engineering at the University of Minnesota, the Director of the University of Minnesota Software Engineering Center (UM-SEC), and the Director of Graduate Studies for the Master of Science in Software Engineering program. He earned an M.S. in Computer Science and Engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden and a Ph.D. in Information and Computer Science from the University of California at Irvine.

His research interests are in software engineering, safety critical systems, software safety, testing, requirements engineering, formal specification languages, and automated analysis of specifications.

He is the recipient of the NSF CAREER award, a McKnight Land-Grant Professorship, the McKnight Presidential Fellow award, and the awards for Outstanding Contributions to Post-Baccalaureate, Graduate, and Professional Education at the University of Minnesota.

The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage

GREGORY GAY, University of South Carolina

AJITHA RAJAN, University of Edinburgh

MATT STAATS, Google, Inc.

MICHAEL WHALEN and MATS P.E. HEIMDAHL, University of Minnesota

Test adequacy metrics defined over the structure of a program, such as Modified Condition and Decision Coverage (MC/DC), are used to assess testing efforts. Unfortunately, MC/DC can be “cheated” by restructuring a program to make it easier to achieve the desired coverage. This is concerning, given the importance of MC/DC in assessing the adequacy of test suites for critical systems domains. In this work, we have explored the impact of implementation structure on the efficacy of test suites satisfying the MC/DC criterion using four real world avionics systems.

Our results demonstrate that test suites achieving MC/DC over implementations with structurally complex Boolean expressions are generally larger and more effective than test suites achieving MC/DC over functionally equivalent, but structurally simpler, implementations. Additionally, we found that test suites generated over simpler implementations achieve significantly lower MC/DC and fault finding effectiveness when applied to complex implementations, whereas test suites generated over the complex implementation still achieve high MC/DC and attain high fault finding over the simpler implementation. By measuring MC/DC over simple implementations, we can significantly reduce the cost of testing, but in doing so we also reduce the effectiveness of the testing process. Thus, developers have an economic incentive to “cheat” the MC/DC criterion, but this cheating leads to negative consequences. Accordingly, we recommend organizations require MC/DC over a structurally complex implementation for testing purposes to avoid these consequences.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Testing Tools*

General Terms: Experimentation, Verification

Additional Key Words and Phrases: Coverage, Fault Finding

ACM Reference Format:

Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats P.E. Heimdahl. 2014. The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 0 (2014), 31 pages.

DOI :<http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

Test adequacy metrics defined over the structure of a program, such as statement coverage, branch coverage, and decision coverage, have been used for decades to assess the adequacy of test suites. Of particular note is Modified Condition and Decision Coverage (MC/DC) criterion [Chilenski and Miller 1994], as it is used as an exit criterion when testing highly critical software in the avionics industry. For certification of such software, a vendor must demonstrate that the test suite provides MC/DC of the source code [RTCA 1992].

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, NSF grants CNS-0931931 and CNS-1035715, and the L-3 Titan Group.

Author’s addresses: G. Gay, Department of Computer Science & Engineering, University of South Carolina. greg@greggay.com; A. Rajan, School of Informatics, University of Edinburgh. ajitha.rajan@gmail.com; M. Staats, Google. Inc. staatsm@gmail.com; M. Whalen and M.P.E. Heimdahl, Department of Computer Science & Engineering, University of Minnesota. [whalen,heimdahl]@cs.umn.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1049-331X/2014/-ART0 \$15.00

DOI :<http://dx.doi.org/10.1145/0000000.0000000>

Unfortunately, it is well known that structural coverage criteria, including MC/DC, can easily be “cheated” by restructuring a program to make it easier to achieve the desired coverage [Rajan et al. 2008]. This is concerning: a straightforward way to reduce the difficulty of achieving MC/DC over a program is to introduce additional variables to factor complex Boolean expressions into simpler expressions.

We have examined the effect of program structure transformations by comparing test suites necessary to cover programs consisting of simple decisions (consisting of at most one logical operator) versus programs in which more complex expressions are used. We refer to these versions as the non-inlined and inlined programs, respectively.

To understand the effect of program structure on the MC/DC criterion, we have focused on four dimensions—the cost to produce a MC/DC-satisfying test suite, the fault finding effectiveness of the produced suite, the ability of that suite to cover other structural representations of that system, and the effectiveness of that suite when applied to other representations. To that end, we have produced inlined and non-inlined implementations of four real-world avionics systems, generated test cases over those structures, generated hundreds to thousands of mutants, and used those mutants and various test oracles to assess fault finding when those tests are executed.

From our results, we can conclude that program structure has a dramatic effect on not only the coverage achieved, but also the cost and fault finding effectiveness of the testing process. Test suites achieving MC/DC over inlined implementations are generally larger than test suites achieving MC/DC over non-inlined implementations, requiring 125.00%-1,566.66% more tests. While it is clearly more expensive to achieve MC/DC over the inlined system, this effort yields fault finding improvements of up to 4,542.47%. Test suites generated over non-inlined implementations achieve significantly less MC/DC when applied to inlined implementations, attaining 13.08%-67.67% coverage on the more complex implementation. We also found that test suites generated over non-inlined implementations cannot be expected to yield effective fault finding on inlined implementations, finding 17.88-98.83% fewer faults than tests generated and executed on the inlined implementation. On the other hand, tests generated using the inlined implementation generally attained 100% MC/DC on the non-inlined system, and found up to 5,068.49% more faults than tests generated and executed on the non-inlined implementation.

Given that the inlined and non-inlined programs are semantically equivalent, the degree to which this simple transformation influences the effectiveness of the MC/DC criterion is cause for concern. We believe these results are concerning, particularly in the context of certification: we have demonstrated that by measuring MC/DC over simple implementations, we can significantly reduce the cost of testing, but in doing so we also reduce the effectiveness of testing. Thus developers have an economic incentive to “cheat” the MC/DC criterion (and by building tools similar to those used in our study, the means), but this cheating leads to negative consequences.

Based on these results, we strongly recommend that organizations using MC/DC metric in their software development efforts measure MC/DC on an inlined version of the system under test. While developers may choose to create non-inlined versions of the program with numerous intermediate variables for clarity or efficiency, such programs make it significantly easier to achieve MC/DC, with negative implications for testing effectiveness. In the absence of a metric that is robust to structural changes in the system under test, the use of tools, similar to those used in our study, can significantly improve the quality of the testing process.

This work is an expansion of two previous papers in which we studied the effect of program structure on *coverage* of MC/DC obligations [Rajan et al. 2008; Heimdahl et al. 2008]. In the first study, we examined how well a test suite generated to achieve maximum achievable coverage over a non-inlined system achieves coverage over an inlined system [Rajan et al. 2008]. Although this coverage result is important, the utility of a coverage metric is ultimately defined by how well test suites satisfying that metric are able to detect faults in the code. Thus, in the second study, we examined the effectiveness of test suites generated over inlined and non-inlined implementations on a set of seeded mutations [Heimdahl et al. 2008]. This report repeats those experiments using an improved experimental framework and updated definitions of the MC/DC obligations. We have

Version 1: Non-inlined Implementation

```
expr_1 = in_1 or in_2;      //stmt1
out_1 = expr_1 and in_3;    //stmt2
```

Version 2: Inlined Implementation

```
out_1 = (in_1 or in_2) and in_3;
```

Sample Test Sets for (in_1, in_2, in_3):

```
TestSet1 = { (TFF), (FTF), (FFT), (TTT) }
TestSet2 = { (TFT), (FTT), (FFT), (TFF) }
```

Fig. 1. Example of behaviorally equivalent implementations with different structures

conducted more rigorous studies, including: generating many more test suites for each system structure using more advanced test generation techniques, using most—if not all—mutants instead of a small random sampling, considering an additional type of test oracle—the largest common oracle—and performing more thorough statistical analyses. Our expanded and re-executed studies confirm the trends observed in the results of the previous work in a more rigorous fashion, and allow us to discuss the implications of our results in more detail. Additionally, we have added a fourth analysis to our experiment studying the effect of using one structure for test generation and another for test execution.

The remainder of the paper is organized as follows. In the next section, we provide background on MC/DC and problems related to program structure. Section 3 introduces our experimental setup and the case examples used in our investigation. Results and statistical analyses are presented in Section 4. In Section 5, we summarize our findings and their implications for practice. Section 6 discussed threats to validity. Finally, Section 7 discusses related work and we conclude in Section 8.

2. BACKGROUND AND MOTIVATION

A test suite provides MC/DC over the structure of a program or model if every condition within a decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision’s outcome (note here that when discussing MC/DC, a decision is defined to be an expression involving any Boolean operator).

Independent effect is defined in terms of *masking*, which means that the condition has no effect on the value of the decision as a whole. As an example, consider the trivial program fragments in Figure 1. The program fragments have different structures but are functionally equivalent. Version 1 is non-inlined with intermediate variable `expr_1`, and Version 2 is inlined with no intermediate variables. Given a decision of the form `in_1 or in_2`, the truth value of `in_1` is irrelevant if `in_2` is true, so we state that `in_1` is masked out. A condition that is not masked out has *independent effect* for the decision.

Based on the definition of MC/DC, `TestSet1` in Figure 1 provides MC/DC over program Version 1 but not over Version 2; the test cases with `in_3 = false` contribute towards MC/DC of the expression `in_1 or in_2` in Version 1 but not over Version 2 since the masking effect of `in_3 = false` is revealed in Version 2.

In contrast, MC/DC over the inlined version requires a test suite to take the masking effect of `in_3` into consideration as seen in `TestSet2`. This disparity in MC/DC over the two versions can have significant ramifications with respect to fault finding of test suites. Suppose the code fragment in Figure 1 is faulty and the correct expression should have been `in_1 and in_2` (which was erroneously coded as `in_1 or in_2`). `TestSet1` would be incapable of revealing this fault since there would be no change in the observable output, `out_1`. On the other hand, any test set providing MC/DC of the inlined implementation would be able to reveal this fault.

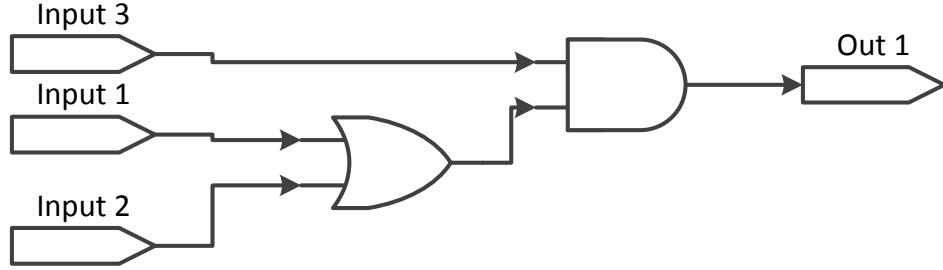


Fig. 2. Simulink model of example in Figure 1

Programs may be structured with significant numbers of intermediate variables for many reasons, for example, for clarity (nice program structure), efficiency (no need to recompute commonly used values), or to make it easier to achieve the desired MC/DC (MC/DC tests are easier to find if the decisions are simple). The programs may be restructured automatically via machine transformation, or they may be structured by the developers to improve program comprehension.

The potential problems with decision structure are not confined only to code. The move towards Model-based Development in the critical systems community makes test-adequacy measurement a crucial issue in the modeling domain. Coverage criteria such as MC/DC are being used in conjunction with modeling tools such as Simulink [Mathworks Inc. 2015] and SCADE [Esterel-Technologies 2004] for testing models. Currently, MC/DC measurement over models in these tools is being done in the weakest possible manner [Mathworks Documentation 2015]. For example, Figure 2 is a Simulink model equivalent to the example in Figure 1. MC/DC of such models is currently defined on a “gate level” (analogous to the MC/DC measurement over Version 1 in Figure 1). Since there are no complex decisions in this definition of MC/DC, MC/DC measured this way is susceptible to the masking problem discussed above, and test suites designed to provide MC/DC over the models may therefore provide poor fault finding capability. Thus, the current approach to measuring MC/DC over such models is cause for concern. For simplicity, in the remainder of this report, we refer to a “model” or “program” as the “implementation” since the concerns discussed here are the same regardless of whether we are discussing a model or a program.

Given the economic incentive to change implementation structure to ease certification, and the natural tendency of developers to simplify system structure, we believe understanding how implementation structure impacts the effectiveness of the MC/DC criterion is worthwhile. In particular, we are concerned that using a simpler implementation structure may be detrimental to the effectiveness of the testing process. In particular, we would like to know (1) how implementation structure impacts the coverage achieved by test suites built to satisfy MC/DC across differing implementation structures, and (2) how constructing test suites to satisfy MC/DC for different implementation structures impacts fault finding effectiveness.

3. STUDY OBJECTIVES AND DESIGN

To investigate how MC/DC is affected by the structure of an implementation, we explored three key research questions:

- **Question 1 (Q1):** How does the implementation structure impact the cost, as measured by the number of test inputs required, of achieving MC/DC?
- **Question 2 (Q2):** How does the implementation structure impact the effectiveness, as measured by the number of faults detected, of test suites achieving MC/DC?
- **Question 3 (Q3):** How well do test suites generated over a structurally simple implementation satisfy MC/DC for a semantically equivalent, but structurally complex implementation? Similarly, how well do test suites generated over the structurally complex implementation satisfy MC/DC for the simpler non-inlined implementation?

- **Question 4 (Q4):** How effective are test suites generated over a structurally simple implementation at detecting faults in a semantically equivalent, but structurally complex implementation? Similarly, how effective are test suites generated over the structurally complex implementation at detecting faults for the simpler non-inlined implementation?

The first two questions address how varying implementation structure impacts two key questions related to a test coverage criterion: *how much does it cost?*, and *how effective are tests suites that satisfy it?* Such questions are relevant when assessing the practical differences of cost and effectiveness that occur when varying the implementation structure. The third question is relevant in the context of MC/DC's role in certification of software. We would like to know if measuring coverage over a structurally simple implementation (e.g., as represented in Simulink) instead of more structurally complex implementation (e.g., as written by developers in C code) results in different measurements, as lower coverage results may imply inadequate test data. A related question is whether the source of the test suites impacts the effectiveness of those tests when those tests are executed against a different structure. If tests are generated using a structurally simple implementation, will they be effective at detecting faults in the structurally complex implementation?

Note that while the answers to these questions are likely to be related, as demonstrated by work exploring the impact of test suite size and structural coverage on fault finding effectiveness [Inozemtseva and Holmes 2014; Namin and Andrews 2009], a practically significant relationship is not guaranteed. It is trivial to construct small examples in which changing the structure has a strong impact on fault finding effectiveness, the number of test inputs required, etc. However, it is possible that—in practice—substantial differences in MC/DC correspond to only insignificant differences in fault finding. It is, therefore, important to quantify the degree to which implementation structure can impact the cost and effectiveness of test suites satisfying MC/DC in practice, as it is this information that should drive decision making.

3.1. Choice of MC/DC Criterion

Chilenski investigated three different notions of MC/DC [Chilenski 2001], namely: Unique-Cause MC/DC, Unique-Cause + Masking MC/DC, and Masking MC/DC. In this report we use *Masking* MC/DC [Hayhurst et al. 2001] to determine the independence of conditions within a Boolean expression. In Masking MC/DC, a basic condition is *masked* if varying its value cannot affect the outcome of a decision due to the structure of the decision and the value of other conditions. To satisfy masking MC/DC for a basic condition, we must have test states in which the condition is not masked and takes on both *true* and *false* values.

Unique-Cause MC/DC requires a unique cause—when a single condition is flipped, the result of the expression changes—to independent impact. This means that, when faced with strongly-coupled conditions in complex decision statements, there may be situations where no test can establish a unique cause. Unique Cause + Masking MC/DC relaxes the unique cause requirement to all uncoupled conditions, and Masking MC/DC allows masking in all cases. Masking MC/DC is the easiest of the three forms of MC/DC to satisfy since it allows for more independence pairs per condition and more coverage test sets per expression than the other forms. In contrast, Chilenski's analysis showed that even though Masking MC/DC could allow fewer tests than Unique-Cause MC/DC, its performance in terms of probability of error detection was nearly identical to the other forms of MC/DC. This led Chilenski to conclude in [Chilenski 2001] that Masking MC/DC should be the preferred form of MC/DC.

To better illustrate the definition of masking MC/DC, consider the expression *A and B*. To show the independence of *B*, we must hold the value of *A* to *true*; otherwise varying *B* will not affect the outcome of the expression. Independence of *A* is shown in a similar manner. Table I shows the test suite required to satisfy MC/DC for the expression *A and B*. When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked out by the behavior of other operators. For example, given *A or (B and C)* the tests for *B and C* will not affect the outcome of the decision if *A* is *true*. Table II gives one of the test suites that would

Table I. Example of a test suite that provides Masking MC/DC over A and B

A	B	A and B
T	T	T
T	F	F
F	T	F

satisfy masking MC/DC for the expression A or (B and C). Note that in Table II, test cases {2,4} or {3,4} will demonstrate independence of condition A under Masking MC/DC. However, these test cases will not be sufficient to show independence of A under Unique Cause MC/DC since the values for conditions B and C are not fixed between the test cases.

Table II. Example of a test suite that provides Masking MC/DC over A or (B and C)

A	B	C	A or (B and C)
F	T	T	T
F	T	F	F
F	F	T	F
T	F	F	T

3.2. Experimental Design

In our study, we performed the following steps for each case example:

- **Generated inlined and non-inlined implementation versions:** We transformed the implementation structure, creating one version with a high degree of expression complexity (inlined version) and one version with a low degree of expression complexity (non-inlined version). This is detailed in Section 3.4. Note that unless specified, the remaining steps are applied to both the inlined and non-inlined version separately (e.g., separate mutant sets are generated for the inlined and non-inlined versions).
- **Generated mutants:** We generated the full pool of mutants, each containing a single fault, for each system. We then removed functionally equivalent mutants. (Section 3.5.)
- **Generated tests satisfying the MC/DC criterion:** We generated a set of tests satisfying the MC/DC criterion. The JKInd model checker [Hagen 2008; Gacek 2015] was used to generate each set of tests, resulting in a test set with one test case for each obligation. (Section 3.6.)
- **Generated reduced test suites:** We generated 100 reduced test suites using a greedy reduction algorithm. Each reduced test suite maintains the coverage provided by the full test suite. (Section 3.6.)
- **Selected oracles:** We used two test oracles in our study: an output-only oracle considering all outputs, and a maximum oracle considering all internal variables and all outputs. (Section 3.7.)
- **Ran tests on mutants:** We ran each mutant and the original case example using every test generated, and collected the internal state and output variable values produced at every step. This yields raw data used for assessing fault finding in our study. (Section 3.8.)
- **Assessed fault finding ability of each oracle and test suite combination:** We determined how many mutants were detected by every oracle and a reduced test suite combination. (Section 3.8.)

We generated test obligations, extracted concrete test cases from JKInd counter-examples, and executed the tests on the mutants using an in-house suite of tools designed for testing models written in the Lustre synchronous language. This framework is open-source and freely available from <https://github.com/djyou/lustre>.

3.3. Case Examples

We used four industrial synchronous reactive systems developed by Rockwell Collins Inc. in our experiment. Information related to these systems is provided in Table III. We list the number of Simulink subsystems, which are analogous to functions, and the number of blocks, analogous to operators. We also list the number of calculated expressions in the Lustre implementations.

Table III. Case example information. NI = non-inlined, I = inlined

	# Simulink Subsystems	# Blocks	Lustre Expressions NI	Lustre Expressions I
DWM_1	3,109	11,439	576	28
DWM_2	128	429	121	19
Vertmax_Batch	396	1,453	417	31
Latctl_Batch	120	718	129	20

All four systems were modeled using the Simulink notation from Mathworks Inc. [Mathworks Inc. 2015] and were translated to the Lustre synchronous programming language [Halbwachs 1993] to take advantage of existing automation. This is analogous to the automated code generation done from Simulink using Real Time Workshop from Mathworks. In practice, Lustre would be automatically translated to C code. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

3.3.1. Flight Guidance System. A Flight Guidance System (FGS) in an aircraft compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between those two states. The FGS consists of the mode logic, which determines what lateral and vertical modes of operation are active at any given time, and the flight control laws that are used to compute the pitch and roll guidance commands. The two FGS systems used in this work, *Vertmax_Batch* and *Latctl_Batch*, describe the vertical and lateral mode logic for the Rockwell Collins FCS 5000 flight guidance system family.

3.3.2. Display Window Manager. The Display Window Manager (DWM) models, *DWM_1* and *DWM_2*, represent two of the five major subsystems of the DWM of the Rockwell Collins ADGS-2100, an air transport-level commercial display system. The DWM acts as a “switchboard” for the system, routing information to the aircraft displays and managing the location of two cursors that can be used to control applications by the pilot and copilot. The DWM must update the applications being displayed in response to user selections and must handle reversion in case of hardware application failures. The DWM systems decide what information is most critical and moves this information to the remaining displays.

3.4. Implementation Structure

For each case example, we generate two versions that are semantically equivalent, but syntactically different. We term these the *inlined* and *non-inlined* versions of the implementation, described below.

3.4.1. Non-inlined Implementation Structure. In a non-inlined implementation, the structure of the implementation is similar to the structure of the original Simulink model. Each signal from the Simulink model has been preserved, resulting in a very large number of internal state variables, with each internal state variable corresponding to a relatively simple expression. A small example of a non-inlined implementation is given in Figure 3.

Unlike our previous studies exploring program structure [Rajan et al. 2008; Heimdahl et al. 2008], in this study, the structure is flattened such that the implementation has only one Lustre node. In Lustre, a node is a subprogram—a single callable block of code. If the model consists of multiple nodes, all of the nodes are combined into a single node (i.e., the code of external functions is imported into the calling block). This node flattening is conducted for compatibility purposes with experimental infrastructure and does not impact the results of the study.

```

node exampleProgramNode(input_1: bool;
    input_2: bool;
    input_3: int;
    input_4: bool)
    returns (output_1: bool);
var
    internal_1: bool;
    internal_2: bool;
    internal_3: bool;
    internal_4: bool;
    internal_5: bool;
    internal_6: bool;
let
    internal_1 = input_1 AND input_2;
    internal_2 = input_3 > 100;
    internal_3 = internal_1 OR input_4
    internal_4 = IF (internal_3) THEN internal_2 ELSE input_1;
    internal_5 = input_3 > 50;
    internal_6 = IF (internal_5) THEN internal_4 ELSE input_2;
    output_1 = internal_6;
tel;

```

Fig. 3. Example non-inlined implementation.

3.4.2. Inlined Implementation Structure. As with the non-inlined implementation, in the inlined implementation, the structure is first flattened such that the implementation has only one Lustre node. Once this has been completed, we then inline most, but not all, of the intermediate variables into the model. When *Inlining* a variable, we substitute the expression corresponding to the variable wherever the variable is referenced, and then remove the variable from the implementation (as it is no longer referenced). This has the effect of (1) reducing the number of internal state variables, as inlined variables are removed from the model, (2) increasing the complexity of expressions, as inlined variables are substituted wherever they are referenced, and (3) increasing the number of nested if-then-else expressions, as such expressions are often substituted into then and else branches.

MC/DC is defined exclusively over traditional imperative structures (such as those found in C, Java, etc.); we therefore restrict our inlining to prevent syntactic constructs impossible in imperative implementations from arising. In particular, we do not place if-then-else expressions inside if conditions. In Lustre, a statement of the form `result = if (if (internal_1 and internal_2) then internal_3 else internal_4) then true else false` is a valid expression. In an imperative language, the if condition would need to be contained in a separately evaluated expression, or through the use of the ternary operator. MC/DC, as traditionally defined, does not have a definition for the ternary operator [Hayhurst et al. 2001]. While such a definition could be added, we have chosen to stay within the MC/DC definitions commonly used for imperative languages. Therefore, our use of if-then-else expressions in Lustre corresponds directly with MC/DC-compliant if-then-else statements in C.

In Figure 4, we present an inlined version of the implementation from Figure 3. As we can see, the inlined version has been reduced from five internal state variables to zero, with the set of expressions condensed to one, considerably more complex, expression. This expression illustrates how inlining can increase complexity both in terms of Boolean/relational expressions and nesting of if-then-else statements. For example, we see that the condition formerly represented by `internal_3` has been inlined, and the if-then-else expression formerly represented by `internal_4` has been nested inside another if-then-else statement. As we will see shortly,

```

node exampleProgramNode(input_1: bool;
    input_2: bool;
    input_3: int;
    input_4: bool)
    returns (output_1: bool);
let
    output_1 =
        IF (input_3 > 50)
        THEN
            IF ((input_1 AND input_2) OR input_4)
            THEN input_3 > 100
            ELSE input_1 ELSE input_2;
tel;

```

Fig. 4. Example inlined implementation.

while these transformations result in semantically equivalent implementations, their impact on testing is significant.

3.5. Mutant Generation

We have created *mutations* (faulty implementations) of each case example by automatically introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. The type of faults used to create mutants may impact the effectiveness of the selected oracle data when used to test the actual system under test. Note that the type of mutants used in the evaluation in this report are similar to those used by Andrews et al.—where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [Andrews et al. 2006]—and similar to those used by Just et al., who found a significant correlation between mutant detection and real fault detection [Just et al. 2014].

We seed the following classes of faults:

- **Arithmetic:** Changes an arithmetic operator (+, -, /, *, mod, exp).
- **Relational:** Changes a relational operator (=, ≠, <, >, ≤, ≥).
- **Boolean:** Changes a Boolean operator (∨, ∧, XOR).
- **Negation:** Introduces the Boolean operator \neg .
- **Delay:** Introduces the delay operator on a variable reference (that is, uses the stored value of the variable from the previous computational cycle rather than the newly computed value).
- **Constant:** Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating Boolean constants.
- **Variable Replacement:** Substitutes a variable occurring in an equation with another variable of the same type.

For each system, we generated all mutants that could possibly be created using the selected mutation operators. The number and types of mutants are listed in Table IV. One risk of mutation testing is *functionally equivalent* mutants, in which faults exist but these faults cannot cause a *failure*, which is an externally visible deviation from correct behavior. For our study, we used model checking to detect and remove functionally equivalent mutants. This is made possible due to our use of synchronous reactive systems as case examples—each system is finite, and thus determining equivalence is decidable. (Equivalence checking is fairly routine on the hardware side of the synchronous reactive system community; Van Eijk provides a good introduction [Van Eijk 2002].) Thus for every mutant used in our study, there exists at least one trace that can lead to a user-visible failure, and all fault finding measurements indeed measure actual faults detected.

In order to make fair fault finding comparisons between implementations, we used *min(noninlined,inlined)* mutants in our experiments. For the implementation where we did not use

Table IV. Mutant information for each case example. NI = non-inlined, I = inlined

		DWM_1		DWM_2		Vertmax_Batch		Latctl_Batch	
		NI	I	NI	I	NI	I	NI	I
Arithmetic	Total Mutants	8,489	13,303	931	1,858	4,411	6,712	1,036	954
	Equivalent Mutants	344	919	15	61	152	182	40	42
	Total Used	8,145	8,145	916	916	4,259	4,259	912	912
Relational	Total	8	277	0	0	0	0	0	0
	Equivalent	0	19	0	0	0	0	0	0
	Used	8	170	0	0	0	0	0	0
Boolean	Total	691	792	14	73	15	231	5	36
	Equivalent	83	99	0	1	1	30	1	16
	Used	608	458	14	37	14	131	4	20
Negation	Total	11	111	44	189	452	861	64	85
	Equivalent	0	25	0	9	99	95	4	6
	Used	11	57	44	92	353	500	55	79
Delay	Total	1,041	2,004	290	574	1,443	2,187	333	303
	Equivalent	18	81	7	18	17	20	14	9
	Used	1,023	1,270	283	283	1,426	1,413	292	294
Constant	Total	3,466	5,081	337	671	1,511	2,432	373	352
	Equivalent	21	128	0	22	1	15	8	3
	Used	3,445	3,270	337	331	1,510	1,576	334	349
Replacement	Total	1,522	3,167	54	112	84	300	41	84
	Equivalent	149	395	7	0	16	17	7	7
	Used	1,373	1,830	47	57	68	185	31	77
	Total	1,750	1,824	192	239	906	701	220	94
	Equivalent	73	172	1	11	18	5	6	1
	Used	1,677	1,091	191	116	888	454	196	93

the full pool of mutants, we selected mutants so that the *fault ratio* for each fault class was approximately uniform. That is, assume for some example, there are R possible Relational faults and B possible Boolean faults. For a uniform fault ratio, we would seed x relational faults and y Boolean faults in the implementation so that $x/R = y/B$. For example, if there are 88 possible Boolean faults and 12 possible relational faults, and we wanted to select 25 mutants, we could select 0.25 as our fault ratio and our resulting set would contain 22 Boolean faults and 3 relational faults. This uniform ratio means that—despite setting a controlled number of mutants—we do not bias the distribution of fault types used in our experiments. The number of mutants for each type of fault reflects the distribution if all possible mutants were used.

3.6. Coverage Directed Test Input Generation

There exist several methods of generating tests to satisfy coverage criteria. We adopt counterexample-based test generation approaches based on existing model checking approaches to generate tests satisfying the MC/DC criterion [Gargantini and Heitmeyer 1999; Rayadurgam and Heimdahl 2001]. We have used the JKInd model checker in our experiments [Hagen 2008; Gacek 2015].

We performed the following steps separately using the inlined and non-inlined case examples:

- (1) We processed the Lustre source code to generate *coverage obligations* for MC/DC. Each coverage obligation represents a property that should be satisfied by some test (e.g., some branch covered, some condition evaluates to true).
- (2) These obligations were inserted into the Lustre source code as *trap properties*—that is, the negation of the properties [Gargantini and Heitmeyer 1999]. Essentially, by asserting that these obligations can *never* be made true, the model checker can produce a counterexample showing how the property can be met.
- (3) We ran the model and properties through JKInd. This produces a list of counterexamples, each of which corresponds to the satisfaction of some coverage obligation.
- (4) Each counterexample is translated into a test input.

Table V. Unreduced test suite measurements. NI = non-inlined, I = inlined.

	NI # of MC/DC Obligations	% NI MC/DC Obligations Achievable	I# of MC/DC Obligations	% I MC/DC Obligations Achievable
DWM_1	2038	100.00%	3806	98.74%
DWM_2	530	98.68%	2192	100.00%
Vertmax_Batch	1732	100.00%	1858	96.99%
Latctl_Batch	380	100.00%	260	99.62%

Note that some coverage obligations are unsatisfiable, i.e., there does not exist a test that satisfies the coverage obligation. (This can occur, for example, if infeasible combinations of conditions are required by some coverage obligation.) Nevertheless, by using this approach, we ensure that the maximum number of satisfiable obligations are satisfied. A listing of the number of obligations and the percent of achievable coverage can be found in Table V.

This approach generates a separate test for each coverage obligation. While a simple method of generating tests, in practice, this results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. For example, in branch coverage, a test satisfying an obligation for a nested branch will also satisfy obligations for outer branches. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to satisfy the coverage criterion. Given the correlation between test suite size and fault finding effectiveness [Namin and Andrews 2009], this has the potential to yield misleading results—unnecessarily large test suites may lead us to conclude that a coverage criterion yields an effective test suite, when, in reality, it is the size of the test suite that is responsible for the effectiveness.

To avoid this, we reduce each naive test suite generated while maintaining the coverage achieved. This reduction is done using a simple greedy algorithm following these steps:

- (1) Each test is executed, and the coverage obligations satisfied by each test are recorded. This provides the coverage information used in subsequent steps.
- (2) We initialize two empty sets: *obs* contains satisfied obligations and *reduced* contains our reduced test suite. We initialize a set *tests* containing all tests from our naive test suite.
- (3) We randomly select and remove a test *t* from *tests*. If the test satisfies an obligation not currently in *obs*, we add it to *reduced*, and add the obligations satisfied by *t* to *obs*. Otherwise we discard the test.
- (4) We repeat the previous step until *tests* is empty or all obligations have been satisfied.

Due to the randomization, the size and contents of reduced suites may vary quite a bit between suite generations. For example, due to the random test selection, this reduction approach may choose ten tests that each cover a single new obligation when there may exist a single test that covers all ten. Thus, while this reduction approach results in smaller test suites, it is not guaranteed to produce the smallest possible test suites. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the set of possible reduced test suites, we produce 100 randomly reduced test suites using this process.

3.7. Test Oracles

In our study, we use what are known as *expected value oracles* as our test oracles [Staats et al. 2012a]. Consider the following testing process for a software system: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgment; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. Past experience with industrial practitioners indicates that such oracles are commonly used in testing critical systems, such as those in the avionics or medical device fields.

We explore the use of two types of oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice, whereas the

maximum oracle represents an idealistic scenario where we can monitor each expression assignment in the system. In practice, the maximum oracle is often prohibitively expensive to specify. However, it offers the ability to assess the effect of oracle selection on fault finding [Gay et al. 2015a; Staats et al. 2012a].

Table VI. Oracle sizes for each case example and implementation

	Implementation	# Output Variables	Total # Variables
DWM.1	Inlined	7	28
	Non-inlined	7	576
DWM.2	Inlined	9	19
	Non-inlined	9	124
Vertmax.Batch	Inlined	2	32
	Non-inlined	2	417
Latctl.Batch	Inlined	1	20
	Non-inlined	1	129

The number of variables in each oracle for the inlined and non-inlined implementations can be seen in Table VI. Regardless of implementation, the number of variables in the output-only oracle will remain the same. However, as a result of the inlining process, the number of expressions in the inlined implementation will be far smaller than in the simple non-inlined implementation. Thus, the size of the maximum oracle will be far smaller for the inlined implementation. We will explore the implications of oracle size in the following section.

3.8. Data Collection

After generating the full test suites and mutant set for a given case example, we ran each test suite against every mutant and the original case example. For each run of the test suite, we recorded the value of every internal variable and output at each step of every test using the Lustre interpreter in our test generation framework. This process yielded a complete set of values produced by running the test suite against every mutant and the correct implementation for each case example.

To determine the fault finding of a test suite t and oracle o for a case example we simply compare the values produced by the original case example against every mutant using (1) the subset of the full test suite corresponding to the test suite t and (2) the subset of variables corresponding to the oracle data for oracle o . The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or “killed”) divided by the total number of non-equivalent mutants used (see Table IV). We perform this analysis for each oracle and test suite for every case example yielding a very large number of measurements per case example. We use the information produced by this analysis in later sections to explore our research questions.

4. RESULTS

For each case example, we began with a large test suite that achieved the maximum achievable fault finding for the criterion. We then subsequently used a randomized test suite reduction algorithm to generate 100 reduced test suites. Finally, we computed the size and fault finding of each resulting test suite on our pool of non-equivalent mutants using both test oracles.

Using these numbers, we measured the following:

- **Original Number of Obligations:** The number of MC/DC obligations. This is usually slightly larger than the size of the original unreduced test suite, as unachievable obligations have no corresponding test input.
- **Percent Achievable Obligations:** The percent of MC/DC obligations that can be covered. This coverage is maintained during test suite reduction.
- **Median Fault Finding:** Median fault finding across reduced test suites.
- **Median Test Suite Size:** Median test suite size across reduced test suites.

Table VII. Median (μ) reduced test suite size across case example.

	Non-inlined μ Size	Inlined μ Size	% Size Increase
DWM_1	27.00	450.00	1,566.66%
DWM_2	69.00	250.00	262.32%
Vertmax.Batch	68.00	334.00	391.18%
Latctl.Batch	28.00	63.00	125.00%

Table VIII. Median (μ) fault finding measurements across case example. FF = fault finding, OO = output-only oracle, MX = maximum oracle

	Non-inlined μ FF (OO)	Inlined μ FF (OO)	% FF Increase (OO)
DWM_1	1.46%	67.78%	4,542.47%
DWM_2	85.04%	88.81%	4.43%
Vertmax.Batch	51.22%	85.60%	67.12%
Latctl.Batch	61.02%	74.94%	22.81%

	Non-inlined μ FF (MX)	Inlined μ FF (MX)	% FF Increase (MX)
DWM_1	72.28%	68.01%	-6.28%
DWM_2	94.54%	88.97%	-5.89%
Vertmax.Batch	92.32%	85.68%	-7.19%
Latctl.Batch	92.33%	78.07%	-15.44%

- **Relative Change in Median Test Suite Size:** We measure the relative increase (as a percentage) in median test suite size when generating tests using the inlined implementation as compared to when generating tests using the non-inlined implementation. Positive percentages indicate that the median test suite size is larger for the inlined implementation.
- **Relative Change in Median Fault Finding:** We measure the relative increase (as a percentage) in median test suite fault finding effectiveness when generating tests using the inlined implementation as compared to when generating tests using the non-inlined implementation. Positive percentages indicate that the median effectiveness is larger when generating test suites using the inlined implementation.
- **Median Coverage of Non-inlined Suites over Inlined Implementation:** We measure the median MC/DC achieved when using test suites generated from the non-inlined system over the inlined implementation. 100% indicates no change; the test suite achieves 100% of the achievable MC/DC over both systems.
- **Median Coverage of Inlined Suites over Non-inlined Implementation:** We measure the median MC/DC achieved when using test suites generated from the inlined system over the non-inlined implementation.
- **Median Fault Finding of Non-inlined Suites over Inlined Implementation:** We measure the median fault finding across reduced test suites generated from the non-inlined implementation over the inlined implementation.
- **Median Fault Finding of Inlined Suites over Non-inlined Implementation:** We measure the median fault finding across reduced test suites generated from the inlined implementation over the non-inlined implementation.

The measurements for unreduced test suites are given in Table V, and the measurements for reduced test suites are given in Tables VII (test suite size), VIII and IX (fault finding), X (coverage across implementation types), and XI (fault finding across implementation types).

We can see four patterns. First, per *Q1* (see Table VII), the number of test inputs required to satisfy MC/DC over the inlined implementation is often significantly higher than the number of test inputs required to satisfy MC/DC over the non-inlined implementation, with increases of 125.00%–1,566.66%. Thus, the cost of satisfying MC/DC, in terms of the number of test inputs we must create and execute, tends to increase as the implementation structure's complexity increases.

Second, per *Q2* (see Table VIII), the effectiveness of test suites achieving maximum MC/DC also varies with implementation structure. When making use of the common output-only oracle, test suites satisfying MC/DC over the inlined structure outperform those satisfying MC/DC over

Table IX. Measurements across case example for the largest oracle common to both implementations. μ = median, FF = fault finding, LCO = largest common oracle

	Non-inlined μ FF (LCO)	Inlined μ FF (LCO)	% FF Increase (LCO)
DWM_1	5.27%	68.01%	1,190.51%
DWM_2	85.26%	88.97%	4.35%
Vertmax_Batch	53.48%	85.68%	60.21%
Latctl_Batch	67.00%	78.07%	16.52%

the non-inlined structure, with relative differences ranging from a slight increase (4.43%) to a large increase (4,542.47%) in median fault finding.

Fault finding results when making use of the maximum oracle demonstrate the opposite effect—test suites that satisfy MC/DC over the non-inlined structure slightly outperform those that satisfy MC/DC over the inlined structure, finding between 5.89 and 15.44% fewer mutants. However, this can easily be explained by the size of the maximum oracles for each implementation, as noted in Table VI. Because of the inlining process, the maximum oracles for the inlined systems are far smaller than those for the non-inlined systems. The inlined systems calculate and store the results of a smaller number of expressions. These additional points of observation give the non-inlined implementation a clear advantage in observing faults when using an oracle that checks the behavior of all possible observation points.

In practice, however, using such a large oracle would be prohibitively expensive, as the tester would have to explicitly specify the expected behavior for each variable. Even if they did that, the monitoring overhead from observing all of those variables may negatively impact the behavior of the system. If testing software on an embedded system, a tester may not even be able to observe all of the internal variables. Thus, although the maximum oracle for non-inlined systems does yield effective fault finding potential, its use may not be practical.

As the non-inlined implementation always contains more variables than the inlined implementation, the non-inlined implementation can always have an oracle that checks the behavior at more points of observation than the inlined implementation. A more equal comparison would be to compare the fault finding effectiveness of tests generated from both implementations with the largest possible fixed-sized oracle with variables that appear in both implementations (dubbed the **largest common oracle**). If we use the maximum oracle for the inlined implementation to compare fault finding effectiveness, we can increase the number of observation points while assessing the fault finding capability of the generated tests on a more equivalent basis. The results for the largest common oracle can be seen in Table IX.

The effectiveness of test suites achieving maximum MC/DC paired with the largest common oracle is similar to that using the output-only oracle, with relative differences ranging from a 4.35 to 1,190.51% improvement in median fault finding when using tests generated over the inlined implementation. While the improvement in fault finding is more subdued than when using the output-only oracle, we can still conclude that differences in fault finding roughly correspond to the previously noted differences in test suite cost, with larger—but more effective—test suites resulting from the use of more complex decisions in the implementations.

Third, per *Q3* (see Table X), we can see that test suites achieving 100% achievable MC/DC of the non-inlined implementations always achieve less than 100% achievable coverage over the inlined implementations, with coverage ranging from 13.08% to 67.67%. The reverse is not true, test suites that achieve 100% achievable MC/DC of the inlined implementation generally achieve close to, if not, 100% MC/DC of the non-inlined implementation. This highlights that not only does the implementation structure impact the cost and effectiveness of test suites satisfying the MC/DC criterion, it also can have a strong impact on the coverage achieved. We cannot expect to satisfy MC/DC on a structurally simple implementation, and still achieve 100% coverage (or even similar coverage levels) on a semantically equivalent—but more structurally complex—implementation.

Finally, to address *Q4*, we have examined the effect of implementation structure on test effectiveness in more detail. We have executed tests generated over the non-inlined implementation on the

Table X. Median (μ) coverage results across implementation type

	μ % Achievable MC/DC of Non-inlined Suites over Inlined Implementation
DWM_1	13.08%
DWM_2	53.25%
Vertmax_Batch	31.95%
Latctl_Batch	67.67%
	μ % Achievable MC/DC of Inlined Suites over Non-inlined Implementation
DWM_1	100.00%
DWM_2	100.00%
Vertmax_Batch	100.00%
Latctl_Batch	99.31%

Table XI. Median (μ) fault finding measurements across implementation type. I-on-NI means that tests are generated using the inlined implementation and executed on the non-inlined implementation. NI-on-I means that tests were generated using the non-inlined implementation and executed on the inlined implementation. FF = fault finding, OO = output-only oracle, MX = maximum oracle, LCO = largest common oracle, Incr. = increase.

	I-on-NI			NI-on-I		
	I-on-NI μ FF (OO)	% FF Incr. NI-on-NI	% FF Incr. I-on-I	NI-on-I μ FF (OO)	% FF Incr. I-on-I	% FF Incr. NI-on-NI
DWM_1	75.46%	5,068.49%	11.33%	0.79%	-98.83%	-45.89%
DWM_2	90.50%	6.40%	1.90%	72.93%	-17.88%	-14.24%
Vertmax_Batch	90.23%	76.16%	5.41%	29.37%	-65.69%	-42.66%
Latctl_Batch	82.57%	35.48%	10.18%	48.58%	-25.17%	-20.39%
	I-on-NI μ FF (MX)	% FF Incr. NI-on-NI	% FF Incr. I-on-I	NI-on-I μ FF (MX)	% FF Incr. I-on-I	% FF Incr. NI-on-NI
				NI-on-I μ FF (MX)		
DWM_1	97.29%	34.60%	43.05%	2.74%	-95.97%	-96.21%
DWM_2	92.80%	-1.80%	4.30%	72.93%	-18.03%	-22.86%
Vertmax_Batch	96.90%	4.96%	13.10%	100.00%	16.71%	8.32%
Latctl_Batch	98.47%	6.65%	26.13%	55.37%	-29.07%	-40.03%
	I-on-NI μ FF (LCO)	% FF Incr. NI-on-NI	% FF Incr. I-on-I			
DWM_1	75.75%	1,217.39%	11.38%			
DWM_2	90.50%	6.15%	1.72%			
Vertmax_Batch	90.40%	69.04%	5.51%			
Latctl_Batch	85.64%	27.82%	9.70%			

more complex inlined implementation, and executed tests generated over the inlined implementation against the non-inlined implementation. In Table XI, we list the median fault finding effectiveness of tests generated over the inlined implementation when executed against a non-inlined implementation. We then note the increase in fault finding over executing tests generated from the non-inlined implementation against the same implementation and executing those test generated over the inlined implementation against the same implementation. We then compare tests generated over the non-inlined implementation and executed on the inlined structure against executing those same tests on the non-inlined structure and executing the tests based on inlined structure against the inlined structure.

From these results, we can see two key trends. First, we can see that not only are tests generated over the non-inlined implementation inadequate at covering the structure of the structurally complex implementation, but they also generally achieve worse fault finding over the complex inlined implementation. Tests generated over the non-inlined implementation attain between 17.88 to 98.83% fewer faults over the inlined implementation than than tests generated using the complex implementation to begin with (see Table XI). The exception to this is for the *Vertmax_Batch* system, paired with the maximum oracle, where fault finding improves by 16.71%.

The opposite is true when tests generated over the inlined system are executed against the simpler non-inlined implementation. With the exception of *DWM_2* with the impractically-large maximum

Table XII. Summary of statistical results

Hypothesis	Results
1	H_0_1 rejected for all systems.
2	H_0_2 rejected for all systems.
3	H_0_3 rejected for all systems.
4	H_0_4 rejected for all systems.
5	H_0_5 rejected for <i>Latctl.Batch</i> . We fail to reject H_0_5 for <i>DWM_1</i> , <i>DWM_2</i> , and <i>Vertmax.Batch</i> .
6	H_0_6 rejected for <i>DWM_1</i> , <i>DWM_2</i> , <i>Latctl.Batch</i> , and <i>Vertmax.Batch</i> (OO/LCO oracle). We fail to reject H_0_6 for <i>Vertmax.Batch</i> with the maximum oracle.
7	H_0_7 rejected for all systems.
8	H_0_8 rejected for <i>DWM_1</i> , <i>DWM_2</i> , <i>Latctl.Batch</i> , and <i>Vertmax.Batch</i> (OO oracle). We fail to reject H_0_8 for <i>Vertmax.Batch</i> with the maximum oracle.
9	H_0_9 rejected for <i>DWM_1</i> , <i>Vertmax.Batch</i> , <i>Latctl.Batch</i> , and <i>DWM_2</i> (OO/LCO oracle). We fail to reject H_0_6 for <i>DWM_2</i> with the maximum oracle.

oracle, tests generated over the complex inlined implementation achieved fault finding gains ranging from a modest 4.96% to a massive 5,068.49% over executing the tests generated using the simpler non-inlined implementation. This bolsters the results of *Q3*—we cannot expect tests generated from the non-inlined implementation to be effective at identifying faults in other implementation structures. However, we *can* expect tests generated over more complex implementations to be effective against other implementations.

Second, these results add an interesting extension to *Q2*. Not only are tests generated from the inlined implementation effective when executed against the inlined implementation—but they may be *more effective* when executed on the simpler version of the system structure. Tests generated from the inlined structure are 1.72-43.05% more effective when executed over the non-inlined implementation than when executed over the inlined implementation. This result reinforces the idea that we should use the more structurally complex version of the implementation when producing test cases, regardless of the final form that the system takes. It further indicates that we may be able to find more faults by simplifying the implementation when we execute those test cases.

The results to these four analyses raise a number of questions and concerns. In particular, we can see that the implementation structure potentially has a strong impact on both the cost and effectiveness of an MC/DC-driven testing process. This indicates that our choice of implementation structure, like our choice of coverage criterion, is an important practical aspect of the software testing process. This is further reflected in our results for *Q3*, which highlight the potential impact on a certification process using MC/DC simply achieving MC/DC over *some* implementation does not imply 100% coverage, or even high coverage, over all possible implementations. Similarly, our results for *Q4* reinforce that the choice of implementation is important when producing test cases. Tests generated using a simple implementation cannot be expected to be effective when executed on a complex implementation. The choice of implementation is important at *both* generation and execution time. We discuss these issues in detail in Section 5.

4.1. Demonstration of Statistical Significance

To ensure that the results for *Q1-Q4* are not due to chance, we propose and evaluate the following hypotheses:

- **Hypothesis (H_1):** A test suite satisfying MC/DC over the non-inlined implementation will require fewer tests relative to a test suite satisfying MC/DC over the inlined implementation.
- **Hypothesis (H_2):** When making use of the output-only oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.
- **Hypothesis (H_3):** When making use of the largest common oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.

- **Hypothesis (H_4)**: A test suite satisfying MC/DC over a non-inlined implementation will achieve less than 100% MC/DC over an inlined implementation.
- **Hypothesis (H_5)**: A test suite satisfying MC/DC over an inlined implementation will achieve less than 100% MC/DC over a non-inlined implementation.
- **Hypothesis (H_6)**: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding when executed against the inlined implementation than on the non-inlined implementation.
- **Hypothesis (H_7)**: When using any oracle, a test suite satisfying MC/DC over the inlined implementation will achieve a lower level of fault finding when executed against the inlined implementation than on the non-inlined implementation.
- **Hypothesis (H_8)**: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding when executed against the inlined implementation than a test suite satisfying MC/DC on the inlined implementation.
- **Hypothesis (H_9)**: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding when executed against the non-inlined implementation than a test suite satisfying MC/DC on the inlined implementation.

To evaluate our hypotheses, we first formed null hypotheses as follows:

- H_{01} : A test suite satisfying MC/DC over the non-inlined implementation will contain the same number of tests as a test suite satisfying MC/DC over the inlined implementation.
- H_{02} : When making use of the output-only oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.
- H_{03} : When making use of the largest common oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.
- H_{04} : A test suite satisfying MC/DC over a non-inlined implementation will achieve 100% coverage over an inlined implementation.
- H_{05} : A test suite satisfying MC/DC over an inlined implementation will achieve 100% coverage over a non-inlined implementation.
- H_{06} : When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding on both implementations.
- H_{07} : When using any oracle, a test suite satisfying MC/DC over the inlined implementation will achieve the same level of fault finding on both implementations.
- H_{08} : When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding on the inlined implementation as a test suite satisfying MC/DC over the inlined implementation.
- H_{09} : When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding on the non-inlined implementation as a test suite satisfying MC/DC over the inlined implementation.

To accept H_1-H_9 , we must reject $H_{01}-H_{09}$. Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate our hypotheses without any assumptions on the distribution of our data, we use a one-sided (strictly lower) Mann-Whitney-Wilcoxon rank-sum test [Wilcoxon 1945], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. As we cannot generalize across non-randomly selected case examples, we apply the statistical test for each pairing of case example and oracle type with $\alpha = 0.05^1$.

¹Note that we do not generalize across case examples or oracles as the needed statistical assumption, random selection from the population of case examples or oracles, is not met. The statistical tests are used to only demonstrate that observed differences are unlikely to have occurred by chance.

For H_1 through H_3 , there exist two sets of data: one containing measurements for the 100 reduced test suites satisfying 100% achievable coverage for non-inlined implementation, and one containing measurements for the 100 reduced test suites satisfying 100% achievable coverage for inlined implementation. For H_1 , these measurements are test suite sizes. For H_2 and H_3 , these measurements are fault finding effectiveness measurements. The application of the permutation test is therefore straightforward. For H_6-H_9 , a similar process is applied to that used for H_1-H_3 . The sole difference is that, for H_1-H_3 , we compared tests generated and executed on the same implementation to tests generated and executed on the other implementation. For H_6-H_9 , the generation and execution source may differ, as indicated by the individual hypotheses.

For H_4 and H_5 , we have only one set of data: the percentage of MC/DC achieved by each test suite generated from one implementation and executed over the other implementation. When performing the permutation test, we therefore use a set of equal size (100 records) consisting only of 100% coverage—that is, we compare against test suites achieving the maximum achievable coverage.

We perform this statistical test using each case example. For H_1-H_4 , our resulting p-values are each very small—less than 0.0001. Given a traditional $\alpha = 0.05$, we reject our null hypotheses for all instances. We see that the results support our hypotheses, and in these scenarios we accept H_1-H_4 for each case example.

For one of the case examples—*Latctl_Batch*—we reject the null hypothesis H_{05} . For the other three systems—*DWM_1*, *DWM_2*, and *Vertmax_Batch*—we fail to reject H_{05} . In the latter cases, tests generated to satisfy MC/DC over the inlined implementation almost always achieve 100% of the achievable MC/DC obligations over the non-inlined implementation. In the case of *Latctl_Batch*, the tests generated over the inlined implementation achieve very high MC/DC, but fail to achieve 100% of the achievable coverage on average.

A natural question to ask is—why would tests created to achieve MC/DC over the inlined system fail to achieve 100% coverage on the non-inlined system? Intuitively, the reverse is easy to imagine. MC/DC on the inlined system will require specific combinations of input that will not be required to achieve coverage of the non-inlined system. However, in general, the reverse can only happen when every instance of the condition in the non-inlined decision, once inlined, does not independently affect the outcome of any decision into which it is embedded. In this case, those conditions will show up as “unachievable” in the inlined model—no test can be produced that satisfies the test obligation. In satisfying the obligation for the inlined system, we would be required to produce a test that demonstrates this condition in its context. This test, by the nature of inlined code, would typically be stronger than necessary to demonstrate it outside of its context (i.e., on the non-inlined version of the system). These missing tests actually demonstrate that those conditions are, in fact, dead code, once considered in the context in which they are used.

As indicated in Table X, the only system where tests created for the inlined implementation fail to achieve 100% coverage on the non-inlined implementation is *Latctl_Batch*. There are two obligations that the inlined version of the suite fails to cover. One of those is explained by the above. The equivalent obligation for the inlined version is impossible to satisfy (see Table V), therefore, no test exists in the suite produced for the inlined version that covers the equivalent condition in the non-inlined implementation. That expression is, in fact, dead code.

The second can be explained as a result of the inlining process. *Latctl_Batch* runs in a dual-redundant configuration. There is an input that turns this model into a passive controller that just feeds through the outputs from the other side. If you allow this input to be true, you cannot prove anything about the model—the inputs from the other side are arbitrary. Therefore, this input was set to a constant value of false. The non-inlined system contains an expression used to set the system to the passive mode. However, as this expression, an *and* statement, would always evaluate to false given the environmental settings, the automated inlining process removed the expression completely. In practice, where code transformations would likely be done by hand, that expression would have remained in the system and an equivalent test for the inlined system would have been produced.

Regarding H_6 and H_8 —in both cases, we can reject the null hypothesis for three of the four systems for all oracles, but we fail to reject the null hypothesis for *Vertmax_Batch* when paired with the maximum oracle. For the former systems, tests generated over the non-inlined system and executed on the inlined system fail to outperform tests generated and executed on the non-inlined system and tests generated and executed on the inlined system. For the latter case, the tests generated on the non-inlined version of *Vertmax_Batch* find all non-equivalent mutants in the inlined system when the expensive maximum oracle is used. In that case, the oracle provides the necessary observability to identify those faults. When the more common output-only oracle is used, we can reject the null hypotheses for *Vertmax_Batch*.

We can reject H_{09} for three of the four systems for all oracles, but we fail to reject the null hypothesis for *DWM_2* when paired with the maximum oracle. For the former systems, tests generated over the non-inlined system and executed on the non-inlined system fail to outperform tests generated on the inlined system and executed on the non-inlined system. In the latter case, tests generated on the inlined system and executed on the non-inlined system perform slightly worse (-1.80%) on average than those generated and executed on the non-inlined system, when the maximum oracle is used. Again, when the more common output-only or the largest common oracle is used, we can reject the null hypotheses for *DWM_2*.

Based on these results, we conclude that a clear, statistically significant pattern exists in the four case examples used in our study: by varying implementation structure, we can impact both the number of test inputs required to satisfy MC/DC as well as the fault finding effectiveness of test suites achieving 100% achievable MC/DC.

5. DISCUSSION

For each research question in our study, we have found that the structure of the implementation has a strong, consistent impact on the testing process when using the MC/DC criterion.

Our results for *Q1* indicate that, by varying the structure of the implementation, we can influence—positively or negatively—the cost of generating test inputs to satisfy MC/DC. This observation is potentially useful in the context of MC/DC’s role in the certification process for critical avionics systems, as satisfying MC/DC during testing can be extremely expensive. By applying syntactic transformations to avionics systems, developers can generate systems for which MC/DC is significantly easier and cheaper to achieve. For our examples, inlined implementations required test suites at least twice as large as the size of the test suites generated over the semantically equivalent non-inlined systems—meaning that there is a potentially dramatic savings in testing costs when using a non-inlined system.

Unfortunately, our results for *Q2* indicate that test suites generated to satisfy structural coverage criteria over the inlined implementation generally outperformed those test suites generated to satisfy structural coverage criteria over the non-inlined implementation, with relative improvements of up to 4,542.47% when using the fixed-sized test oracles. Thus, we see a potential tradeoff—we *can* restructure our implementation to reduce the cost of satisfying coverage criteria, but we incur the risk of reducing the effectiveness of the testing process.

Both of these implications point at a deeper issue: the sensitivity of the MC/DC criterion to the structure of the implementation. Our results for *Q3* indicate that while implementation structure impacts the cost and effectiveness of the MC/DC criterion, it also impacts the measurement of the criterion. Tests generated using a simpler implementation cannot be expected to attain high coverage of the structurally complex implementation, while tests generated over the structurally complex implementation *can* be expected to cover the test obligations of the structurally simple implementation. Similarly, the results of *Q4* indicate a similar idea. Tests generated using the complex implementation are not only effective at finding faults in the same implementation, but are effective when that implementation is varied. Tests generated using the simple implementation are less effective regardless of the implementation used during test execution.

In the avionics domain, the role of the MC/DC criterion is to determine if our test inputs are adequate, as implied in the synonymous term *test adequacy metric*. It is therefore worth questioning if

this metric is *itself* adequate. Given that the effectiveness of MC/DC can vary considerably depending on the structure, ranging from very high to—in the case of the *DWM_I* system—very low, when asking the question: *is this set of test inputs adequate?* it is difficult to fully trust the results of the criterion. Therefore, we believe that either care must be taken when using MC/DC as an adequacy metric, or that another method of measuring test adequacy be constructed. By specifying a specific implementation structure for measuring and producing MC/DC-satisfying test suites, we can ensure that the potential benefits of using MC/DC as an adequacy criterion are more likely to be gained through its use. In the remainder of this section, we discuss observations and concerns raised from these results.

5.1. Cost and Effectiveness Increase with Complexity

Citing Tables VII and VIII, we again note that both size and fault finding effectiveness increase when using a more complex implementation structure, sometimes dramatically. An extreme example of this is the *DWM_I* system: on average, 1,566.66% more tests are required on average when satisfying MC/DC on the inlined system versus the non-inlined system, resulting in 4,542.47% more faults detected on average.

This increase in tests can be attributed to the increased complexity of the coverage obligations generated when using the inlined implementation as opposed to the non-inlined implementation. Recall that when transforming the non-inlined implementation into the inlined implementation, the complexity of conditional expressions increases, as intermediate variables formerly used as atomic Boolean conditions are instead replaced with more complex subexpressions. We illustrate an example of this in Figure 5.

Table XIII. Impact of implementation structure on coverage obligations

Non-inlined MC/DC Obligations	Inlined MC/DC Obligations
(1) $(x \wedge y)$	(1) $(x \wedge y) \wedge \neg(z \wedge w)$
(2) $(\neg x \wedge y)$	(2) $\neg(x \wedge y) \wedge (z \wedge w)$
(3) $(x \wedge \neg y)$	(3) $(\neg x \wedge y) \wedge \neg(z \wedge w)$
(4) $(z \wedge w)$	(4) $(x \wedge \neg y) \wedge \neg(z \wedge w)$
(5) $(\neg z \wedge w)$	(5) $\neg(x \wedge y) \wedge (\neg z \wedge w)$
(6) $(z \wedge \neg w)$	(6) $\neg(x \wedge y) \wedge (z \wedge \neg w)$
(7) $(x \wedge y) \wedge \neg(z \wedge w)$	
(8) $\neg(x \wedge y) \wedge (z \wedge w)$	
(9) $\neg(x \wedge y) \wedge \neg(z \wedge w)$	

In the case of MC/DC the complexity of the obligations we must meet to satisfy MC/DC is related to the complexity of the expressions. To understand why, consider Figure 5. Here we see the number of atomic conditions on line 4 grows from two to four when line 1 and 2 are inlined. Consider the obligations we must satisfy to achieve MC/DC over these expressions: we must demonstrate each condition can *positively* and *negatively* affect the outcome of the expression. In other words, we must show each condition can cause the expression to be true and false. Consequently, as the number of

```
[1] internal42 = x AND y
[2] internal13 = z AND w
[3]
[4] output1 = (internal42 OR internal13)
```

Non-inlined Implementation

```
[1] output1 = (x AND y) OR (z AND w)
```

Inlined Implementation

Fig. 5. Increasing complexity of boolean decisions during inlining transformation

conditions grows, the number of MC/DC obligations for the expression also grows, and with it the number of test inputs we must generate also generally grows.

The MC/DC obligations to satisfy can be schematically derived, resulting in the obligations in Table XIII [Whalen et al. 2006]. We can see that, while the non-inlined implementation actually has more obligations (nine rather than six)—due to the larger number of expressions (three small expressions rather than one large one)—the inlined implementation’s obligations are generally more complex and require more specific inputs. As a result, opportunities to generate test inputs satisfying multiple obligations—allowing a reduction in the total number of inputs required—are more limited for the obligations generated from the inlined implementation.

For example, consider the obligations 7-9 for the non-inlined system. These obligations are logically disjoint, and thus require three test cases. However, by carefully selecting the test cases, we can also satisfy all the obligations for lines 1-6, as shown in Table XIV.

Table XIV. Impact of implementation structure on coverage obligations

Inputs				Non-inlined Obligations									Inlined Obligations					
x	y	z	w	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6
T	F	T	T	F	F	T	T	F	F	F	T	F	F	T	F	F	F	F
T	T	T	F	T	F	F	F	T	T	F	F	T	F	F	F	F	F	F
F	T	F	T	F	T	F	F	T	F	F	F	T	F	F	T	F	T	F

We have less freedom in selecting test inputs for the inlined implementation’s more complex obligations, and thus, this scenario is not possible. The test inputs above only satisfy inlined obligations 1, 2, 3, and 5, leaving the remaining obligations to be covered by other test inputs. Therefore, while fewer obligations are generated, more test inputs are required.

These increases in test suite size result in generally improved fault finding, for two apparent reasons. First, and perhaps most obvious, is the increase in test suite size. Recent work on modeling the effectiveness of testing has indicated that effectiveness is highly dependent on test suite size [Namin and Andrews 2009; Gligoric et al. 2013; Inozemtseva and Holmes 2014]. By simply increasing the number of test inputs used, we can improve the effectiveness of our testing process.

Second, by changing the implementation structure, we have strengthened the constraints on our coverage obligations, resulting in not only more test inputs, but test inputs that are differentiated—exploring combinations of conditions that would not otherwise be explored. For example, consider a situation where the implementations in Figure 5 are incorrect—where the final implementation line should be $((x \text{ AND NOT } y) \text{ OR } (z \text{ AND NOT } w))$. The test suite depicted in Table XIV, while sufficient to satisfy MC/DC over the non-inlined implementation, would not pick up on this fault. However, if we created a test suite to satisfy MC/DC over the inlined implementation, any test sufficient to satisfy inlined obligation 4 would detect this particular fault.

5.2. Coverage Measurements Vary with Complexity

In the previous section, we discussed the factors contributing to the results observed for $Q1$ and $Q2$, demonstrating how changing implementation structure increases the number of test inputs required and the effectiveness of said test inputs. The same factors contribute to the results observed for $Q3$, in which we found that test inputs generated to satisfy 100% achievable MC/DC over a non-inlined implementation result in less than 100% achievable coverage over an inlined implementation.

The low average levels of MC/DC—as low as 13.08%—achieved over the inlined implementation by the non-inlined test suites were somewhat surprising. One reason for the reduction in coverage is likely to be the reduction in the number of test inputs. As seen in Table VII, the average size of a test suite generated over the non-inlined system is—at largest—less than half of the average size of the test suite for the inlined version of the same system.

However, we believe that the dramatic drop in achieved coverage is not only a result of test suite size, but also of the test generation method. By using counterexample-based test generation, we ensure that each coverage obligation is satisfied. However, each generated test input is specifically

targeted at satisfying a single coverage obligation. Thus, each input tends to perform the minimum number of actions needed to cover that obligation. This is because, in the context of model checking, generated counterexamples are ideally short and simple, so as to be better understood by the user. In particular, default input values (e.g., *0 or false*) are used whenever possible, and generally only the variables that must be manipulated to satisfy the coverage obligation are changed. Thus, when generating test inputs to satisfy MC/DC over the non-inlined implementation, the resulting test inputs may not be sufficiently complex to satisfy coverage obligations over the inlined implementation; the quantity of inputs is not only low, but each individual test input is also less complex.

To demonstrate this, for each case example, we used the inlined test suites to generate reduced test suites of size equal to the reduced test suites from the non-inlined implementation. In other words, for each reduced test suite generated using the non-inlined implementation, we generated a reduced test suite of equal size using the inlined implementation². We then computed the average coverage achieved over the inlined implementation.

We present the results in Table XV, along with the sizes and original average coverages for the non-inlined test suites (also found previously in Table VII). As shown, we see that reduced test suites of size equal to those drawn from the non-inlined implementation can provide much higher coverage levels, provided sufficiently effective test inputs are used to construct the test suites. In our case examples, increases in coverage of 41.24% to 643.27% are observed.

Table XV. Average coverage of reduced suites. NI = Non-inlined, I = Inlined

	DWM_1	DWM_2	Latcl.Batch	Vertmax.Batch
Median Size	27.00	69.00	28.00	68.00
NI Suite Coverage Over I	13.08%	53.25%	67.67%	31.95%
Paired I Suite Coverage Over I	97.22%	90.60%	95.58%	84.04%
% Increase	643.27%	70.14%	41.24%	163.04%

One key observation that can be seen in Table XV is that a small number of tests, created for the inlined implementation, are often sufficient to cover a large number of test obligations. When formulating test obligations, some will be simple and some will require complex combinations of input to satisfy. When working with an inlined implementation—by the nature of inlining—more obligations will be of the latter format. Thus, when designing tests to satisfy those obligations, the tests will cover a large variety of the possible input combinations. Some of those inputs will be so specific that they only satisfy a small number of obligations, but a more common result is that a number of tests will be required to execute the system for enough execution cycles or method calls to trigger the exact required combination of conditions. These tests are highly likely to cover several additional obligations in the process of covering the one specific obligation they were designed to cover.

This is another piece of evidence in favor of designing tests over the inlined implementation—tests are likely to exercise the system more thoroughly. Tests may cover a larger portion of the state space or execute the system for a longer period of time, exploring a wider variety of combinations of input as they execute, and satisfying a large number of obligations in the process. As a result, even a small number of tests, created for the inlined version of the system, may achieve higher coverage and fault detection capability than tests created for the non-inlined implementation.

5.3. Effectiveness Varies With Complexity At Both Generation and Execution Time

Given the ease of satisfying MC/DC over a non-inlined implementation, one could imagine a scenario where test generation is performed on a structurally simple implementation such as a Simulink model or an uninlined version of the source code, then such tests are later executed against a different implementation (such as the real code, if the Simulink model was used during generation).

²The randomized greedy algorithm was again used, but the stopping point was no longer 100% coverage, but instead a test suite size.

Table XVI. Median (μ) Fault finding measurements when tests are generated using the inlined implementation, the suites are reduced to match the size of suites generated using the non-inlined implementation, and then the suites are executed on the non-inlined implementation. FF = fault finding. OO = output-only oracle, MX = maximum oracle, LCO = largest common oracle.

	Reduced I-on-NI μ FF(OO)	% FF Increase Over NI-on-NI
DWM_1	25.02%	1,613.70%
DWM_2	88.32%	3.86%
Vertmax_Batch	74.38%	45.22%
Latctl_Batch	72.15%	18.24%
	Reduced I-on-NI μ FF(MX)	% FF Increase Over NI-on-NI
DWM_1	83.68%	15.77%
DWM_2	91.05%	-3.69%
Vertmax_Batch	86.17%	-6.66%
Latctl_Batch	95.29%	3.21%
	Reduced I-on-NI μ FF(LCO)	% FF Increase Over NI-on-NI
DWM_1	27.10%	414.23%
DWM_2	88.43%	3.48%
Vertmax_Batch	75.09%	40.41%
Latctl_Batch	76.75%	14.55%

The results of *Q3* indicate that this would not be advisable, as tests generated over a simple implementation cannot be expected to attain coverage over a complex implementation. The results of *Q4* further make this point—tests generated over the non-inlined implementation were less effective at finding faults in the inlined implementation than tests generated and executed over the inlined implementation. In the reverse case, tests generated over the inlined implementation are *more effective* at finding faults in the non-inlined implementation than tests generated and executed over the non-inlined implementation.

As the results of *Q2* indicated, the size of the test suite is a factor in the effectiveness of that suite. Thus, these results are not entirely surprising—by generating on the non-inlined system, we end up with a smaller number of tests to execute on the inlined system than when we generated using the inlined system. By generating on the inlined system, we end up with substantially more tests that can be executed on the non-inlined system than when we generated using the non-inlined system. To better understand the effect of test suite size on these results, we took test suites generated on the inlined system and reduced them to the size of suites generated using the non-inlined system (the same suites used in Section 5.2). We then computed the median fault finding of these reduced test suites when executed against the non-inlined implementation.

The results of this experiment can be seen in Table XVI. We can see that, while there was a drop in fault finding effectiveness when the suite size was reduced, the tests generated using the inlined implementation were still more effective at fault finding in nearly all situations. When using the output-only or largest common oracle, the suites generated using the inlined implementation found 3.48-1,613.70% more faults in the non-inlined implementation than the tests generated using that implementation. This suggests that the larger number of tests required to satisfy MC/DC over the inlined implementation are not the sole source of the improved effectiveness of those suites. Indeed, just as with the coverage analysis in Section 5.2, it seems that the complex combinations of input required to satisfy MC/DC over complex implementations are also likely to reveal faults. It is not enough to simply satisfy the letter of the law—to attain MC/DC in any manner possible. Whether a test obligation is covered is less important than *how it was covered*.

Regardless of the structure used during test execution, these results provide more evidence in favor of using the structurally complex implementation to generate test cases. Although the cost of producing the tests will be higher than when a simple structure is used, the resulting tests will be effective at attaining coverage *and* finding faults. By more thoroughly exercising the system under test, even a small number of tests created using the inlined implementation may be effective at finding faults in the source code.

The results of *Q4* in Table XI offered one additional observation of interest. Test suites generated using the inlined implementation were not only more effective at finding faults in the non-inlined implementation than tests generated on that implementation, but they attained *higher* fault finding on the non-inlined implementation than on the inlined implementation. This hints that there may, in fact, be a use for a simple implementation—as a version of the system to *execute* tests on. If a tool can be used to automatically inline or uninlne code, then the code could be maximally inlined for test generation purposes—offering the benefits of MC/DC satisfaction on a complex implementation. The code could then be maximally uninlne during test execution, potentially making it easier to observe the effect of program faults triggered by those test cases.

More research is needed to confirm this hypothesis, but these results make it clear that implementation structure is important not only during test generation, but when those tests are executed as well. If tests are generated using a simple implementation, they cannot be expected to be effective when executed on a complex implementation. If tests are generated using a complex implementation, they are more likely to be effective no matter what implementation is used at execution time. Further, additional improvement in effectiveness may be possible by generating over a complex implementation, then executing over a simple implementation.

5.4. Less Than 100% Fault Finding

Fault finding was often significantly lower than 100% over the case examples—regardless of the test oracle employed, and particularly for non-inlined test suites. Although it is well known that testing is an incomplete form of verification, we were initially surprised by the poor fault finding of test suites that meet the rigorous MC/DC criterion. As we studied the models closely, we identified several possible reasons for the poor absolute fault finding, including:

Faults in uncovered portions of the model: As seen from our results in Table V, in several instances, the achievable MC/DC over the implementation is less than 100%. This implies that there are portions of the implementation for which there exists no test case that can provide MC/DC. We term these as the “uncovered” portion in the implementation. Note here that the test suites we use in our experiment provide maximum achievable MC/DC over their respective implementation. When seeding faults in an implementation, we may seed faults in the uncovered portion. Since no MC/DC test case could be constructed to cover this portion, the test suite will likely (though not necessarily) miss such faults. In our experiment, we did not attempt to identify the faults that are seeded in the uncovered portion of the implementation. Such a task would be time consuming and difficult since it would require manually examining the implementation, test suites, and mutations.

Delay expressions: All of the studied case examples execute on some form of execution cycle; they sample the environment, execute to completion, and then wait until it is time for a new execution cycle. A *delay expression* is one that uses the value of an expression from a previous execution cycle. This delay expression is explicit in modeling languages such as SCADE and Simulink; in programming languages like C the same effect is achieved using state variables that are assigned during one cycle and used by statements earlier in the loop during the next cycle. Delay faults are not a classic mutation to perform; however, this kind of fault occurs with regularity in control software, which runs as a periodic polling loop. Errors occur when the “previous” value is used when the “current” value is expected or vice-versa; these kind of errors affect edge-detection, input smoothing, integration, and other state-requiring operations.

The key here is that delay expressions (or state variables) are assigned one or more cycles before use. Thus, it is possible—especially when using a model checker that intentionally generates short tests—that a test can terminate too early (i.e., after covering the structure that is assigned to a state variable, but before the variable can propagate to an output)³.

To illustrate this problem in a C-like implementation language, consider the program fragment in Figure 6. In this code, execution steps are represented as loop iterations. Variables used in future

³Note that this problem is different from semantically equivalent mutants since it is *possible* to reveal the mutation, but only with a test case longer than what is necessary to achieve MC/DC.

```

void Delay_Expr()
{
    bool pre_var;
    bool var_a = false;

    while(1)
    {
        in1 = sample(in1); // Update value
        in2 = sample(in2); // Update value
        pre_var = var_a; // Store previous value
        var_a = in1 or in2; // Calculate new value
        print pre_var; // Display new output
    }
}

```

MC/DC Test Set for (in1, in2):

TestSet1 = { (TF) , (FT) , (FF) }

Fig. 6. Sample program fragment that uses variable values from previous step

execution steps are stored away as intermediate variables for use in later loop iterations. Because of this delayed usage mechanism, these intermediate variables cannot be inlined. TestSet1 with one step test cases will provide MC/DC of this program fragment, since we only need to exercise the *or* Boolean expression up to MC/DC. If we were to erroneously replace the *or* operator with *and*, or any other Boolean operator, TestSet1 providing MC/DC would not be able to reveal the fault, since the test cases are too short to affect the output—the test cases only consist of one input step, they would need to be at least 2 steps long for failures to propagate to the output. Many systems in the domains of vehicle or plant control (such as the avionics domain) are designed to use variable values from previous steps. Thus, test cases generated to provide MC/DC over such systems will often be shorter than needed to allow erroneous state information to propagate to the outputs. Based on this observation and our results in Table VIII, we believe that delay expressions represent a serious concern related to the effectiveness of MC/DC test suites.

Intermediate variable masking: As mentioned previously, generated test suites do not ensure that intermediate variables affect the output. While we expect such masking to occur with non-inlined implementations, masking is also possible with inlined versions of implementations as they are not completely inlined (this is discussed in Section 3.4 and is also seen in the delay expression discussion). We may, therefore, still have some intermediate variables in the inlined implementations that present opportunities for masking. Figure 7 shows a sample C like program with an intermediate variable *no_alarm* that cannot be inlined and can, therefore, potentially be masked out in a test case.

TestSet1 in Figure 7 provides MC/DC over the *compute* function; the test cases with *in3 = false* (bold faced) contribute towards MC/DC of the *in1 or in2* condition in the if-then-else statement. However, these test cases mask out the effect of the intermediate variable *no_alarm* in the *and* expression since *in3 = false*. Suppose the code fragment in Figure 7 was faulty, the correct expression should have been *in1 and in2* (which was erroneously coded as *in1 or in2*). TestSet1 providing MC/DC would be incapable of revealing this fault, since there would be no change in the observable output. Thus, seeded faults like the one mentioned here cannot be revealed by a test suite achieving MC/DC because of intermediate variable masking. This observation serves as a reminder that masking is a crucial consideration for generating test suites that are effective in fault finding.

One potential solution for this is to use stronger test oracles which consider the value of internal state. Such test oracles, when used in conjunction with test inputs satisfying MC/DC, have been empirically and theoretically shown to provide increased fault finding capability in this domain and others [Gay et al. 2015a; Staats et al. 2012a; Staats et al. 2011b]. Additionally, Whalen et al. have proposed a stronger variant of MC/DC (dubbed Observable MC/DC) that addresses mask-

```

bool Compute(bool in1,in2,in3)
{
    bool no_alarm;

    if (in1 or in2)
        no_alarm = true
    else
        no_alarm = false;
    return (in3 and no_alarm);
}

MC/DC Test Set for (in1, in2, in3):
TestSet1 = { (TFF) , (FTF) , (FFT) , (TTT) }

```

Fig. 7. Sample C like inlined program fragment

ing by requiring a propagation path from a decision statement to a variable observed by the test oracle [Whalen et al. 2013]. Observable MC/DC demonstrates higher fault finding effectiveness than masking MC/DC—particularly for non-inlined systems. Our experimental results highlight the importance of such work.

6. THREATS TO VALIDITY

External Validity: We have conducted our study on four synchronous reactive critical systems. We believe these systems are representative of the class of systems in which we are interested, and our results are thus generalizable to other systems in the domain.

We have used implementations expressed in Lustre rather than a more common language such as C or C++. Nevertheless, systems written in the Lustre language are similar in style to traditional imperative code produced by code generators used in embedded systems development. We therefore believe that testing Lustre code is sufficiently similar to testing reactive systems written in traditional imperative languages.

We have generated our test inputs using a model checker. Other possible options would be to use tests manually created by testers, tests generated through other automated processes, or randomly generated tests. It is possible these other options would yield results that are significantly different. However, in our experience, tests generated using a model checker are relatively *less effective* than other options; we, therefore, are effectively studying worst-case (or at least not exceptionally positive) behavior of the MC/DC criterion. One reason for this, as discussed in Section 5.2, is that the tests produced lack *diversity*—the model checker prefers to leave inputs at default values when not required to satisfy a particular property. Another reason, as discussed in Section 5.4, is that the tests are often not long enough to ensure fault propagation to output (however, this is more of a problem with the coverage criterion than the generation tool—the tool does exactly what it is asked to do and no more). Given that we are evaluating the impact of implementation structure on the effectiveness of this criterion, both of these traits allow us to clearly explain the risks involved in changing the implementation structure.

We have examined 100 test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [Rajan et al. 2008; Staats et al. 2010].

We have varied implementation structure using an in-house tool for transforming implementations. It is possible that one or both structures used do not correspond to an implementation structure we are likely to see in actual implementations. Nevertheless, the non-inlined implementation structure is very similar to the original structure of the system before it is translated from Simulink, and we thus believe it is representative of systems likely to be considered. The inlined implementation structure is similar to the code generated from tools such as Real-Time Workbench.

Internal Validity: Our study makes extensive use of automation. It is possible that some effects observed are due to errors in the automation of the experiment. However, much of this automation is part of an industrial framework (courtesy of Rockwell-Collins), particularly the more complex automation for implementation transformation, and has been extensively verified [Miller et al. 2010]. Other automation for running tests, producing oracles, etc., developed at the University of Minnesota has also undergone verification, and is relatively simple. We therefore believe it is highly unlikely that errors that could lead to erroneous conclusions exist in the automation.

Construct Validity: In our study, we measure fault finding over seeded faults, rather than real faults encountered during development of the software. It is possible using real faults would lead to different results. However, Andrews et al. have shown the use of seeded faults leads to conclusions similar to those obtained using real faults in fault finding experiments [Andrews et al. 2006]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [Just et al. 2014].

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions for these analyses are met, and thus have favored non-parametric methods. In cases where the base assumptions are clearly not met, we have avoided using statistical methods. Notably, we have avoided statistical inference across case examples, as we have not randomly sampled these examples from the larger population.

7. RELATED WORK

Work related to this study can be divided into roughly two groups: work related to structural coverage criteria, and work related to the MC/DC criterion specifically.

7.1. Structural Coverage

The current favored proxy for measuring the adequacy of testing efforts is the coverage of structural elements of the source code of the software, such as individual statements, branches of the softwares control flow, and complex boolean conditional statements [Kit and Finzi 1995; Perry 2006; Pezzé and Young 2006]. The idea is simple, but compelling—unless code is executed, faults will never be found. Using coverage as a measure for the adequacy of testing gives developers a number that can be quickly calculated and used as a target, as a measure of what has been accomplished, as the basis for planning future efforts, and—importantly—as the means by which we can enable to automated generation of test cases.

According to Chilenski and Miller [Chilenski and Miller 1994], structural coverage criteria are divided into two types: data flow and control flow. Data flow criteria measure the flow of data between variable assignments and references to the variables. Data flow metrics, such as all-definitions and all-uses [Beizer 1990], involve analysis of the paths (or subpaths) between the definition of a variable and its subsequent use. The structural coverage criteria in many standards, including DO-178C [RTCA/DO-178C], are often control flow criteria. Control flow criteria measure the flow of control between statements and sequences of statements. Pezzé and Young [Pezzé and Young 2006] discuss most of the well known control flow metrics used in structural testing, including, statement coverage, branch coverage, condition coverage, MC/DC, path coverage, and call coverage.

Automated test generation methods can use coverage criteria as optimization targets, producing test cases that cover a large portion of code with minimal human effort [McMinn 2004]. If coverage is, in fact, correlated with fault finding effectiveness, then automated test generation represents a powerful direction in lowering the cost and human effort associated with software testing.

That said, recent work has endeavored to revise and clarify the exact role of structural coverage in testing, and to quantify the actual power of coverage to predict for faults [Groce et al. 2014]. Studies on the effectiveness of test suites created with the goal of satisfying common coverage metrics have yielded inconclusive results, some noting positive correlation between coverage level and fault detection [Namin and Andrews 2009; Mockus et al. 2009], while more recent work paints a negative portrait of the effect of coverage on improving the detection of real faults [Inozemtseva and Holmes 2014]—particularly when tests have been automatically generated [Fraser et al. 2013]. Previous

work from the authors of this publication echoes the latter results—tests generated with the goal of maximizing coverage over the code, including MC/DC-satisfying tests, were often outperformed by test cases generated at random, and even for systems where the generated tests outperformed the random tests, the generated tests often found fewer than 50% of the seeded faults [Gay et al. 2015b; 2014; Staats et al. 2012b]. These results match recent experiments performed using state-of-the-art test generation techniques, where—despite noting positive correlation between fault finding and coverage—the authors found that automated testing performed poorly *overall* (collectively detecting only 55.7% of the faults) [Shamshiri et al. 2015].

In this work, we have focused on the idea that a major factor in determining the power of coverage as a proxy for adequate testing is the structure of the code and the sensitivity of the coverage criterion to that structure. Another factor that has been explored is the size of the test suite, with the majority of work confirming the intuitive idea that larger test suites are more effective at detecting faults [Gligoric et al. 2013; Inozemtseva and Holmes 2014]. Our previous work, while noting the same effect, has found that an even more important factor is the number and selection of variables examined by the test oracle [Staats et al. 2011a; Staats et al. 2012a; Gay et al. 2015a]. Careful selection of which variables to monitor and check for failing values is a major factor in improving the fault finding effectiveness of tests created to satisfy structural coverage criteria.

7.2. MC/DC

Hayhurst et al. first observed the sensitivity of the MC/DC metric, stating that “*if a complex decision statement is decomposed into a set of less complex (but logically equivalent) decision statements, providing MC/DC for the parts is not always equivalent to providing MC/DC for the whole*” [Hayhurst et al. 2001]. Our work attempts to quantify the difference, both in terms of coverage and fault finding effectiveness, between measuring MC/DC on complex decisions versus simple decisions.

Gargantini et al. have also observed the sensitivity of structural coverage metrics to modification of the code structure and have proposed a method of automatically measuring the resilience of a piece of code to modification [Gargantini et al. 2013]. The AURORA tool produces copies of a piece of software where the code has been transformed through user-defined rules. This process—similar to the practice of mutation testing [Andrews et al. 2006]—allows for the calculation of a fragility index measuring the sensitivity of the coverage level to changes in code structure.

Chilenski made the observation that “*If the number of tests M is fixed at $N + 1$ (N being the number of conditions), the probability of distinguishing between incorrect functions grows exponentially with N , $N > 3$* ” [Chilenski 2001]. This observation is based only on the number of tests, not on which tests were run. The results in our experiments support this observation. For our industrial examples, test suites that provide MC/DC over the non-inlined implementation provided poor coverage over the inlined implementations. The results indicate that MC/DC, when measured, should be on the inlined implementation with complex decisions where N is usually much larger than on the non-inlined implementation with simple decisions (with smaller N). Using our results along with Chilenski’s observation, we infer that a given test suite would be more effective in revealing incorrect functions in the inlined implementation than in the non-inlined implementation.

Despite the importance of the MC/DC criterion [Chilenski and Miller 1994; RTCA 1992], studies of its effectiveness are few. Yu and Lau study several structural coverage criteria, including MC/DC, and find MC/DC is cost effective relative to other criteria [Yu and Lau 2006]. Kndl and Kirner evaluate MC/DC using an example from the automotive domain, and note less than perfect fault finding [Kndl and Kirner 2011]. Dupuy and Leveson evaluate MC/DC as a complement to functional testing, finding that the use of MC/DC improves the quality of tests [Dupuy and Leveson 2000]. Our previous work found that automatically generating tests with the goal of achieving MC/DC led to poor results, but that the use of MC/DC as a stopping criterion for existing testing efforts yielded some benefit [Staats et al. 2012b; Gay et al. 2015b]. None of these studies, however, explore the impact of program structure on the criterion.

8. CONCLUSION

In this work, we have explored the impact of implementation structure on the efficacy of test suites satisfying the MC/DC criterion using four real world avionics systems. For each system, we automatically constructed two semantically equivalent implementations: a *non-inlined* implementation which uses only simple Boolean expressions, and an *inlined* implementation which uses more complex Boolean expressions.

Our results demonstrate that test suites achieving maximum achievable MC/DC over inlined implementations are generally larger than test suites achieving maximum achievable MC/DC over non-inlined implementations, with increases in size of 125.00%-1,566.66% observed. This increase in test suite size also generally corresponds with an increase in fault finding effectiveness, with improvements up to 4,542.47% observed at fixed oracle sizes. We found that test suites generated over non-inlined implementations achieve significantly less MC/DC when applied to inlined implementations, 13.08%-67.67% in our study. We also found that test suites generated over non-inlined implementations cannot be expected to yield effective fault finding on inlined implementations, finding 17.88-98.83% fewer faults than tests generated and executed on the inlined implementation. Tests generated using the inlined implementation generally attained 100% MC/DC on the non-inlined system, and found up to 5,068.49% more faults than tests generated and executed on the non-inlined implementation.

We believe these results are concerning, particularly in the context of certification: we have demonstrated that by measuring MC/DC over simple implementations, we can significantly reduce the cost of testing, but in doing so we also reduce the effectiveness of testing. Thus developers have an economic incentive to “cheat” the MC/DC criterion (and by building tools similar to those used in our study, the means), but this cheating leads to negative consequences.

Accordingly, we recommend organizations adopt a canonical—at least moderately structurally complex—implementation form for testing purposes to avoid these negative consequences.

REFERENCES

- J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on* 32, 8 (aug. 2006), 608 –624. DOI :<http://dx.doi.org/10.1109/TSE.2006.83>
- Boris Beizer. 1990. *Software Testing Techniques, 2nd Edition*. Van Nostrand Reinhold, New York.
- J. Chilenski. 2001. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Technical Report DOT/FAA/AR-01/18. Office of Aviation Research, Washington, D.C.
- J. J. Chilenski and S. P. Miller. 1994. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal* (September 1994), 193–200.
- A. Dupuy and N. Leveson. 2000. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software. In *Proc. of the Digital Aviation Systems Conf. (DASC)*. Philadelphia, USA.
- Esterel-Technologies. 2004. SCADe Suite Product Description. <http://www.estrel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html>. (2004).
- Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2013. Does Automated White-box Test Generation Really Help Software Testers?. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 291–301. DOI :<http://dx.doi.org/10.1145/2483760.2483774>
- Andrew Gacek. 2015. JKInd - a Java implementation of the KIND model checker. <https://github.com/agacek>. (2015).
- Angelo Gargantini, Massimo Guarneri, and Eros Magri. 2013. AURORA: AUTomatic RObustness coveRage Analysis Tool. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 463–470.
- A. Gargantini and C. Heitmeyer. 1999. Using Model Checking to Generate Tests from Requirements Specifications. *Software Engineering Notes* 24, 6 (November 1999), 146–162.
- G. Gay, M. Staats, M. Whalen, and M. Heimdahl. 2015a. Automated Oracle Data Selection Support. *Software Engineering, IEEE Transactions on* PP, 99 (2015), 1–1. DOI :<http://dx.doi.org/10.1109/TSE.2015.2436920>
- G. Gay, M. Staats, M. Whalen, and M.P.E. Heimdahl. 2015b. The Risks of Coverage-Directed Test Case Generation. *Software Engineering, IEEE Transactions on* PP, 99 (2015). DOI :<http://dx.doi.org/10.1109/TSE.2015.2421011>
- Gregory Gay, Matt Staats, Michael W. Whalen, and Mats P. E. Heimdahl. 2014. Moving the Goalposts: Coverage Satisfaction is Not Enough. In *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST 2014)*. ACM, New York, NY, USA, 19–22. DOI :<http://dx.doi.org/10.1145/2593833.2593837>

- Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-adequate Test Suites Using Coverage Criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 302–313. DOI :<http://dx.doi.org/10.1145/2483760.2483769>
- Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 255–268. DOI :<http://dx.doi.org/10.1145/2661136.2661157>
- G. Hagen. 2008. *Verifying safety properties of Lustre programs: an SMT-based approach*. Ph.D. Dissertation. University of Iowa.
- N. Halbwachs. 1993. *Synchronous Programming of Reactive Systems*. Kluwer Academic Press.
- K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report TM-2001-210876. NASA.
- M.P.E. Heimdahl, M.W. Whalen, A. Rajan, and M. Staats. 2008. On MC/DC and implementation structure: An empirical study. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*. 5.B.3–1–5.B.3–13. DOI :<http://dx.doi.org/10.1109/DASC.2008.4702848>
- Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. DOI :<http://dx.doi.org/10.1145/2568225.2568271>
- René Just, Darioosh Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, 654–665.
- S. Kandl and R. Kirner. 2011. Error Detection Rate of MC/DC for a Case Study From the Automotive Domain. *Software Technologies for Embedded and Ubiquitous Systems* (2011), 131–142.
- Edward Kit and Susannah Finzi. 1995. *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- Mathworks Documentation. 2015. Types of Model Coverage. <http://www.mathworks.com/help/slvnv/ug/types-of-model-coverage.html>. (2015). Accessed: 2015-08-19.
- Mathworks Inc. 2015. Mathworks Inc. Simulink. <http://www.mathworks.com/products/simulink>. (2015).
- Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14 (2004), 105–156.
- Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. 2010. Software model checking takes off. *Commun. ACM* 53, 2 (2010), 58–64. DOI :<http://dx.doi.org/10.1145/1646353.1646372>
- A. Mockus, N. Nagappan, and T.T. Dinh-Trong. 2009. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. 291–301. DOI :<http://dx.doi.org/10.1109/ESEM.2009.5315981>
- A.S. Namin and J.H. Andrews. 2009. The influence of size and coverage on test suite effectiveness. (2009).
- William Perry. 2006. *Effective Methods for Software Testing, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA.
- M. Pezzé and M. Young. 2006. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons.
- A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. 2008. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proc. of the 30th Int'l Conf. on Software engineering*. ACM, 161–170.
- S. Rayadurgam and M.P.E. Heimdahl. 2001. Coverage Based Test-Case Generation Using Model Checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*. IEEE Computer Society, 83–91.
- RTCA. 1992. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA.
- RTCA/DO-178C. Software Considerations in Airborne Systems and Equipment Certification. (????).
- Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2015)*. ACM, New York, NY, USA.
- M. Staats, G. Gay, and M.P.E. Heimdahl. 2012a. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 2012 Int'l Conf. on Software Engineering*. IEEE Press, 870–880.
- Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. 2012b. On the Danger of Coverage Directed Test Case Generation. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Lecture Notes in Computer Science, Vol. 7212. Springer Berlin Heidelberg, 409–424. DOI :http://dx.doi.org/10.1007/978-3-642-28872-2_28
- M. Staats, M.W. Whalen, and M.P.E. Heimdahl. 2011a. Better testing through oracle selection (NIER track). In *Proceedings of the 33rd Int'l Conf. on Software Engineering*. 892–895.

- M. Staats, M.W. Whalen, and M.P.E. Heimdahl. 2011b. Programs, Testing, and Oracles: The Foundations of Testing Revisited. In *Proceedings of the 33rd Int'l Conf. on Software Engineering*. 391–400.
- Matt Staats, Michael W. Whalen, Ajitha Rajan, and Mats P.E. Heimdahl. 2010. Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness. In *Proceedings of the Second NASA Formal Methods Symposium*. NASA.
- CAJ Van Eijk. 2002. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19, 7 (2002), 814–819.
- M. Whalen, G. Gay, D. You, M.P.E. Heimdahl, and M. Staats. 2013. Observable Modified Condition/Decision Coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM.
- M.W. Whalen, A. Rajan, M.P.E. Heimdahl, and S.P. Miller. 2006. Coverage metrics for requirements-based testing. In *Proceedings of the Int'l Symposium on Software Testing and Analysis*. 25–36.
- Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), pp. 80–83. <http://www.jstor.org/stable/3001968>
- Y.T. Yu and M.F. Lau. 2006. A Comparison of MC/DC, MUMCUT and Several Other Coverage Criteria for Logical Decisions. *Journal of Systems and Software* 79, 5 (2006), 577–590.

Challenges in Using Search-Based Test Generation to Identify Real Faults in Mockito

Gregory Gay

University of South Carolina, Columbia, SC, USA,
greg@greggay.com

Abstract. The cost of test creation can potentially be reduced through automated generation. However, to impact testing practice, automated tools must be at least as effective as manual generation. The Mockito project—a framework for mocking portions of a system—offers an opportunity to assess the capabilities of test generation tools on a complex real-world system. We have identified 17 faults in the Mockito project, added those to the Defects4J database, and assessed the ability of the EvoSuite tool to detect these faults. In our study, EvoSuite was only able to detect one of the 17 faults. Analysis of the 16 undetected faults yields lessons in how to improve generation tools. We offer these faults to the community to assist in benchmarking future test generation advances.

Keywords: Search-based testing, automated unit test generation, real faults

1 Introduction

Software testing is a notoriously expensive and difficult activity. With the exponential growth in the complexity of software, the cost of testing has only continued to rise. Much of the cost of testing can be traced directly to the human effort required to conduct most testing activities—such as producing test input and expected outputs. However, such effort is often in service of goals that can be framed as *search* problems, and automated through the use of optimization algorithms [1].

Test case generation can naturally be seen as a search problem. There are hundreds of thousands of test cases that could be generated for any particular SUT. From that pool, we want to select—systematically and at a reasonable cost—those that meet our goals and are expected to be fault-revealing [1]. Automated unit test generation tools have become very effective—even covering more code than tests manually constructed by developers [4]. However, to make an impact on testing practice, automated test generation techniques must be *as effective*, if not more so, at detecting faults as human-created test cases [7].

The Mockito project¹ offers an opportunity to assess the capabilities of test generation tools. Mockito is a *mocking framework* for Java unit testing, allowing users to create customized stand-ins (*mock objects*) for classes in a system, permitting testers to isolate units of a system from their dependencies. Rather than performing the functions

¹ <http://mockito.org/>

of the mocked object, the mock instead issues preprogrammed output. Mockito is an essential tool of modern development, and is one of the most used Java libraries [8].

Mockito serves as an interesting benchmark for two reasons. First, it is a complex project. Much of its functionality is, naturally, related to the creation and manipulation of mock objects. The inputs required by many Mockito functions are complex objects—which are difficult for many test case generators to produce [2]. Second, Mockito is a mature project, having undergone eight years of active development. Recent Mockito faults are unlikely to be the simple syntactic mistakes modeled by mutation coverage. Faults that emerge in a mature project are more likely to require specific, difficult to trigger, combinations of input and method calls. If a test generation tool can detect such faults, then it is likely ready for real-world use. If not, then by studying these faults—and others like them—we may be able to learn lessons that will improve these tools.

We have identified 17 real faults in the Mockito project, and have added them to the Defects4J fault library [6]. We generated test suites using the search-based EvoSuite generation framework [3], and measured the suites' ability to cover the affected classes and detect each fault. EvoSuite was only able to detect one of the 17 faults discovered in the project. Some of the issues preventing fault detection include poor guidance for the fitness function, the need for complex input to methods and object constructors, specific environmental configurations and factors, uncertainty in which classes to generate tests for, and simplistic handling of interface changes between software versions. We have made this set of Mockito faults available to provide data and examples for benchmarking future test generation advances.

2 Study

Recent studies have assessed the capabilities of test generation tools on faults in open-source projects [7], but more data is needed to understand where such tools excel and where they need to be improved. In this study, we have generated tests using the search-based EvoSuite framework [3] on classes of the Mockito project. In doing so, we wish to answer the following research questions:

1. Can EvoSuite detect faults found in Mockito?
2. What factors prevented EvoSuite from detecting faults?

In order to answer these questions, we have performed the following experiment:

1. **Derived Faults:** We have identified 17 real faults in the Mockito project, and added them to the Defects4J fault database (See Section 2.1).
2. **Generated Test Cases:** For each fault, we generated tests on the fixed version of fault-affected classes. (See Section 2.2).
3. **Removed Non-Compiling Tests:** Any tests that do not compile or that fail on the fixed system are automatically removed (See Section 2.2).
4. **Assessed Fault-finding and Coverage:** For each suite and fault, we measure the number of tests that pass on the fixed version and fail on the faulty version. We also record the achieved code coverage.
5. **Analyzed Faults That Were Not Detected:** For each undetected fault, we examined the report and source code to identify possible detection-preventing factors.

2.1 Fault Extraction

Using Mockito’s version control and issue tracking systems, we have identified 17 faults. Each fault is required to meet three properties. First, the fault must be related to the source code. For each reported issue, we attempted to identify a pair of code versions that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix to the fault must be isolated from unrelated code changes such as refactorings.

In order to focus on the faults typical of a mature project, we limited our extraction to the GitHub-based issue tracking system that Mockito began using in July 2014 (previously, Google Code was used). To help identify candidate faults, we used automation provided by Defects4J [6]—a library of faults from five open-source Java programs and tools for assessing tests intended to find such faults.

We have added Mockito as a sixth Defects4J project. This consisted of developing build files that work across project versions, extracting candidate faults, ensuring that each candidate could be reliably reproduced, and minimizing the “patch” used to distinguish fixed and faulty classes. Following this process, we extracted 17 faults from a pool of 89 candidate faults. Six of the 17 faults were “false-positives”, fixes to issues reported in the old issue tracker that shared an issue ID with issues in the newer tracking system. As these six faults met reasonable system maturity and complexity thresholds, we also added them to Defects4J.

The faults used in this study can be accessed by cloning the `bug-mining` branch of <https://github.com/Greg4cr/defects4j>. Additional data about each fault can be found at <http://greggay.com/data/mockito/mockitofaults.csv>, including commit IDs, fault descriptions, and a list of triggering tests. We plan to add additional faults and improvements in the future.

2.2 Test Generation and Removal

EvoSuite applies a genetic algorithm in order to evolve test suites over several generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [7]. In this study, we used EvoSuite version 1.0.3 with the default fitness function—a combination of branch, context branch, line, exception, weak mutation, method-output, top-level method, and no-exception top-level method coverage. Given the potential difficulty in achieving coverage over Mockito classes, the search budget was set to 10 minutes. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 30 trials for each fault by generating tests for the classes patched to fix the fault.

Tests are generated from the fixed version of the system and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario. Due to changes introduced to fix faults, such as altered method

ID	Fault Detected	# Tests Generated	# Tests Removed	% LC	% BC	% EC	% WMC	% OC	% MC	% MNEC	% CBC	Resulting % Coverage
1	X	4.23	0.00	10.00	7.00	100.00	2.00	2.00	25.00	8.00	3.00	20.00
2	✓	92.00	1.00	86.97	87.95	100.00	62.85	50.00	100.00	50.50	87.95	72.47
3	X	4.31	0.00	10.00	8.00	100.00	2.00	2.00	25.00	8.00	3.00	20.00
4	X	84.70	0.00	73.67	85.33	100.00	24.50	0.00	100.00	1.00	85.33	46.67
5	X	15.03	0.00	61.80	77.63	98.90	77.00	100.00	100.00	100.00	77.63	87.00
6	X	60.13	0.00	100.00	100.00	100.00	100.00	44.50	100.00	100.00	100.00	93.00
7	X	14.82	0.00	12.86	20.86	92.86	12.82	0.00	10.00	0.00	20.86	26.00
8	X	14.97	0.00	12.97	20.93	100.00	12.93	0.00	10.00	0.00	20.93	26.00
9	X	1.00	0.00	33.00	33.00	100.00	0.00	0.00	100.00	50.00	33.00	38.00
10	X	2.00	0.00	8.00	10.00	100.00	0.00	0.00	100.00	33.00	10.00	24.00
11	X	1.00	0.00	6.00	12.00	100.00	20.00	0.00	10.00	0.00	12.00	20.67
12	X	1.00	0.00	12.00	11.00	100.00	0.00	0.00	100.00	50.00	11.00	29.00
13	X	10.07	0.00	45.90	59.78	100.00	25.00	67.00	100.00	75.00	59.77	62.93
14	X	41.89	4.63	81.59	83.52	93.48	67.81	62.63	99.84	83.96	83.26	80.02
15	X	16.70	7.37	65.36	64.09	92.48	55.42	50.56	85.56	80.97	64.09	64.00
16	X	73.90	7.57	86.43	84.91	80.68	83.33	38.68	100.00	77.43	84.41	71.36
17	X	35.50	3.43	99.04	97.43	95.21	94.91	57.50	100.00	100.00	97.43	91.03

Table 1. Average test generation results for each fault—whether the fault was detected, number of generated tests, number of non-compiling tests, line coverage (LC), branch coverage(BC), exception coverage (EC), weak mutation coverage (WMC), method-output coverage (OC), method coverage (MC), no-exception top-level method coverage (MNEC), context branch coverage (CBC), and the resulting average across all coverage metrics.

signatures or new classes, some tests may not compile on the faulty version of the system. We have automatically removed such tests. We have also removed tests that fail on the fixed version of the system, as these do not assist in identifying faults. On average, 4.48% of the tests are removed from each suite. More statistics are included in Table 1.

3 Results and Discussion

The results of our experiment can be seen in Table 1. In our study, only one of the 17 faults was detected—Fault 2. This particular fault—revolving around incorrect handling of negative time values—is an excellent example of the kind of fault that automated test generation is able to handle. The code fix adds conditional behavior to handle time input. By covering the new branches, the tests are guided to detect the fault in all 30 trials. However, EvoSuite failed to detect the other 16 faults. Therefore, our next step was to examine these faults to identify factors preventing detection. These factors include:

Poor Guidance for the Fitness Function: While EvoSuite is often able to achieve reasonable levels of coverage across Mockito classes, coverage is sometimes quite low. While coverage does not guarantee fault detection, unexecuted code cannot reveal faults [5]. One reason coverage may not be achieved is that the code offers no guidance to the search tool in selecting *better* test suites.

Many fitness functions are designed to measure the distance from optimality of generated test cases. However, it is not always obvious how to calculate this distance. The code that must be covered to detect Fault 12² provides a good example. Both branches use the `instanceof` operator. Without a method of determining the “distance” between class types, the search devolves into a random search.

² <https://github.com/mockito/mockito/commit/7a647a702c8af81ccf5d37b09c11529c6c0cb1b7>

Complex Input is Required to Trigger a Fault: A challenge for test generation techniques is generating inputs of complex data types [2]. As Mockito generates objects that mimic other objects, many of its methods require complex objects as input. Even in cases where coverage is high, test generators may have difficulty producing the intricate, highly-specific, input required to detect that fault.

Consider Fault 13³, which occurs when Mockito’s verification capabilities are invoked on a method call that, itself, has an embedded method call within it. Triggering this fault requires generating two different mock objects, then embedding a call to one object within a call to the second. Coverage alone is unlikely to suggest such input. Rather, fitness functions that incorporate domain expertise may be needed to help generate more complex input scenarios. Promising work has been conducted using grammars to produce complex input [2].

Complex Input is Required to Generate Any Tests: Unit tests instantiate an object and call the methods offered by that object. At times, objects must be provided with input when they are instantiated (there is no “default” constructor). Many of the code changes made to fix Fault 3⁴ are contained within one method. EvoSuite not only fails to fully cover this method, it fails to invoke this method at all. In this case, EvoSuite attempts to instantiate the `InvocationMaster` class, but many of these attempts fail due to invalid input. EvoSuite cannot cover the methods of an object that it cannot instantiate.

Faults Require Specific Environmental Factors: Fault 5⁵ revolves around an undesired dependency on the JUnit framework. Fixing this fault requires code changes—yet, coverage of this code will not reveal this fault. Rather, the fault is detected when JUnit is removed from the local classpath. This is an example of a fault that depends on environmental factors—in this case, the classpath used to compile code. EvoSuite does manipulate certain environmental factors, such as file system access, but more examination of such factors is needed in future test generation research.

Fault Detection Requires Generating Tests for Related Classes: The classes affected by Fault 6⁶ offer another interesting example. Mock objects can be configured to return different values based on the type of function input. Due to this fault, a mock can produce a value intended for certain data types when a `null` object is passed instead of the intended type. The fault-fixing changes are primarily in methods that do not require input—methods that are called by Mockito’s argument matchers. Because these methods do not require input, this fault cannot be detected without generating tests for the argument matcher classes that, in turn, call these methods. Under normal circumstances, EvoSuite could produce the required `null` input, but tests would need to be generated for classes that do not contain faulty code, and instead depend on faulty code. Some consideration should be given to which classes are used when generating tests, and the dependencies between those classes.

Changes to Code Invalidate Test Cases: When tests are generated on one version of a system and applied to another, code changes such as the addition of new classes or altered method signatures can result in tests that do not compile on one version. In

³ <https://code.google.com/archive/p/mockito/issues/138>

⁴ <https://github.com/mockito/mockito/commit/3eec7451d6c83c280743c39b39c77a179abb30f9>

⁵ <https://github.com/mockito/mockito/issues/152>

⁶ <https://github.com/mockito/mockito/commit/dc205824dbc289acbcde919e430176ad72da847f>

this study, we removed those tests. This may prevent fault detection. Fault 17⁷ affects the ability to set mock objects as serializable. EvoSuite is correctly guided to create serializable mock objects. However, any time this occurs, interactions take place with a new class. These tests are removed, as they do not compile on the faulty version of the system. In normal practice, this is not an issue, as tests are generated on the version they are applied to, but during regression testing, similar issues may occur. Intelligent strategies are needed to generate tests that compile across multiple versions of systems.

4 Conclusion

The capabilities of test generation techniques have increased. Yet, from the examples extracted from the Mockito project, we can see that there are still fault-detection hurdles to overcome. EvoSuite was only able to detect one of the 17 faults. Some of the issues preventing fault detection include poor guidance for the fitness function, the need for complex input to methods and object constructors, environmental factors, uncertainty in which classes to generate tests for, and simplistic handling of interface changes between multiple software versions. We hope that the set of faults extracted from Mockito will provide data and examples for benchmarking new test generation advances.

References

1. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on* 36(6), 742–762 (2010)
2. Feldt, R., Pouling, S.: Finding test data with specific properties via metaheuristic search. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 350–359 (Nov 2013)
3. Fraser, G., Arcuri, A.: Whole test suite generation. *Software Engineering, IEEE Transactions on* 39(2), 276–291 (Feb 2013)
4. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 291–301. ISSTA 2013, ACM, New York, NY, USA (2013), <http://doi.acm.org/10.1145/2483760.2483774>
5. Gay, G., Staats, M., Whalen, M., Heimdalh, M.: The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on* PP(99) (2015)
6. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), <http://doi.acm.org/10.1145/2610384.2628055>
7. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). ASE 2015, ACM, New York, NY, USA (2015)
8. Weiss, T.: We analyzed 30,000 GitHub projects - here are the top 100 libraries in Java, JS and Ruby (2013), <http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-javascript-and-ruby/>

⁷ <https://github.com/mockito/mockito/commit/77cb2037314dd024eb53ffe2e9e06304088a2d53>

Summary of the 3rd International Workshop on Requirements Engineering and Testing (RET 2016)

[Co-located with REFSQ 2016]

Michael Unterkalmsteiner
Software Engineering Research
Lab Sweden
Blekinge Institute of Technology
Karlskrona, Sweden
mun@bth.se

Elizabeth Bjarnason
Lund University
Lund, Sweden
elizabeth.bjarnason@cs.lth.se

Gregory Gay
University of South Carolina
Columbia, SC, USA
greg@greggay.com

Markus Borg
SICS Swedish ICT AB
Lund, Sweden
markus.borg@sics.se

Michael Felderer
University of Innsbruck
Innsbruck, Austria
michael.felderer@uibk.ac.at

Mirko Morandini
Fondazione Bruno Kessler
Trento, Italy
morandini@fbk.eu

DOI: 10.1145/2934240.2934249

ABSTRACT <http://doi.acm.org/10.1145/2934240.2934249>
The RET (Requirements Engineering and Testing) workshop series provides a meeting point for researchers and practitioners from the two separate fields of Requirements Engineering (RE) and Testing. The goal is to improve the connection and alignment of these two areas through an exchange of ideas, challenges, practices, experiences and results. The long term aim is to build a community and a body of knowledge within the intersection of RE and Testing, i.e. RET. The 3rd workshop was held in co-location with REFSQ 2016 in Gothenburg, Sweden. The workshop continued in the same interactive vein as the predecessors and included a keynote, paper presentations with ample time for discussions, and panels. In order to create an RET knowledge base, this cross-cutting area elicits contributions from both RE and Testing, and from both researchers and practitioners. A range of papers were presented from short positions papers to full research papers that cover connections between the two fields.

RET 2016 accepted technical papers with a maximum length of 15 pages presenting research results or industrial practices and experiences related to the coordination of RET, as well as position papers with a maximum length of 6 pages introducing challenges, visions, positions or preliminary results within the scope of the workshop. Experience reports and papers on open challenges in industry were especially welcome.

RET 2016 accepted four technical papers and one position paper. The workshop was visited by 17 participants and the proceedings are available online [2].

2. ORGANIZATION

The 3rd International Workshop on Requirements Engineering and Testing (RET 2016) was held on March 14, 2016, and was co-located with the 22nd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016). The website for the workshop is available online¹. The workshop was organized by Michael Unterkalmsteiner (Blekinge Institute of Technology) as general chair, Gregory Gay (University of South Carolina) and Michael Felderer (University of Innsbruck) as program co-chairs, as well as, Elizabeth Bjarnason (Lund University), Markus Borg (SICS Swedish ICT AB) and Mirko Morandini (Fondazione Bruno Kessler) as co-chairs.

3. PROGRAM SUMMARY

The program of RET 2016 comprised of an introductory part with a keynote, a discussion on the past and future of RET, and two paper presentation sessions followed by panels with the paper presenters.

After a welcome note, Baldwin Gislason Bern (R&D Expert at Axis Communications AB) gave an invited talk entitled “Tests as requirements - Why we don’t do requirements at Axis”. He illustrated why and how Axis, a company with approximately 100 different products, can be successful with a team of 120 test engineers and very little resources dedicated at requirements engineering and management. A major factor is the technology-driven domain the company is operating in, allowing product managers to push new technologies to the market, collecting data from customers and then acting upon this feedback to improve future iterations of

¹<http://ret.cs.lth.se/16>

Categories and Subject Descriptors

D.2.1 [Requirements / Specifications]; D.2.4 [Software / Program Verification]; D.2.5 [Testing and Debugging]

General Terms

Management, Documentation, Human Factors, Verification

Keywords

requirements engineering, testing, coordination, alignment

1. INTRODUCTION

The main objective of the RET workshop series is to explore, characterize and understand the interaction of Requirements Engineering (RE) and Testing, both in research and industry, and the challenges that result from this interaction. The workshop provides a forum for exchanging experiences, ideas and best practices to coordinate RE and testing. A primary goal of this exchange is to enable and provide incentives for research that crosses research areas and is relevant for industry. Towards this end, RET invites submissions exploring how to coordinate RE and Testing, including practices, artifacts, methods, techniques and tools. Submissions on softer aspects like the communication between roles in engineering processes are also welcome.

the product. The key take-way idea from the presentation is that knowledge within a company on a technology/product/market changes over time, requiring flexible strategies for product development. With a novel technology, little knowledge and experience exist and requirements are unclear. The role of testing is to explore limits to gain knowledge. Therefore, test cases document decisions, as opposed to requirements which would document intentions. Knowledge that stems from testing the product is documented in test cases, rendering them a living documentation that is actively used and maintained over time. As the knowledge on technology in Axis matures and to be able to maintain a competitive edge, the company is adapting its strategy towards value driven product development. This will put customer feedback in the center, and, as a new source of requirements, drive product development.

The keynote was followed by a panel on future industry needs with respect to coordinating requirements engineering and testing. To kick-off the discussion, the workshop chair presented a thematic summary of the three instances of workshop in 2014, 2015 and 2016, generating a topic model from the 23 abstracts that were accepted and presented in total at the workshop. He illustrated the predominant themes with Serendip [1], a visualization tool for topic models (see Figure 1). The rows represent the papers presented at RET in three years. The columns represent the identified topics². The size of the circle on the crossing between article and topic represents the probability that the document was generated by the terms that represent the respective topic. The predominant topics at the respective workshop instances were:

- RET 2014: Tools, Security Requirements, System Testing, Experience, Development and Models
- RET 2015: Test Design, Testers, Quality
- RET 2016: Language, Quality, Artifacts and Data

This result suggests that the three workshops were driven by different themes, quality being a commonality between 2015 and 2016. In Figure 1, the topics are ordered from left to right, by the total proportion. The “tool” topic predominates, followed by “testing experience” and “requirements model”. This result suggests that the accepted papers are thematically in line with the goals of RET (see Section 1), at least from the perspective of the used vocabulary in the abstracts. Based on these results and the preceding keynote, the panelists consisting of two researchers and three industry participants discussed future research avenues for RET. A common interest seemed to be the effective use of customer feedback to drive product development. This approach is quite feasible as one panelist from the mobile app domain observed, may however be impractical in other domains such as the automobile industry where feedback cycles with customers are more difficult to establish.

The panel was followed by the first paper session themed quality requirements. The talk “Testing Quality Requirements of a System-of-Systems in the Public Sector - Challenges and Potential Remedies” identified five main challenges when testing quality requirements: (1) evolving RE documents while testing is planned

²The number of topics, 10, is a required parameter when generating the model and was set rather arbitrarily. However, 10 topics seemed to be enough to provide some differentiation between papers and not too much to be too fine-grained. Most of the topics were rather easy to label, based on the most frequent terms per topic.

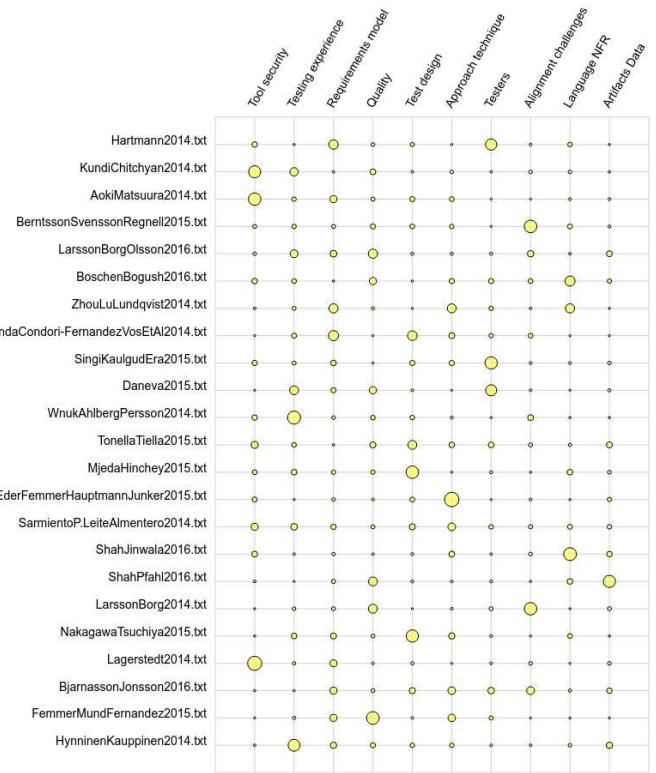


Figure 1: Topic model of the papers accepted at RET 2014-2016

and ongoing, (2) test managers need to understand the business side of the company, (3) quality requirements are not quantified or (4) prioritized, (5) difficulty to generate test data that exercises all operational states. These challenges were matched with solution proposals from the scientific literature.

The talk “Evaluating and Improving Software Quality Using Text Analysis Techniques - A Mapping Study” identified 81 primary studies. The most frequent application of text analysis techniques was in defect management (bug classification and severity assignment), followed by requirements engineering (concept extraction). The most common data sources are bug reports and requirement documents. Interesting avenues for future research are to study mobile app reviews to understand software quality and to combine multiple data sources.

The talk “Specification of Non-Functional Requirements: A Hybrid Approach” proposed an approach to identify NFRs in natural language requirements using text processing techniques and ontologies, which are then modeled as use cases. The approach has been illustrated in a case study with a proof-of-concept example.

The second paper session comprised of a position paper and a full technical paper. The talk “Improving Project Coordination through Data Mining and Proximity Tracking” proposed to analyze what project members work on and to support direct interaction based on common work related themes when individuals meet. Proximity tracking would also allow to identify communication patterns that could provide insights who collaborates with whom and when.

The talk “Bridging the Gap between Natural Language Requirements and Formal Specifications” proposed to use requirements

boilerplates to support the formalization process of natural language requirements. The industry case study conducted at Airbus illustrated how a requirements quality tool was used to extract semantics from boilerplates to semi-automatically generate formal requirements specifications.

4. FUTURE

We plan to organize the workshop again next year since the topic attracted interested from both industry and academia. Our aim is to organize RET 2017 co-located with the 25th International Requirements Engineering Conference (RE 2017) in Portugal. If the workshop is accepted, the expected date for paper submissions is in June 2017.

5. ACKNOWLEDGMENTS

We want to thank the participants of the workshop and all the authors of submitted papers for their important contribution to the event. In addition, we want to thank the organizers of the 22nd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016) and the members of the program committee:

Armin Beer, Beer Test Consulting, Austria
Ruzanna Chitchyan, Leicester University, UK
Nelly Condori-Fernandez, PROS Research Centre, Spain
Robert Feldt, Blekinge Institute of Technology, Sweden
Henning Femmer, Technische Universität München, Germany
Vahid Garousi, Hacettepe University, Turkey
Joel Greenyer, University of Hannover, Germany
Andrea Herrmann, Herrmann & Ehrlich, Germany
Mike Hinckey, Lero - the Irish Software Engineering Research Centre, Ireland
Jacob Larsson, Capgemini, Sweden
Annabella Loconsole, Malmö University, Sweden
Alessandro Marchetto, FIAT research, Italy
Cu Duy Nguyen, University of Luxembourg, Luxembourg
Magnus C. Ohlsson, System Verification, Sweden
Barbara Paech, University of Heidelberg, Germany
Dietmar Pfahl, University of Tartu, Estonia
Sanjai Rayadurgam, University of Minnesota, USA
Giedre Sabaliauskaitė, Singapore University of Technology and Design, Singapore
Hema Srikanth, IBM, USA
Marc-Florian Wendland, Fraunhofer FOKUS, Germany
Dongjiang You, University of Minnesota, USA
Yuanyuan Zhang, University College London, UK

6. REFERENCES

- [1] E. Alexander, J. Kohlmann, R. Valenza, M. Witmore, and M. Gleicher. Serendip: Topic model-driven visual exploration of text corpora. In *Proceedings Conference on Visual Analytics Science and Technology (VAST)*, pages 173–182, Paris, France, 2014. IEEE.
- [2] E. Bjarnason, M. Borg, O. Dieste, S. España, M. Felderer, P. Forbrig, X. Franch, G. Gay, A. Hermann, J. Horkoff, M. Kirikova, M. Morandini, A. L. Opdahl, B. Paech, C. Palomares, K. Petersen, A. Seffah, B. Tenbergen, and M. Unterkalmsteiner. Joint proceedings of the REFSQ 2016 co-located events. Online: <http://ceur-ws.org/Vol-1564/>.

Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors

Gregory Gay, Sanjai Rayadurgam, Mats P.E. Heimdahl, *Senior Member, IEEE*

Abstract—The *test oracle*—a judge of the correctness of the system under test (SUT)—is a major component of the testing process. Specifying test oracles is challenging for some domains, such as real-time embedded systems, where small changes in timing or sensory input may cause large behavioral differences. Models of such systems, often built for analysis and simulation, are appealing for reuse as test oracles. These models, however, typically represent an *idealized* system, abstracting away certain issues such as non-deterministic timing behavior and sensor noise. Thus, even with the same inputs, the model’s behavior may fail to match an acceptable behavior of the SUT, leading to many false positives reported by the test oracle.

We propose an automated *steering* framework that can adjust the behavior of the model to better match the behavior of the SUT to reduce the rate of false positives. This *model steering* is limited by a set of constraints (defining the differences in behavior that are acceptable) and is based on a search process attempting to minimize a dissimilarity metric. This framework allows non-deterministic, but bounded, behavioral differences, while preventing future mismatches by guiding the oracle—with limits—to match the execution of the SUT. Results show that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing false positives and, consequently, testing and debugging costs while improving the quality of the testing process.

Index Terms—Software Testing, Test Oracles, Model-Based Testing, Model-Based Development, Verification

1 INTRODUCTION

When running a suite of tests, the *test oracle* is the judge that determines the correctness of the execution of a given system under test (SUT). Over the past decades, researchers have made remarkable improvements in automatically generating effective test stimuli [1], but it remains difficult to build an automated method of checking behavioral correctness. Despite increased attention, the *test oracle problem* [2]—the set of challenges related to the construction of efficient and robust oracles—remains a major problem in many domains.

One such domain is that of real-time embedded systems—especially those that interact with a physical domain such as implanted medical devices. Systems in this domain are particularly challenging since their behavior depends not only on the values of inputs and outputs, but also on their time of occurrence [3]. When executing the software on an embedded hardware platform, several sources of non-determinism, such as input processing delays, execution time fluctuation, and hardware inaccuracy, can result in the SUT exhibiting non-deterministic—but acceptable—behaviors.

Behavioral models [4], typically expressed as state-transition systems, represent the system specifications by prescribing the behavior (the system state) to be exhibited in response to given input. Common modeling tools in this category are Stateflow [5], Statemate [6], and Rhapsody [7].

G. Gay is with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: greg@greggay.com

S. Rayadurgam and M. Heimdahl are with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: [rsanjai, heimdahl]@cs.umn.edu

This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

Models built using these tools are used for many purposes in industrial software development, particularly during requirements and specification analysis. Behavior modeling is common for the analysis of embedded and real-time systems, as the requirements for such systems are naturally *stateful*—their outcome depends strongly on the current system mode and a number of additional factors both internal and external to the system. Because such models can be “executed”, a potential solution to the need for a test oracle is to execute the same tests against both the model and the SUT and compare the resulting behaviors.

These models, however, provide an abstract view of the system that typically simplifies the actual conditions in the execution environment. For example, communication delays, processing delays, and sensor and actuator inaccuracies may be omitted. Therefore, on a real hardware platform, the SUT may exhibit behavior that is acceptable with respect to the system requirements, but differs from what the model prescribes for a given input; the system under test is “close enough” to the behavior described by the model. Over time, these differences can build to the point where the execution paths of the model and the SUT diverge enough to flag the test as a “failure,” even if the system is still operating within the boundaries set by the requirements. In a rigorous testing effort, this may lead to tens of thousands of false reports of test failures that have to be inspected and dismissed—a costly process.

We take inspiration for addressing this model-SUT mismatch problem from *program steering*, the process of adjusting the execution of live programs in order to improve performance, stability, or correctness [8]. We hypothesize that behavioral models can be adapted for use as oracles for real-time systems through the use of *steering actions* that override

the current execution of the model [9], [10]. By comparing the state of the model-based oracle (MBO) with that of the SUT following an output event, we can guide the model to match the state of the SUT through a search process that seeks a steering action that transitions the model to a reachable state that obeys a set of user-specified constraints and general steering policies and that minimizes a dissimilarity metric. The result of steering is a widening of the behaviors accepted by the oracle, thus compensating for allowable non-determinism, without unacceptably impairing the ability of the model-based oracle to correctly judge the behavior of the SUT.

We present an automated framework for oracle steering, expanding on previous research [10], [11]. In this manuscript, we present an updated steering framework, discuss the theory, current implementation, and implications of oracle steering in detail, and present a case study where we examine the effectiveness of steering on two systems with complex, time-based behaviors—the control software of a patient controlled analgesia pump (a medical infusion pump) and a pacemaker. We also present an automated method for learning the constraints that guide the steering process.

Case study results indicate that steering improves the accuracy of the final oracle verdicts—outperforming both default testing practice and a filtering technique. Oracle steering successfully accounts for within-tolerance behavioral differences between the model-based oracle and the SUT—eliminating a large number of spurious “failure” verdicts—with minimal masking of real faults. By pointing the developer towards behavior differences more likely to be indicative of real faults, this approach has the potential to reduce testing effort and reduce development cost.

Our results indicate that the choice of constraints limiting the choice of steering actions has a major impact on the ability of steering to account for allowable non-determinism. Relatively strict, well-considered constraints strike the best balance between the ability to account for non-determinism and the risk of masking faults. As constraints are loosened, steering may be able to account for more acceptable deviations, but it will often mask more faults or even choose suboptimal steering actions that cause undesired side-effects.

For developers are unsure of what constraints to employ, we offer a technique that can automatically learn a set of constraints through a process known as *treatment learning*. Given a set of developer-classified test cases, we can extract information on the steering actions chosen when no constraints are employed, and apply a treatment learner to identify the steering actions highly correlated to correct test verdicts. We can then apply these learned constraints when steering for a broader set of test executions. For our case examples, the derived constraint sets were small, strict, and able to successfully steer the model with only minimal tuning.

We have found that steering is able to automatically adjust the execution of the oracle to handle non-deterministic, but acceptable, behavioral divergence without covering up most fault-indicative behaviors. We recommend the use of steering as a tool for focusing and streamlining the testing process.

2 BACKGROUND

There are two key artifacts necessary to test software, the *test data*—inputs given to the system under test—and the *test oracle* [12], [13]. A *test oracle* is a predicate that judges the resulting behavior according to some specification of correctness [2].

The most common form of test oracle is a *specified oracle*—one that judges behavioral aspects of the system under test with respect to some formal specification [2]. Commonly, such an oracle checks the behavior of the system against a set of concrete expected values [14] or behavioral constraints (such as assertions, contracts, or invariants) [15]. However, specified oracles can be derived from many other sources of information; we are particularly interested in using behavioral models, such as those often built for purposes of simulation, analysis and testing [4].

Although behavioral models are useful at all stages of the development process, they are particularly effective in addressing testing concerns. Models allow testing activities to begin before the actual implementation is constructed, and models are suited to the application of verification and automated test generation techniques [16]. As models are often executable, in addition to serving as the basis of test generation [4], models can be used as a source of expected behavior—as a *test oracle*.

An example of such a model can be seen in Figure 1. This model, written in the Stateflow notation [5], represents the behavior of a simplified pacemaker—a medical device that regulates the heart rate of a patient by issuing electrical stimuli (paces) in the absence of natural activity. This pacemaker regulates a ventricle by polling sensors every millisecond (timestamped with *timeIn*) for a sensed event (*sense*), and issuing paces (*pace*) as regulated by the Lower Rate Limit (*LRL*)—a user-specified desired pace per minute rate. Paces are timestamped using the variable *timeOut*. Following a sense or pace, the pacemaker will ignore all sensor values for a user-specified length of time—*VRP*, or the Ventricular Refractory Period—to avoid acting on electrical noise. We will refer to this example throughout this work as SimplePacing.

Non-determinism is a major concern in embedded time-based systems. The task of monitoring the environment and pushing signals through layers of sensors, software, and actuators can introduce points of failure, delay, and unpredictability. Input and observed output may be skewed by noise in the physical hardware, timing constraints may not be met with precision, or inputs may arrive faster than the system can process them. Often, the system behavior may be acceptable, even if that behavior is not exactly what was captured in the model—a model that, by its very nature, incorporates a simplified view of the problem domain. It is common when modeling is to omit any details that distract from the core system behavior in order to ensure that analysis of the models is feasible and useful.

As a simple example, in this model, the time that the sensor is polled is the same as the time that output is issued (*timeIn* = *timeOut*). This is unlikely to be true in the actual software—differences may arise from computation time, clock drift, and the difficulty of synchronizing the paral-

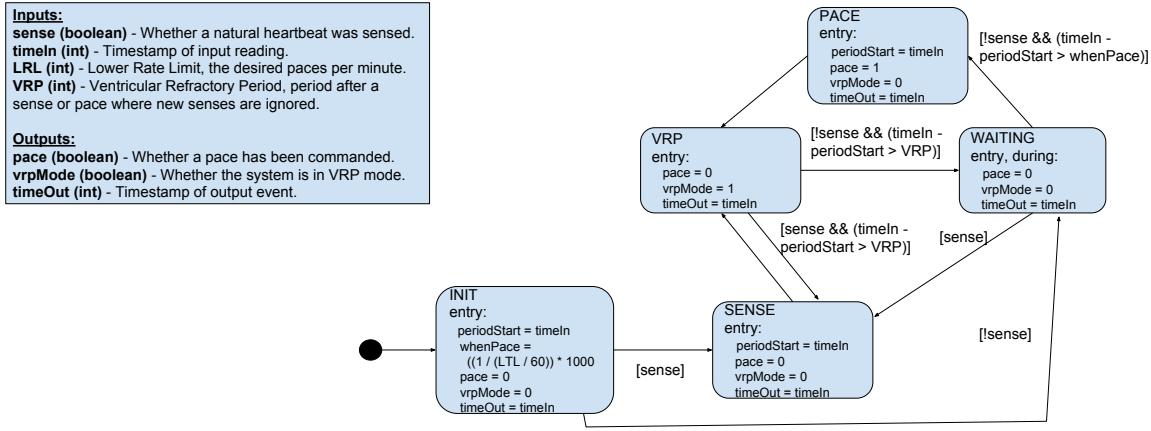


Fig. 1: SimplePacing model—a system that delivers paces (electrical impulses) at a prescribed rate in the absence of sensed ventricular activity.

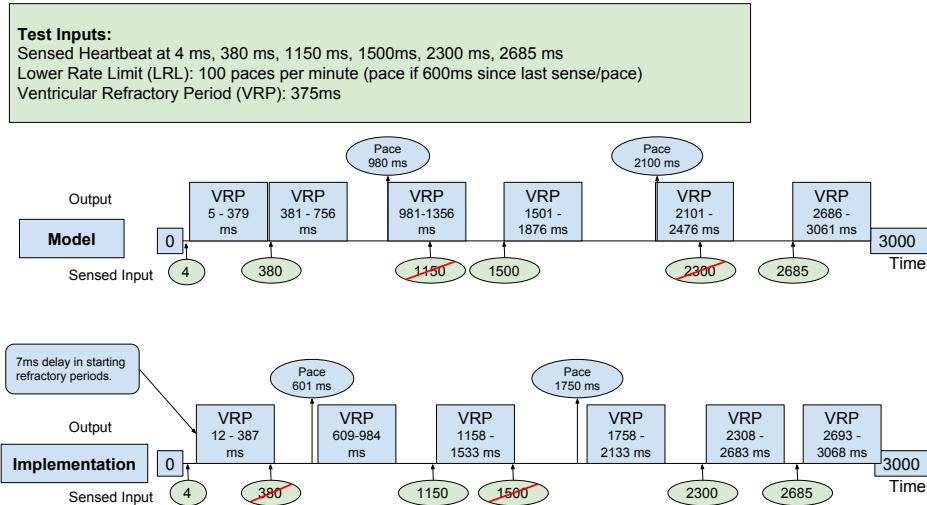


Fig. 2: Abstraction-induced behavioral differences between SimplePacing’s model and implementation during execution of a test case. The Lower Rate Limit describes the minimum number of paces per minute to be issued by the pacemaker. In the absence of a sensed heartbeat, a pace will be delivered every 600 ms. The Ventricular Refractory Period is a length of time after a sense or pace where further senses are ignored, as they might be aftershocks or other noise from the last measured activity.

lel components of the software—but assuming instantaneous computation time (or a constant computation time) is a common abstraction when modeling. The SimplePacing model receives a simple binary sense. However, in the real world, the electrical impulses being sensed are complex noise-prone analog readings that the system must decide how to interpret. These omitted or simplified details may manifest themselves as differences between the behavior defined in the model and the behavior observed in the implementation during test execution. Furthermore, such behaviors are commonly non-deterministic. Repeated application of the same test stimulus may not result in the same output if, say, processing time varies.

An example of this is illustrated in Figure 2, where the behavior of the SimplePacing model and implementation differ during test execution. In this case, the difference is simple—minor computation delays result in a situation where there is a 7ms delay in starting a Ventricular Refractory Period

following a sense or pace. The developers of systems often recognize the likelihood of such scenarios and prescribe a *tolerance period*, a bounded period of time where the activity is still legal, in the specification. In this case, they might declare that the SUT is correct as long as the start and end of the VRP falls in a +/- 8ms window of time of when it is expected to occur. In fact, this is the actual tolerance prescribed in a publicly available pacemaker specification [17]. In this case, the implementation—despite the small delay—initiates a VRP within the required window of time. However, as the comparison procedure expects the model and system to conform, the test will fail.

If this were the sole difference between the model and system’s behavior, it would be easy enough to adjust the model or comparison to account for this difference. However, the behavior of SimplePacing—and many critical embedded or cyber-physical systems—is *stateful*. The behavior at one

step of execution is dependent on the actions and reactions of the system at all previous execution steps. As can be seen, execution over the entire life of the test case differs greatly between the two systems. Because of the delay in starting the VRP, the input at 380 ms is ignored by the implementation. The model receives this input 5 ms after its VRP ends, and it starts a new VRP. However, because that input was ignored, the implementation issues a pace at 601 ms, while the model does not do so until 980 ms. In both cases, the behavior exhibited by that artifact is “correct” with respect to the specifications, but the two diverge significantly as execution proceeds. In this case, the source of the divergence is clear, and the implementation is consistent in how it acts. In practice, such issues are often more complex—the implementation might vary when the VRP starts and ends non-deterministically, it might react to a sense later than intended, or it might issue a pace either earlier or later than expected.

This raises the question—why use models as oracles? Alternative approaches could be to turn to an oracle based on explicit behavioral constraints—assertions or invariants—or to build declarative behavioral models in a formal notation such as Modelica [18]. These solutions, however, have their limitations. Assertion-based approaches only ensure that a limited set of properties hold at *particular points in the program* [15]. In some cases, models can more easily account for a wider range of input scenarios. Xie and Memon found that oracles that incorporate state information (such as models) can have higher fault-detection capabilities than simpler oracles—even when shorter test cases are used [19]. Declarative models that express the computation as a theory in a formal logic allow for more sophisticated forms of verification and can potentially account for some forms of non-deterministic behavior [20]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [21]. Constructive models are easier to analyze without specialized knowledge and suitable for analyzing failure conditions and events in an isolated manner [20]. The complexity of declarative models and the knowledge needed to design and interpret such models make widespread industrial adoption of the paradigm unlikely.

More importantly, it is a widely held view that constructive models are indispensable in other areas of development and testing, such as requirements analysis or automated test generation [16], [22]. From this standpoint, the motivational case for models as oracles is clear. If these models are already being built—and are useful throughout development and testing—their reuse as oracles could save significant amounts of time and money, and allow developers to automate the execution and analysis of a large volume of test cases. For these reasons, model-based approaches have become common in the development of critical systems [23]—systems that are particularly likely to demonstrate the type of non-determinism discussed previously. Therefore, we seek a way to use constructive model-based oracles even when faced with non-determinism introduced during system execution on the target hardware.

3 ORACLE STEERING

We would like to distinguish between correct, but non-conforming, and fault-indicative behaviors when using a model-based oracle. A simple approach that could potentially address this would be to augment the comparison with a filter that detects and ignores acceptable differences on a per-step basis. For example, if the model and SUT produce behavior that differs only by the timestamp, the filter could allow this difference as long as it falls within a bounded time range. Such filters are relatively common in GUI or graphic rendering testing [24]¹. This is an example of an *indirect* solution—a filter selectively overrides the oracle, but does not interfere with the internal execution of the model.

However, an issue with indirect solutions like filtering is that many of the systems we are interested in not only demonstrate non-determinism, but are stateful. As can be seen in Figure 2, the non-deterministic shift in a single refractory period can influence the execution of the system for the entire test case, leading to irreconcilable differences between the SUT and the model-based oracle. If state has a minimal impact, or the sources of non-determinism are simple and isolated, then a filter is an appropriate solution. If the refractory period is always delayed in the same way, then a filter could simply account for that delay. In practice, however, the beginning and end of that period may change constantly. At the same time, input interactions and other timed events are still occurring that influence system behavior. That shift in the refractory period is likely to be just the first of a series of divergences introduced because the real system is more complex than the model. Indirect actions may not be effective at handling these behavior-impacting events that grow and build off of each other as time progresses. Any potential solution must account for not just a divergence between the SUT and the model-based oracle at a single time step, but with all previous divergences as well.

Rather than indirect actions, such as filtering, we believe that the solution is *direct* action that adjusts the behavior of the model throughout execution to better reflect the details of the actual operating environment of the system. We take inspiration from *program steering*—the process of adjusting the execution of live programs in order to improve performance, stability, or behavioral correctness [25]. Instead of steering the behavior of the SUT, however, we *steer the oracle* to see if the model is capable of matching the SUT’s behavior. When the two behaviors differ, we backtrack and apply a *steering action*—e.g., adjust timer values, apply different inputs, or delay or withhold an input—that changes the state of the model-based oracle to a state more similar to the SUT.

We steer the oracle rather than the SUT because these deviations in behavior result not necessarily from the incorrect functioning of the system, but from a disagreement between the idealized world of the model and the reality of the execution of the system under test. In cases where the system is actually acting incorrectly, we don’t want to steer at all—we want to issue a failure verdict so that the developer can change

¹ Filters could easily be implemented using assertion statements checked following the oracle procedure.

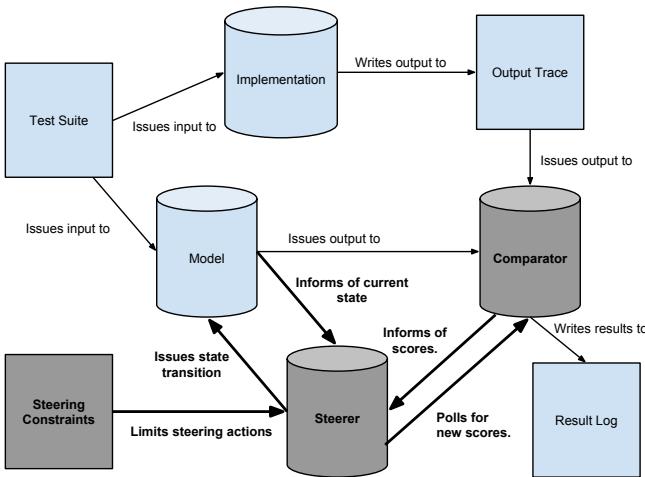


Fig. 3: An automated testing framework employing steering.

```

Input: Model, SUT, Tests
for test ∈ Tests do
1   stepNumber = 0
2   for step ∈ test do
3     previousState = state(Model)
4     applyInput(SUT, step)
5     applyInput(Model, step)
6     Sm = state(Model)
7     Ssut = state(SUT)
8     if Dis(Sm, Ssut) > 0 then
9       instrumentedModel =
10      instrument(Model, previousState)
11      steer(instrumentedModel, Sm, Ssut)
12      Snew = state(Model)
13      if Dis(Snew, Ssut) > 0 then
14        verdict = fail
15        break
16
verdict = pass

```

Fig. 4: Testing process when steering is employed.

the implementation. In many of these deviations, however, it is not the system that is incorrect. If the model does not account for the real-world execution of the SUT, then *the model is the artifact that is incorrect*. Therefore, rather than immediately issuing a failure verdict, we will attempt to correct the behavior of the model.

Each time a divergence occurs, the model is steered to account for the mismatch between model and system. Unlike with indirect solutions, the complexity of handling non-determinism is reduced because the model is guided to adapt to the real-world execution of the system as such events occur. By doing so, steering provides the flexibility to handle non-determinism, while still retaining the power of the oracle as an arbiter. Of course, improper steering can bias the behavior of the model-based oracle, masking both acceptable deviations and actual indications of failures. Nevertheless, we believe that by using a set of appropriate constraints it is possible to sufficiently bound steering so that the ability to detect faults is still retained.

To steer the oracle model, we instrument the model to match the state it was in during the previous step of execution, formulate the search for a new model state as a boolean satisfiability problem, and use a powerful search algorithm to select a target state to transition the model to. This search is guided by three artifacts:

- 1) A set of **tolerance constraints** limiting the acceptable values for the *steering variables*—a set of *model variables* that may be directly manipulated.
- 2) A **dissimilarity function**—a numerical function that compares a candidate model state to the state of the SUT and delivers a score. We seek the candidate solution that minimizes this function.
- 3) A set of generic **policies** that can control steering.

In a typical testing scenario that makes use of model-based oracles, a test suite is executed against both the system under test and the behavioral model. The values of the input, output, and select internal variables are recorded to a *trace file* at certain intervals, such as after each discrete cycle of input and output. Some comparison mechanism examines those trace files and issues a verdict for each test case (a *failure* if any discrepancies are detected and a *pass* if a test executes without revealing any differences between the model and SUT).

Such a framework can be modified to incorporate automated oracle steering. The updated testing process is detailed in Figure 4 and illustrated in Figure 3, with steering-related components shaded in dark gray. In this updated framework, the comparison mechanism issues a dissimilarity score instead of a boolean verdict. If the output does not match, the steering algorithm will instrument the model and attempt to find an action that minimizes that score. If the model and SUT cannot be aligned, the comparator will log the final dissimilarity score.

3.1 System Model

In the abstract, we define a model as a transition system $M = (S, S_0, \Sigma, \Pi, \rightarrow)$, defined as:

- S is a set of states—assignments of values to system variables—with initial state S_0 .
- Σ is an input alphabet, defined as a set of input variables for the model.
- Π is a specially-defined *steering alphabet*— $\Pi \subseteq \Sigma$ —a set of *steerable variables*—the variables that the steering procedure is allowed to directly control and modify the assigned values of.
- \rightarrow is a transition relation (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$.

Any model format that can be expressed as such an M , in theory, can be the target of a steering procedure. In this work, our models are written in the Stateflow notation from Mathworks [5]. When steering, we translate the Stateflow models to the Lustre synchronous programming language to more easily perform automated transformations (see Section 5.6).

The primary difference of this definition from a standard state-transition system is the steering alphabet, Π . By definition, $\Pi \subseteq \Sigma$. That is, the steerable variables are considered to be input variables, but not all input variables must be steerable. Variables internal to the model and output variables may be

specified as steerable, but they are transformed into input variables for the computation steps where steering is applied. This transformation enables search algorithms to directly assign values to these variables. On subsequent, unassisted execution steps, the model will be again transformed such that those variables are again internal to the model.

3.2 Selecting a Steering Action

If the initial comparison of model and SUT states s_m and s_{sut} reveals a difference, we attempt to steer the model. Fundamentally, we treat steering as a search process. We backtrack the model, instrumenting it with the previous recorded state as the new initial state S_0 , and seek a steering action—a set of values for the steerable variables Π that, when combined with the assigned values to the remaining input variables $\Sigma - \Pi$, transitions the model to a new state s_m^{new} . Note that, if the steering process fails to produce a solution, $s_m^{new} = s_m$. The chosen steering action is an assignment to the variables in Π that satisfies the tolerance constraints, minimizes the dissimilarity function, and follows any additional steering policies (for example, it may need to meet some dissimilarity threshold to be chosen).

The set of **tolerance constraints** governs the allowable changes to values of the steerable variables Π . These constraints define bounds on the non-determinism or behavioral deviation that can be accounted for with steering. Constraints can be expressed over any internal, input, or output variables of the model—not just the members of Π . Constraints can even refer to the value of a variable in the SUT. However, implicitly, these constraints limit the values that can be assigned to Π .

Consider the scenario outlined in Figure 2. We could use steering to correct VRP-related mismatches by defining *VRP* as a member of Π and allowing the search algorithm to assign a new value to it. As we want to limit the change that can be imposed on *VRP*, we must set a constraint. One reasonable choice would be to allow the new value of *VRP* to fall anywhere between a minimum of ($VRP^{original} - 8ms$) and a maximum of ($VRP^{original} + 8ms$).

This differs from setting a filter to compare the values of s_m and s_{sut} because, by changing the state of the model, we impact the state of the model in future steps as well. By adjusting the model each time it diverges, we eliminate the need to track the entire execution history.

We could also create a constraint that combines multiple members of Π , that takes into account model variables not in Π , or that depends on the value of a variable in the SUT. For example, we could establish a constraint on *sense* that depends on the output variable *timeOut* as follows: *if* $((timeOut^{sut} \geq timeOut^{original}) \text{ and } (timeOut^{sut} \leq timeOut^{original} + 4ms))$ *then* $((sense^{new} = 0) \text{ or } (sense^{new} = 1))$ *else* $(sense^{new} = sense^{original})$. That is, we can freely change whether an event was sensed, as long as the value of *timeOut* in the SUT is within a certain range of the value of *timeOut* in the model.

After using the tolerance constraints to limit the number of candidate solutions, the search process is guided to a solution

through the use of a **dissimilarity function** $Dis(model\ state, SUT\ state)$, that compares the state of the model to the observable state of the SUT. We seek a minimization of $Dis(s_m^{new}, s_{sut})$. There are many functions that can be used to calculate dissimilarity. Cha provides a good primer on the calculation of dissimilarity [26]. As we primarily used models with numeric variables, dissimilarity functions such as the Euclidean distance—the average difference between two variable vectors—were found to be sufficient to guide this selection. When considering variable comparisons over—for instance—strings, more sophisticated dissimilarity metrics (such as the Levenshtein distance [27]) are appropriate.

We can further constrain the steering process by employing a set of general **policy decisions** on when to steer. For example, one might decide not to steer unless $Dis(s_m^{new}, s_{sut}) = 0$, that is, there must exist a steering action that results in a model state identical to the SUT state.

These policies are intended to capture constraints that can be applied to control steering, regardless of the system under test. While such policies could, in many cases, be expressed as part of the tolerance constraints, they have been separated and expressed as generic options that can be easily enabled or disabled when executing the steering framework. The ability to invoke options like the one above gives a user a set of options to try when they begin to explore steering.

To summarize, the new state of the model-based oracle following the application of a steering action must be a state that is reachable from the current state of the model, must fall within the boundaries set by the tolerance constraints, and must minimize the dissimilarity function. This process is illustrated in Figure 5 for one test step from the test in Figure 2 (when input occurs 380 ms into the test). SimplePacing is instrumented such that the initial state matches the state that the model was in, following the application of input in the immediately-preceding time step. The steerable variable set Π consists of the input *VRP*, with the tolerance constraint that the chosen value of *VRP* must fall between $VRP^{original} - 8ms$ and $VRP^{original} + 8ms$. We examine the candidate states, evaluate their dissimilarity score, then choose the steering action that minimizes this score. We transition the model to this state and continue test execution.

We have implemented the basic search approach outlined in Figure 6. Our search process is based on bounded model checking [28], an automatic approach to property verification for concurrent, reactive systems [29]. The problem of identifying an appropriate steering action can be expressed as a Satisfiability Modulo Theories (SMT) instance, a generalization of a boolean satisfiability instance in which atomic boolean variables are replaced by predicates expressed in classical first-order logic [30]. A SMT problem can be thought of as a form of a constraint satisfaction problem—in our specific case, we seek a set of values for the steerable variables that obeys the set of tolerance constraints and has a lower dissimilarity score than the original. We have made use of the Kind [28] model checker, and the Z3 constraint solver [31].

When we execute tests, each test step is checked explicitly for conformance violations by comparing the state of the oracle and the SUT. If a mismatch is detected, we allow the

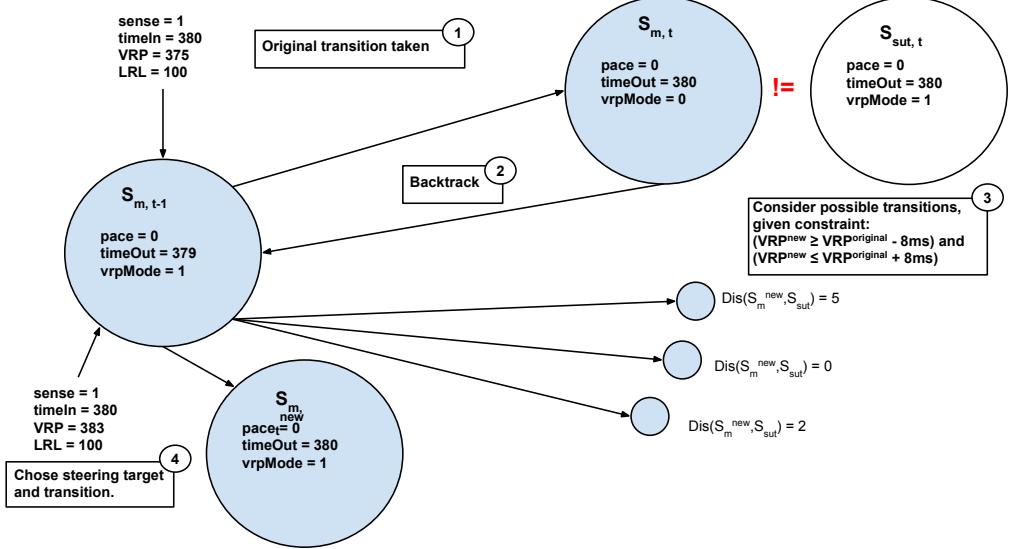


Fig. 5: Illustration of steering process for SimplePacing (Figure 1) for one step of the test depicted in Figure 2.

Input: $Model, S_m, S_{sut}$

- 1 **if** $Dis(S_m, S_{sut}) > 0$ **then**
- 2 $DisConstraint := \lambda threshold \rightarrow \lambda state \rightarrow$
 $Dis(state, S_{sut}) \leq threshold$
- 3 $targetState := searchForNewState$
 $(Model, S_m, S_{sut}, Constraints, DisConstraint(0))$
- 4 **if** $targetState = NULL$ **then**
- 5 $newState := S_m$
- 6 $T := 1$
- 7 **while** $newState \neq NULL$ **do**
- 8 $targetState := newState$
- 9 $newState := searchForNewState$
 $(Model, targetState, S_{sut}, Constraints,$
 $DisConstraint(T \times Dis(targetState, S_{sut}))$
- 10 $T := 0.5 \times T$
- 11 $transitionModel(Model, targetState)$

Fig. 6: Outline of steering process.

steering framework to search for a new solution. In order to achieve this, we explicitly instrument the model such that the current state (before applying the chosen steering action) is the “initial” state. This instrumentation also embeds the calculation of the dissimilarity function and the tolerance constraints directly into the model. Constants are also embedded in the model for each steerable variable and output variable in the model and each output variable in the SUT describing what occurred originally during test execution. That is, they tell us what happens when we do not steer. These values are used both for calculating the value of the dissimilarity score and in the tolerance constraints².

An expression is formulated containing the tolerance constraints and the threshold that we want the new dissimilarity score to beat. This expression is then negated because we want a *counterexample*—we assert that the constraints *can not* be satisfied, and ask the search algorithm to find a set of values

2. An example of this instrumentation and additional technical details can be seen in [32].

for the steerable variables that *can* satisfy those constraints within a single transition. We take the counterexample offered by the search algorithm, extract the values of the steerable variables, and replace the original values of those variables in the trace. We then apply the new set of input (non-steerable inputs that retain their original values and the new values of the steerable variables) to the instrumented model, record new variable values in the trace, and continue test execution.

It should be noted that an SMT solver may not be able to directly minimize $Dis(s_m^{new}, s_{sut})$. Instead, such solvers will offer any solution with a lower dissimilarity score. As outlined in Figure 6, we instead find a minimal solution by first using the constraint $Dis(s_m^{new}, s_{sut}) = 0$ —we ask the solver for a solution where $s_m^{new} = s_{sut}$. If an exact minimization cannot be found, and our policies still allow steering in that situation, we then attempt to narrow the range of possible solutions by setting a threshold value T and using the constraint $Dis(s_m^{new}, s_{sut}) < (T * Dis(s_m, s_{sut}))$. If a solution is possible, we continue to set $T = 0.5 * T$ until a solution is no longer found. Once that constraint can no longer be satisfied, we take the last viable solution and iteratively apply the constraint $Dis(s_m^{new}, s_{sut}) < Dis(s_m^{previous\ new}, s_{sut})$ until we can no longer find a better solution. The best solution found will be selected as the steering action.

It should also be noted that the results and stability of the search process depend on the algorithm employed. The solver used in our implementation, Z3, returns deterministic results. Other algorithms may not return the same steering action each time a test is executed.

3.3 Defining Constraints

The efficacy of the steering process depends on the tolerance constraints and policies employed. If the constraints are too strict, steering will be ineffective—leaving as many “false failure” verdicts as not steering at all. On the other hand, if the constraints are too loose, steering runs the risk of covering up real faults in the system. Therefore, it is important that the constraints to be employed are carefully considered.

Often, constraints can be inferred from the system requirements and specifications. For example, when designing an embedded system, it is common for the requirements documents to specify a desired accuracy range on physical sensors. If the potential exists for a model-system mismatch to occur due to a mistake in reading input from a sensor, than it would make sense to take that range as a constraint on that sensor input and allow the steering algorithm to try values within that range.

We recommend that users err toward strict constraints to avoid masking faults. To give an example, while the inputs of a medical device may include a patient prescription, we would recommend that steering be prohibited from altering the prescription values, as manipulation of those might threaten the life of a patient. Instead, the focus of constraints should be on areas where adding flexibility to the oracle cannot harm a patient, such as minor timer or sensor adjustments.

Constraints may be defined over any of the variables of the system (input, output, or internal to a function). When no constraints are specified for a variable, no steering actions may be taken. We recommend that testers start with a minimal set of strict constraints, then loosen them until the number of false-failure verdicts is sufficiently reduced. While it is possible to steer the value of output variables directly, we do not generally recommend doing so. It is safer to steer the factors leading to unexpected output than to directly overwrite that output. As steering decouples the specification of allowable non-determinism from the model, testers can experiment with different sets of constraints and policies. Alternative options can easily be explored by swapping in a new constraint file and executing the test suite again until they are confident in their selections.

As the software changes during development, tests must often be altered to account for changes in SUT interfaces or other internal behavioral changes. As the constraints employed by steering are independent of the test cases themselves, the use of steering does not increase the required *test* maintenance effort. However, if changes to the SUT make certain steering actions illegal, alter the behavior of system variables, or remove variables used in constraints, then some *constraint* maintenance must take place. As we recommend minimal sets of constraints, the required level of constraint maintenance should be low.

3.4 Handling Non-Determinism Through Steering or the Underlying Model

By incorporating a sophisticated model of time or other forms of non-determinism, some of these modeling approaches discussed in Section 8 could account for a subset of the non-deterministic behaviors induced during execution of the SUT—particularly variance related to timing issues. A natural question, then, is whether to use steering or to simply incorporate this non-determinism into the underlying model.

In such cases, the model *must* be built with those execution behaviors in mind. These behaviors depend specifically on the particular details of the real-world execution—such as the underlying hardware platform. Such details are difficult to

anticipate, and assumptions would need to be made at the time that the model is constructed. If the model is built—as many are—during requirements engineering, then it is difficult, if not impossible, to make correct assumptions. Fixing incorrect assumptions may require extensive overhaul of the model’s structure. This task may need to be performed on a regular basis as the hardware or software evolves. The alternative is to wait until the implementation is ready to be tested to build the model-based oracle. This is also an inefficient outcome if the model would be useful for early-lifecycle tasks such as requirements analysis.

There is an argument to be made for not building these details directly into the model in the first place. Effective requirements analysis requires *clarity*. Being forced to incorporate the full scope of the non-deterministic behavior of the SUT into the model could muddle its clarity, making the analysis of the requirements more difficult and potentially leading to an incorrect implementation.

Through the use of the tolerance constraints, we effectively decouple the model from the rules governing conformance. This decoupling makes non-determinism implicit and the approach more generally applicable. Explicitly specified non-deterministic behavior—as required by the model-based approaches described above—would limit the scope of non-determinism handled by the oracle to what has been planned for by the developer and subsequently modeled. It is difficult to anticipate the non-determinism resulting from deploying software on a hardware platform, and, thus, such models will likely undergo several revisions during development. Steering instead relies on a set of rule-based constraints that may be easier to revise over time. Additionally, by not relying on a specific model format, steering can be made to work with models created for a variety of purposes. By not being tied to a specific test generation framework, we can make use of tests from a variety of tools, or more easily build steering into a number of existing frameworks.

Once allowable behavioral deviations have been observed, if a non-deterministic model is employed, one could choose to either steer to temporarily account for such situations or update the model to permanently account for such situations. These options are not mutually exclusive. Steering could be used to refine the tester’s understanding of the behavioral divergences they see in practice. After using steering, they could update the model. In that case, steering is useful as a discovery tool. If, for example, different hardware platforms are being considered, it may be wise to use steering until stable decisions have been reached. This prevents the need to make frequent—potentially difficult—model revisions.

4 LEARNING CONSTRAINTS

Choosing the correct constraints is essential to accurate steering, but it is not always clear what those constraints should be, or even what variables should be manipulated in the first place. However, even in these situations, the developers of the system under test *should* at least have an idea of whether a test should pass or fail. A human oracle is often the ultimate judge on the correctness, as the developers will have a more

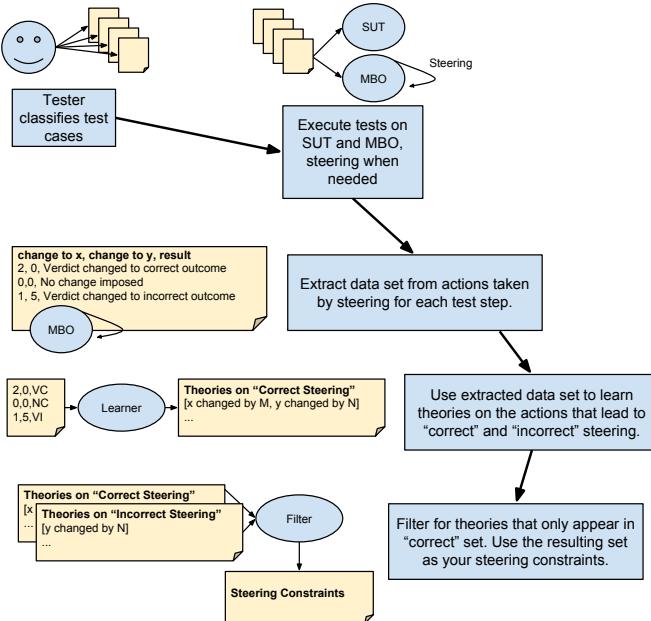


Fig. 7: Outline of learning process.

comprehensive idea of what constitutes correctness than any constructed artifact, even if they are not completely sure of the specific factors that should lead to that verdict.

It is possible to use human-classified test verdicts to *learn* an initial set of constraints. We can treat constraint elicitation as a machine learning problem where we execute a series of tests against the SUT, steer with no value constraints at all—the only limitation being what states are reachable within one transition through changes to the steerable variables—and record information on what changes were imposed by the steering algorithm. If a human serves as an oracle on those tests, we can then evaluate the “correctness” of steering.

For purposes of constraint elicitation, we care about the effects of steering in two situations: successfully steering when we are supposed to steer and successfully steering when we *are not* supposed to steer. By observing the framework-calculated oracle verdict before and after steering and comparing it to the human-classified oracle verdict, we can determine what test steps correspond to those two situations. Using that correctness classification and a set of data extracted from each test step, we can form a data set that can be explored by a variety of learning algorithms. This process is illustrated in Figure 7. The data we extract at each test step is detailed in Table 1. For each steerable variable, we record how much it was altered by steering. For each oracle-checked variable, we record both how much it differed between model and system before steering and how much it was changed after steering. Then we record a classification—did steering perform the correct action? The classification *ChangedCorrect* is assigned when steering acted and arrived at the same verdict as the human. The classification *ChangedIncorrect* is assigned when the post-steering verdict does not match the human-assigned verdict. We also record data when steering does not act at all or does not change the pre-steering verdict. This is assigned the classification *NotChanged*.

We can use this extracted set of data to elicit a set of tolerance constraints. A standard practice in the machine learning field is to *classify data*—to use previous experience to categorize new observations [33]. As new evidence is examined, the accuracy of these categorizations is then refined.

We are instead interested in the reverse scenario. Rather than attempting to categorize new data, we want to work backwards from the classifications to discover *why steering acted correctly or incorrectly*—a process known as treatment learning [34]. Treatment learning approaches take the classification of an observation and try to reverse engineer the evidence that led to that categorization. Such learners produce a *treatment*—a small set of data value boundaries that, if imposed, will produce a subset of the original data matching the desired classification.

Ultimately, classifiers strive to increase accuracy by growing a collection of statistical rules. As a result, if the data is complex, the model employed by the classifier will also be complex. Instead, treatment learning focuses on minimality, delivering the smallest rule that, when imposed, causes the largest impact. This focus is exactly what makes treatment learning interesting as a method of producing constraints. We wish to constrain steering to a small set of steerable variables, with strict limitations on allowed value changes. Treatment learning can be used to generate a minimal initial set of constraints or to tune an existing set of constraints.

To give an example, consider the base class distribution after steering a model-based oracle for a set of classified tests and extracting the data detailed above, as shown on the left in Table 2. This sort of base class distribution makes conceptual sense—on many test steps, steering does nothing. It only kicks in when the oracle and model differ, makes a change, and likely reduces the number of future steps in the same test case where differences occur. By targeting the class *ChangedCorrect*, we can attempt to elicit a treatment that details what happens when steering acts correctly—when it changes the oracle verdict in a way that matches the human-assigned verdict. We can extract treatments, ranked by their score assigned by the treatment learner’s objective function.

Example treatments are shown in Table 3. Each treatment includes variables and their value ranges that are correlated to the targeted classification. The first treatment in Table 3 states that the major indicators of a correct steering action are when the value of Variable1 is reduced between 0-4 and the value of Variable2 is increased by 1-2 by steering. By imposing that treatment, we end up with the class distribution shown on the right in Table 2. This class distribution shows strong support for the produced treatment. By allowing steering to change Variable1 and Variable2 in the prescribed manner, steering tends to match the human-assigned verdict.

In order to create a set of tolerance constraints, we first create 10 treatments like those seen in Table 3 using “*ChangedCorrect*” as our target class (we wish to know what actions steering takes when it works correctly) and extract all of the individual variable and range pairings. Treatment learning algorithms operate scholastically, so 10 treatments are produced to account for randomness. Some of these items may not actually be indicative of successful steering—they

Variables Involved	Attribute	Values
For each steerable variable	How much was it changed?	Continuous
For each oracle-checked variable	How much did it differ before steering?	Continuous
For each oracle-checked variable	How much was it changed?	Continuous
(class variable)	Did steering change the verdict correctly?	NotChanged, ChangedIncorrect, ChangedCorrect

TABLE 1: Data extracted for tolerance elicitation. The classification *ChangedCorrect* means that the post-steering verdict matched the human-assigned verdict, while *ChangedIncorrect* implies that the post-steering verdict does not match. *NotChanged* means that steering does not act at all or does not change the pre-steering verdict.

Class	Percentage	Class	Percentage
NotChanged	96%	NotChanged	0%
ChangedIncorrect	1%	ChangedIncorrect	14%
ChangedCorrect	2%	ChangedCorrect	86%

TABLE 2: Base class distribution and class distribution of the remaining subset of the data after imposing a treatment.

Rank	Treatment
1	[changeToVariable1=[-4.000000..0.000000] [changeToVariable2=[1.000000..2.000000]
2	[changeToVariable2=[1.000000..2.000000] [changeToVariable3=[-10.000000..0.000000]
3	[changeToVariable2=[1.000000..2.000000]
4	[changeToVariable4=[-15.000000..0.000000] [changeToVariable2=[1.000000..2.000000]
5	[changeToVariable2=[1.000000..2.000000] [changeToVariable5=[-176.000000..0.000000]

TABLE 3: Examples of learned treatments. Treatments are assigned a “goodness” score and sorted by that score. Each treatment includes variables and their value range that is correlated to the targeted classification.

```
((Variable3 >= concrete_oracle_Variable3 - 10.0) and
 (Variable3 <= concrete_oracle_Variable3))
((Variable2 = concrete_oracle_Variable2) or
 ((Variable2 >= concrete_oracle_Variable2 + 1.0) and
 (Variable2 <= concrete_oracle_Variable2 + 2.0)))
((Variable1 >= concrete_oracle_Variable1 - 4.0) and
 (Variable1 <= concrete_oracle_Variable1))
((Variable6 >= concrete_oracle_Variable6 - 13.0) and
 (Variable6 <= concrete_oracle_Variable6))
(Variable7 = concrete_oracle_Variable7)
(Variable8 = concrete_oracle_Variable8)
...
(Variable20 = concrete_oracle_Variable20)
```

Fig. 8: Examples of produced tolerances. *concrete_oracle_X* is a constant reflecting the value of that variable when steering is not employed.

may be variable values selected by biases in the algorithm that selects the steering actions that appear in both *correct* and *incorrect* steering. Thus, we also produce 10 treatments using “*ChangedIncorrect*” as the target class. This produces a set of treatments indicating what happens when steering incorrectly changes an oracle verdict. We remove any variable and value range pairings that appear in both the “good” and “bad” sets, leaving only those that appear in the good set. We then form our set of elicited tolerance constraints by locking down any variables that constraints were not suggested for. This results in a set of tolerances similar to that shown in Figure 8.

Notable treatment learning algorithms include the TAR family (TAR2 [34], TAR3 [35], [36], [37], and TAR4.1 [38])—a series of algorithms utilizing a common core structure, but employing different objective functions and search heuristics—and the STUCCO contrast-set learner [39]. Other optimization algorithms have also been applied to treatment learning,

including simulated annealing and gradient descent [38]. In this work, we employed the TAR3 algorithm.

5 CASE STUDY

We aim to assess the capabilities of oracle steering and the impact it has on the testing process. Thus, we pose the following research questions:

- 1) To what degree does steering lessen behavioral differences that are legal under the system requirements?
- 2) To what degree does steering mask behavioral differences that fail to conform to the requirements?
- 3) Are there situations where a filtering mechanism is more appropriate than actively steering the oracle?
- 4) To what degree does the strictness of the employed constraints impact the correctness of steered oracle verdicts?
- 5) How accurate are tolerance constraints learned automatically from previously steered test execution traces?

5.1 Experimental Setup Overview

Our case study centers around models of two industrial-scale medical device systems. The first is the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [40]. This subsystem takes in a prescription for a drug—as well as several sensor values—and determines the appropriate dosage of the drug to be administered to a patient over a given period of time. The second system is based on the pacing subsystem of an implanted pacemaker, built from the requirements document provided to the Pacemaker Challenge [17]. This subsystem monitors cardiac activity and, at appropriate times, commands the pacemaker to provide electrical impulses to the appropriate chamber of the heart.

These models are developed in the Stateflow notations and translated into the Lustre synchronous programming language to more easily perform automated transformations [41]. The simplicity and declarative nature of Lustre make it well-suited to model checking and verification [28]. This also makes it an ideal language to use as a target for steering because the steering constraints and dissimilarity function can be encoded directly into the model, and a steering action can be selected using the same algorithms that are regularly used to perform verification. Because typical discrete state-transition systems are semantically similar to Lustre, it is easy to translate from other modeling paradigms to Lustre while preserving the semantic structure of those models.

Both case examples are complex real-time systems of the type common in the medical device domain. Details on the Stateflow and translated Lustre models are provided in Table 4.

To evaluate the performance of oracle steering, we performed the following for each system:

	# States	# Transitions	Lustre LOC	Input Variables	Internal Variables	Output Variables
Infusion_Mgr	23	50	6,299	19	825	5
Pacing	48	120	24,017	18	545	5

TABLE 4: Case example information—number of states, number of transitions, lines of code when translated to Lustre, number of input variables, number of internal variables, and number of output variables.

- 1) **Generated system implementations:** We approximated systems running on embedded hardware by creating versions of each model with non-deterministic timing elements. We also generated 50 mutated versions of each version of each system with seeded faults (Section 5.2).
- 2) **Generated tests:** We randomly generated 100 tests for each case example, each varying from 30-100 test steps (input to output sequences) in length (Section 5.2).
- 3) **Set steering constraints:** We constrained the variables that could be adjusted through steering and the values that those variables could take on, and established dissimilarity metrics (Sections 5.4 and 5.3).
- 4) **Assessed impact of steering:** For each combination of SUT, test, and dissimilarity metric, we attempted to steer the oracle to match the behavior of the SUT. We compare the test results before and after steering and evaluate the precision and recall of our steering framework, contrasted against the general practice of not steering and a step-by-step filtering mechanism (Section 5.6).
- 5) **Assessed impact of tolerance constraints:** We repeated steps 3-4 for each SUT and five mutants using four different sets of tolerance constraints, varying in strictness, in order to assess the impact of the choice of constraints (Sections 5.4 and 5.6).
- 6) **Learned new tolerance constraints:** Using the original and steered traces for each model and the trace for each SUT, for each of the four constraint levels and both dissimilarity metrics, we extracted 10 sets of tolerance constraints (= 80 per SUT) using the TAR3 treatment learning algorithm (Section 5.5).
- 7) **Assessed performance of learned tolerance constraints:** We repeated steps 3-4 for each SUT using the extracted tolerance constraints in order to assess the quality of those constraints.

5.2 System and Test Generation

To produce implementations of the example systems, we created alternative versions of each model, introducing realistic non-deterministic timing changes to the systems. Non-determinism was simulated in this study in order to more clearly examine the effectiveness of steering under controlled experimental conditions. For the Infusion system, we built (1) a version of the system where the exit of the patient-requested dosage period may be delayed by a short period of time, and (2) a version of the system where the exit of an intermittent increased dosage period (known as a square bolus dose) may be delayed. These changes are intended to mimic situations where, due to hardware-introduced computation delays, the system remains in a particular dosage mode for longer than expected.

For the Pacing system, we introduced a non-deterministic delay on the arrival of sensed cardiac activity. As a pacemaker

is a complex, networked series of subsystems that depend on strict timing conditions, a common source of mismatch between model and system occurs when sensed activity arrives at a particular subsystem later than expected. Depending on the extent of the delay, unnecessary electrical impulses may be delivered to the patient or the pacemaker may enter different operational modes than the model.

For each of the original models and SUT variants, we have also generated 50 *mutants* (faulty implementations) by introducing a single fault into each model. This ultimately results in a total of 152 SUT versions of the Infusion system—two versions with non-deterministic timing behavior, fifty versions with faults, and one hundred versions with both non-deterministic timing and seeded faults (fifty per timing variation)—and 101 SUT variants of the Pacing system—one SUT with non-deterministic timing, fifty with faults, and fifty with both non-deterministic timing and seeded faults.

The mutation testing operators used in this experiment include changing an arithmetic operator, changing a relational operator, changing a boolean operator, introducing the boolean \neg operator, using the stored value of a variable from the previous computational cycle, changing a constant expression by adding or subtracting from integer and real constants (or by negating boolean constants), and substituting a variable occurring in an equation with another variable of the same type. The mutation operators used are discussed at length in an earlier report [42], and are similar to those used by Andrews et al, where the authors found that generated mutants are a reasonable substitute for real faults in testing experiments [43].

Using a probabilistic random testing algorithm, we generated 100 tests for each system. Each test captures a period of time long enough to observe complex time-sensitive behaviors, but still short enough to yield a reasonable experiment cost. For the Infusion system, each test is thirty steps in length, representing thirty seconds of system activity. For the pacing system, the tests range 30-100 steps in length, representing input and output events occurring over 3000 ms of activity. The generation algorithm allows testers to assign probabilities to particular input values or ranges. In this case, these probabilities were chosen to reflect “normal” execution of the system, rather than extreme events—for example, Infusion_Mgr’s power is very unlikely to be turned off during a test case. For this study, normal execution was favored to clearly examine the influence of steering.

These tests were then executed against each model and SUT variant in order to collect traces. In the SUT variants with timing fluctuations, we controlled those fluctuations through the use of an additional input variable. The value for that variable was generated non-deterministically, but we used the same value across all systems with the same timing fluctuation. As a result, we know whether a behavioral mismatch is due to a timing fluctuation or a seeded fault in the system. Using this knowledge, we manually classified each test as an “expected

pass” or as failing due to an “acceptable timing deviation”, an “unacceptable timing deviation”, or a “seeded fault.”

5.3 Dissimilarity Metrics

We have made use of two dissimilarity metrics when comparing states. The first is the Manhattan (or City Block) distance. Given vectors representing the state of the SUT and the model-based oracle—where each member of the vector represents the value of a variable—the dissimilarity between the two vectors can be measured as the sum of the absolute numerical distance between the state of the SUT and the model-based oracle:

$$Dis(s_m, s_{sut}) = \sum_{i=1}^n |s_{m,i} - s_{sut,i}| \quad (1)$$

The second is the Squared Euclidean distance. Given vectors representing the state, the dissimilarity between the vectors can be measured as the “straight-line” numerical distance between the two vectors. The squared variant was chosen because it places greater weight on states that are further apart in terms of variable values.

$$Dis(s_m, s_{sut}) = \sum_{i=1}^n (s_{m,i} - s_{sut,i})^2 \quad (2)$$

A constant difference of 1 is used for differences between boolean variables or values of an enumerated variable. All numerical values are normalized to a 0-1 scale using predetermined minimum and maximum values for each variable.

When computing a dissimilarity metric—or, for the matter, an oracle verdict—we must choose a set of variables to compare. As we cannot assume a common internal structure between the SUT and the model, we calculate similarity using the output variables of both. For the infusion pump, this includes the commanded flow rate, the current system mode, the duration of active infusion, a log message indicator, and a flag indicating that a new infusion has been requested. For the pacemaker, this set of variables consists of an atrial event classification, a ventricular event classification, the time of the event, the time of the next scheduled atrial pace attempt, and the time of the next scheduled ventricular pace attempt.

5.4 Manually-Set Tolerance Constraints

In order to assess the performance and capabilities of steering, we have specified a realistic set of tolerance constraints for both systems. These constraints were developed using the actual software and hardware specifications for each system and by consulting with domain experts. The chosen tolerance constraints for the Infusion system include:

- There are five timers within the system—the duration of the patient-requested bolus dose period, the duration of the intermittent square bolus dosage period, the lockout period between patient-requested bolus dosages, the interval between intermittent square bolus dosages, and the total duration of the infusion period. For each of those, we placed an allowance of $(CurrVal - 1) \leq NewVal \leq (CurrVal + 2)$. E.g., following steering, a dosage duration is allowed to fall between one second shorter and two seconds longer than the original prescribed duration.

and, for the Pacing system:

- The input for a sensed cardiac event includes a timestamp indicating when the system will process the event. For this event, we placed an allowance of $Current\ Value \leq New\ Value \leq (Current\ Value + 4)$. Following steering, the sensed event can take place up to four milliseconds after the original timestamp.
- Boolean input variables indicate whether an event was sensed in the atrial or ventricular chambers of the heart. These can be toggled on or off, to better match the noise filtering conducted by the SUT.

These constraints reflect what we consider a realistic application of steering—we expect issues related to timing, and, thus, allow a small acceptable window around the behaviors that are related to timing. For the Infusion system, we do not expect any sensor inaccuracy, so we do not allow freedom in adjusting sensor-based input. For Pacing, we expect a small amount of event reordering and noise sensitivity, so we allow a small amount of freedom in changing sensor-based values. As these are restrictive constraints, we deem these the **Strict** tolerance constraints.

In order to assess the impact of different sets of constraints, we took each SUT variant, five randomly-selected mutants of the original system, and five randomly-selected mutants for each SUT and attempted to steer them using the Strict tolerances and three additional sets of tolerances: **Medium**, **Minimal**, and **No Input Constraints**.

For the Infusion system, these are as follows:

- **Medium:** All time-based inputs, $(CurrVal - 2) \leq NewVal \leq (CurrVal + 5)$. All other variables are not allowed to be steered.
- **Minimal:** All time-based inputs completely unconstrained. All other variables are not allowed to be steered.
- **No Input Constraints:** All input variables unconstrained.

and, for the Pacing system:

- **Medium:** Ventricle and atrial sensed events unconstrained. Event time, $Current\ Value \leq New\ Value \leq (Current\ Value + 25)$. Refractory periods, $Current\ Value \leq New\ Value \leq (Current\ Value + 10)$. All other variables are not allowed to be steered.
- **Minimal:** Ventricle and atrial sensed events and event time unconstrained. Refractory periods, $Current\ Value \leq New\ Value \leq (Current\ Value + 25)$. Lower and upper rate limit, $Current\ Value \leq New\ Value \leq (Current\ Value + 10)$. All other variables are not allowed to be steered.
- **No Input Constraints:** All input variables unconstrained.

These additional constraint sets reflect a gradual relaxation of the limits on steering, and can demonstrate the impact that the choice of constraints has on the effectiveness of steering.

5.5 Learning Tolerance Constraints

In Section 4, we discussed the process of using treatment learning to extract a set of tolerance constraints. We wish to

know whether we can learn tolerances for our case studies, and whether these tolerances are effective at guiding the steering process. Using the process outlined previously, we generated tolerance constraints for the Infusion variant where the patient bolus period can be extended non-deterministically and for the Pacing system. Starting from the strict, medium, and minimal tolerance constraints and using no input constraints, we executed tests, steered the models, and extracted data from those executions and classifications on what the correct verdict should be post-steering. Because the treatment learners use a stochastic search process, we generate ten sets of constraints per preexisting constraint set (i.e., 10 sets generated after extracting data from steering with strict tolerance constraints, 10 sets generated after extracting data from steering with no input constraints, etc). We repeat this for each dissimilarity metric. This results in eighty sets of tolerance constraints for each system (4 constraint levels x 10 repeats x 2 metrics).

We generate tolerances using the TAR3 treatment learning algorithm [38], [35], [36], [37]. It produces treatments by first being fed a set of training examples. Each example consists of values, discretized into a series of ranges, for a given set of attributes. This set of value ranges is directly mapped to a specific classification. As in Section 4, each example corresponds to a test step, where the attributes of the data set represent the changes made to variable values by steering and the correctness of the changes.

In order to create a set of tolerance constraints, we first generate 10 treatments—like those seen in Table 3—that indicate the correct use of steering to adjust the state of the model. Some of these items may not actually indicate successful steering—they may be steering actions that occur at the same time as actions that are actually good. Thus, we also produce 10 treatments that correspond to incorrect steering actions, and remove any treatments that appear in both the “good” and “bad” sets. We then form our set of elicited tolerance constraints by locking down any variables that constraints were not suggested for. This results in a set of tolerances similar to that shown in Figure 8. We repeat this process ten times for each constraint level and dissimilarity metric in order to control for the effects of variance.

5.6 Evaluation

Using the generated artifacts—without steering—we monitored the output during each test, compared the results to the values of the same variables in the model-based oracle at each time step to calculate the dissimilarity score, and issued an initial verdict³. Then, if the verdict was a failure ($Dis(s_m.s_{sut}) > 0$), we steered the model-based oracle, and recorded a new verdict post-steering. The variables used in establishing a verdict are the five output variables of the system.

In Section 3, we stated that an alternative approach to steering would be to apply a filter on a step-by-step basis. We have implemented such a filter for the purposes of establishing a baseline to which we can compare the performance of

3. This corresponds to “oracle strategy 2” in Li and Offutt’s hierarchy—checking the return values of methods invoked during each transition [44]

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	TN	FP
Fail (Due to Timing, Within Tolerance)	TN	FP
Fail (Due to Timing, Not in Tolerance)	FN	TP
Fail (Due to Fault)	FN	TP

TABLE 5: Verdicts: T(true)/F(false), P(positive)/N(negative).

steering. This filter compares the values of the output variables of the SUT to the values of those variables in the model-based oracle and, if they do not match, checks those values against a set of constraints. If the output—despite non-conformance to the model—meets these constraints, the filter will still issue a “pass” verdict for the test.

For the Infusion_Mgr system, the filter will allow a test to pass if (despite non-conformance) values of the output variables in the SUT satisfy the following constraints:

- The current mode of the SUT is either “patient dosage” mode or “intermittent dosage” mode, and has not remained in that mode for longer than *prescribed duration* + 2 seconds.
- If the above is true, the commanded flow rate should match the prescribed value for the appropriate mode.
- All other output variables should match their corresponding variables in the oracle.

As we expect a non-deterministic duration for the patient dosage and intermittent dosage modes (corresponding to the seeded issues in the SUT variants), this filter should be able to correctly classify many of the same tests that we expect steering to handle.

For the Pacing system, the filter will allow a test to pass if the values of the output variables satisfy:

- The event timestamp on the output and the scheduled time of the next atrial and ventricular events fall within four milliseconds of the time originally predicted by the model.
- All other output variables should match their corresponding variables in the oracle.

Similar to the Infusion_Mgr system, we expect short non-deterministic delays in when the Pacing system issues an output event.

We compare the performance of the steering approach to both the filter and the default practice of accepting the initial test verdict. We can assess the impact of steering or filtering using the verdicts made before and after steering by calculating:

- The number of *true positives*—steps where an approach does not mask incorrect behavior;
- The number of *false positives*—steps where an approach fails to account for an acceptable behavioral difference;
- And the number of *false negatives*—steps where an approach does mask an incorrect behavior.

The testing outcomes in terms of true/false positives/negatives are listed in Table 5. Using these measures, we calculate the *precision*—the ratio of true positives to all positive verdicts—and *recall*—the ratio of true positives to true positives and false negatives:

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (4)$$

We also calculate the *F-measure*—the harmonic mean of precision and recall—in order to judge the accuracy of oracle verdicts:

$$\text{Accuracy (F-measure)} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \quad (5)$$

6 RESULTS AND DISCUSSION

As previously presented in Table 5, testing outcomes can be categorized according to the initial verdict as determined by the model-based oracle before steering; a “fail” verdict is further delineated according to its reason—a mismatch that is attributable to either an allowable timing fluctuation, an unacceptable timing fluctuation or a fault.

For the Infusion_Mgr system—when running all tests over the various implementations (containing either timing deviations or seeded faults as discussed in Section 5.2) using a standard test oracle comparing the outputs from the SUT with the outputs predicted by the model-based oracle (15,200 test runs)—11,364 runs indicated that the system under test passed the test (the SUT and model-based oracle agreed on the outputs) and 3,936 runs indicated that the test failed (the SUT and model-based oracle had mismatched outputs). In an industry application of a model-based oracle, the 3,936 failed tests would have to be examined to determine if the failure was due to an actual fault in the implementation, an unacceptable timing deviation from the expected timing behavior, or an acceptable timing deviation that, although it did not match the behavior predicted by the model-based oracle, was within acceptable tolerances—a costly process. Given our experimental setup, however, we can classify the failed tests as to the cause of the failure: failure due to timing within tolerances, failure due to timing not in tolerance, and failure due to a fault in the SUT. This breakdown is provided in the “No Adjustment” column of Table 6. As can be seen, 1,406 tests failed even though the timing deviation was within what would be acceptable—these can be viewed as false positives and a filtering or steering approach that would have passed these test runs would provide cost savings. On the other hand, the steering or filtering should not pass any of the 268 tests where timing behavior falls outside of tolerance or the 2,229 tests that indicated real faults.

A similar breakdown can be found for the Pacing system in the “No Adjustment” column of Table 7. About a third of the test executions—2,992 in total—pass initially. A further 21%, or 2,208 tests, fail due to acceptable timing deviations. These should, ideally, pass following the application of steering or filtering. A further 571 test executions fail due to unacceptable timing differences, and 4,329 fail due to seeded faults. Steering and filtering should, ideally, not correct these tests.

Results obtained from the case study showing the effect of steering or filtering on oracle verdicts are summarized in Table 6 and 7 respectively, for the Infusion_Mgr and Pacing systems. For both systems, the two dissimilarity metrics performed identically when steering. The raw results are presented as laid out in Table 5. For each category, the post-steering verdict is presented as both a raw number of test

outcomes and as a percentage of total test outcomes. Data from these tables lead to the precision, recall, and accuracy values are shown in Table 8 for the default testing scenario (accepting the initial oracle verdict), steering, and filtering. In the following sections, we will discuss the results presented in these tables with regard to our central research questions.

6.1 Performance

By necessity, steering adds overhead to the test execution process. While steering is performed largely offline—using traces gathered from the SUT—the execution time required to perform steering should not be onerous. For tests where steering is not performed, the framework takes an average of 0.008 seconds of processing time. For tests where steering is employed, execution takes an average of 13.54 seconds for Infusion_Mgr and 162.91 seconds for Pacing on a workstation with an Intel Core i7-4790 four core CPU clocked at 3.60GHz and 16 GB of RAM. For simpler models, such as Infusion_Mgr, this is not a substantial increase in execution time. However, we would like to improve performance on larger models. This will be a focus of future work.

6.2 Allowing Tolerable Non-Conformance

For the Infusion_Mgr system—according to the “No Adjustment” category of Table 6—11% of the tests (1,674 tests) initially fail due to timing-related non-conformance. Of those, 1406 tests (9.2% of the total) fall within the tolerances set in the requirements. Steering should result in a pass verdict for all of those tests. Similarly, of the 2,779 tests (26.9%) that fail due to timing reasons in the Pacing system, 2,208 (21.2%) fail due to differences that are acceptable, and steering should account for these execution divergences (see the “No Adjustment” column of Table 7).

As Table 6 shows, for both dissimilarity metrics, steering is able to account for almost all of the situations where it should be able to correct the model. We see that steering using either distance metric correctly passes 1,245 tests where the timing deviation was acceptable—tests that without steering failed. Therefore, we see a sharp increase in precision over the default situation where no steering is employed (from 0.64 when not steering, to 0.94 when steering, according to Table 8).

Similar results for steering on the Pacing system can be seen in Table 7 . Steering correctly changes the verdicts of 2,065 of the tests that initially failed. As shown in Table 8, this results in a large increase in precision—from 0.69 when not steering to 0.97.

Where previously developers would have had to manually inspect the more than 25% of all test execution traces for the Infusion_Mgr system (the sum of all “fail” verdicts in Table 6) to determine the causes for their failures (system faults or otherwise), they could now narrow their focus to the roughly 17% of test executions that still result in failure verdicts post-steering. For the Pacing system, steering drops this total from 70% inspection rate to a somewhat more manageable 47% (the sum of all remaining post-steering “fail” verdicts in Table 7).

Given the large number of tests in this study, this reduction represents a significant savings in time and effort, removing

Initial Verdict	No Adjustment	Pass (Post-Adjustment)		Fail (Post-Adjustment)	
		Steering	Filtering	Steering	Filtering
Pass		11,364 (74.8%)		0 (0.0%)	
Fail (Due to Timing, Within Tolerance)	1,406 (9.2%)		1,245 (8.2%)		152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	268 (1.8%)		0 (0.0%)		268 (1.8%)
Fail (Due to Fault)	2,229 (14.6%)	43 (0.3%)	1,252 (8.2%)	2186 (14.3%)	977 (6.4%)

TABLE 6: Results when accepting the default verdict, steering, and filtering for Infusion_Mgr. Steering results identical for both dissimilarity metrics.

Initial Verdict	No Adjustment	Pass (Post-Adjustment)		Fail (Post-Adjustment)	
		Steering	Filtering	Steering	Filtering
Pass		2,992 (29.6%)		0 (0.0%)	
Fail (Due to Timing, Within Tolerance)	2,208 (21.2%)	2,065 (20.4%)	1,010 (10.0%)	143 (1.4%)	1,198 (11.9%)
Fail (Due to Timing, Not in Tolerance)	571 (5.7%)		0 (0.0%)		571 (5.7%)
Fail (Due to Fault)	4,329 (42.9%)	297 (2.9%)	258 (2.6%)	4,032 (39.9%)	4,071 (40.3%)

TABLE 7: Results when accepting the default verdict, steering, and filtering for Pacing.

Technique	Infusion_Mgr			Pacing		
	Precision	Recall	Accuracy	Precision	Recall	Accuracy
No Adjustment	0.64	1.00	0.78	0.69	1.00	0.82
Filtering	0.89	0.50	0.64	0.79	0.95	0.86
Steering (Both Metrics)	0.94	0.98	0.96	0.97	0.94	0.95

TABLE 8: Precision, recall, and accuracy results when accepting the initial oracle verdict, steering, and filtering. Best values are marked in bold for each system.

between potentially thousands of execution traces that the developer would have needed to inspect manually. Still, for both systems, there were a small number of tests that steering should have been able to account for (152, or 1% of the test executions, for Infusion_Mgr and 143, or 1.4%, for Pacing). The reason for the failure of steering to account for allowable differences can be attributed to a combination of three factors: the tolerance constraints employed, the dissimilarity metric employed, and internal design differences between the SUT and the model-based oracle.

First, it may be that the tolerance constraints were too strict to allow for situations that should have been considered legal. As discussed in Section 3.3, the employed tolerance constraints play a major role in determining the set of candidate steering actions. By design, constraints should be relatively strict—after all, we are overriding the nominal behavior of the oracle while simultaneously wishing to retain the oracle’s power to identify faults. Yet, the constraints we apply should be carefully designed to allow steering to handle these allowed non-conformance events. In this case, the chosen constraints may have prevented steering from acting in a relatively small number of situations in which it should have been able to account for a behavior difference. This is to be expected, and one of the strengths of steering is that it is relatively easy to tune the constraints and execute tests again until the right balance is struck. Fortunately, for both systems, the chosen constraints were able to account for the vast majority of situations that should have been corrected.

Second, the dissimilarity metric plays a role in guiding the selection of steering action. In our experiments, we noted no differences between the Manhattan and Squared Euclidean metrics in the solutions chosen—both took the same steering actions. By design, the metrics compare the output variables of the model and SUT (i.e., the set of variables that we use to determine a test verdict) and compute a numeric score. For the systems examined, the output variables were relatively simple numeric or boolean values, and we did not witness any situations where the metric could be “tricked” into favoring

changes to one particular variable or another. In other types of systems, the choice of metric may play a more important role—particularly if, say, string comparisons are needed.

However, although the two metrics performed identically well, they may also both share the same blind spot. The metrics compare state in *this* round of execution, and do not consider the implications of a steering action in future test steps. It is possible that multiple candidate steering actions will result in the same score, but that certain choices will cause *eventual* divergences between the model and SUT that cannot be reconciled at that time. Such a possibility is limited in this particular experiment due to the strictness of the tolerance constraints employed, but will be discussed in more detail with regard to the tolerance experiment examined in Section 6.5. It is possible that the “wrong” steering actions were chosen in those cases where steering failed to correct the verdict—initially closing the execution gap, but causing further eventual divergence. This indicates a need for further research work on limiting future side effects when choosing a steering action, and may necessitate further development of dissimilarity metrics.

Third, as previously discussed, the tolerance constraints reduce the space of candidate targets to which the oracle may be steered. We then use the dissimilarity metric to choose a “nearest” target from that set of candidates. Thus, the relationship between the constraints and the metric ultimately determines the power of the steering process. However, no matter how capable steering is, there may be situations where differences in the internal design of the system and model render steering either ineffective or incorrect. We base steering decisions on state-based comparisons, but those comparisons can only be made on the portion of the state variables common between the SUT and oracle model (and, in particular, we limit this knowledge to the variables used for the oracle’s verdict comparison, as these are the only variables we can assume the common existence of). As a result, there may be situations where we should have steered, but could not, as the state of the SUT depended on internal factors not in common with the

oracle. In general, as the oracle and SUT are both ultimately based on the same set of requirements, we believe that some kind of relationship can be established between the internal variables of both realizations. However, in some cases, the model and SUT may be too different to allow for steering in all allowable situations. The inability of steering to account for tolerable differences for at least some tests in this case study can likely be attributed to the changes made to the SUT versions of the models.

In practice, when tuning the precision of steering, the choice of steering constraints seems to have the largest impact on the resulting accuracy of the steering process (see Section 6.5). While we do believe that the choice of metric and the relationship between the metric and the constraints both play an important role in determining the effectiveness of steering, in practice, the set of constraints chosen showed the clearest correlation to the resulting precision. Therefore, if steering results in a large number of false failure verdicts, we would first recommend that testers experiment with different sets of constraints until the number of false failures has decreased (without covering up known faults).

6.3 Masking of Faults

As steering changes the behavior of the oracle and can result in a new test verdict, a risk is that it will mask actual faults in the system. Such a danger is concerning, but with the proper choice of steering policies and constraints, we hypothesize that such a risk can be reduced to an acceptable level.

As can be seen in Table 6, when steering the Infusion_Mgr model, we changed a fault-induced “fail” verdict to “pass” in forty-three tests. This is a relatively small number—only 0.3% of the 15,200 test executions. This, according to Table 8, results in a drop in recall from 1.00 (for accepting the initial verdict) to 0.98. For the Pacing model, as shown in Table 7, steering adjusted a fault-induced failure to a pass for a small, but slightly higher, percentage of test executions—258 runs, or 2.9% of the test executions. The resulting recall is 0.94 for steering (Table 8).

Although any loss in recall is cause for concern when working with safety-critical systems, given the small number of incorrectly adjusted test verdicts for both systems, we believe that it is unlikely for an actual fault to be entirely masked by steering on *every* test in which the fault would otherwise lead to a failure. Of course, we still would urge care when working with steering.

Just as the choice of tolerance constraints can explain cases where steering is unable to account for an allowable non-conformance, the choice of constraints has a large impact on the risk of fault-masking. At any given execution step, steering, as we have defined here, considers only those oracle post-states as candidate targets that are reachable from the the given oracle pre-state. However, this by itself is not sufficiently restrictive to rule out truly deviant behaviors. Therefore, the constraints applied to reduce that search space must be strong enough to prevent steering from forcing the oracle into an otherwise impermissible state for that execution step. It is, therefore, crucial that proper consideration goes into the choice

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11,311 (74.4%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	312 (2.1%)	1,123 (7.4%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	598 (3.9%)	1,688 (11.1%)

TABLE 9: Results for step-wise filtering, (outputs + volume infused oracle), for Infusion_Mgr. Raw number, followed by percent of total.

Technique	Precision	Recall	Accuracy
No Adjustment	0.64	1.00	0.78
Filtering	0.64	0.76	0.70
Steering (Both Metrics)	0.94	0.98	0.96

TABLE 10: Precision, recall, and accuracy values for filtering (outputs + volume infused oracle) for Infusion_Mgr.

of constraints. In some cases, the use of additional policies—such as not steering the oracle model at all if it does not result in an exact match with the system—can also lower the risk of tolerating behaviors that would otherwise indicate faults.

Note that a seeded fault could *cause* a timing deviation (or the same behavior that would result from a timing deviation). In those cases, the failure is still labeled as being induced by a fault for our experiment. However, if the fault-induced deviation falls within the tolerances, steering will be able to account for it. In real world cases, where the faults are not purposefully induced, it is unlikely that even a human oracle would label the outcome differently, as they are working from the same system and domain knowledge that the tolerance constraints are derived from.

In real-world conditions, if care is taken when deriving the tolerance constraints from the system requirements, steering should not cover any behaviors that would not be permissible under those same requirements. Still, as steering carries the risk of masking faults, we recommend that it be applied as a *focusing tool*—to point the developer toward test failures likely to indicate faults so that they do not spend as much time investigating non-conformance reports that turn out to be allowable. The final verdict on a test should come from a run of the oracle model with no steering, but during development, steering can be effective at streamlining the testing process by concentrating resources on those failures that are more likely to point to faults.

6.4 Steering vs. Filtering

In some cases, acceptable non-conformance events could simply be dealt with by applying a filter that, in the case of a failing test verdict, checks the resulting state of the SUT against a set of constraints and overrides the initial oracle verdict if those constraints are met. Such filters are relatively common in GUI testing [24].

The use of a filter is tempting—if the filter is effective, it is likely to be easier to build and faster to execute than a full steering process. Indeed, for Infusion_Mgr, the results in Table 6 appear initially promising. The filter performs identically to steering for the initial failures that result from non-deterministic timing differences. It does not issue a pass verdict for timing issues outside of the tolerance limits, and

it does issue a pass for almost all of the tests where non-conformance is within the tolerance bounds. As can be seen in Table 8, the use of a filter increases the precision from 0.64 for no verdict adjustment to 0.89.

However, when the results for tests that fail due to faults are considered, a filter appears much less attractive. The filter issues a passing verdict for 1,252 tests that should have failed—1,209 more than steering. This is because a filter is a *blunt instrument*. It simply checks whether the state of the SUT meets certain constraints when non-conformance occurs. This allowed the filter to account for the allowed non-conforming behaviors, but these same constraints also allowed a large selection of fault-indicating tests to pass.

This makes the choice of constraints even more important for filtering than it is in steering. The steering process, by backtracking the state of the system, is able to ensure that the resulting behavior of the SUT is even possible (that is, if the new state is reachable from the previous state). The filter does not check the possibility of reaching a state; it just checks whether the new state is globally acceptable under the given constraints. As a result, steering is far more accurate. A filter could, of course, incorporate a reachability analysis. However, as the complexity of the filter increases, the reasons for filtering instead of steering disappear.

In fact, even the initial success of filtering at accounting for allowable non-conformance is somewhat misleading for the Infusion_Mgr case example. Both filtering and steering base their decisions on the output variables of the SUT and oracle, on the basis that the internal state variables may differ between the two. For this case study, all of the output variables reflect *current conditions* of the infusion pump—how much drug volume to infuse *now*, the *current* system mode, and so forth. Internally, these factors depend on both the current inputs and a number of *cumulative factors*, such as the total volume infused and the remaining drug volume. Over the long term, non-conformance events between the SUT and model will build, eventually leading to wider divergence. For example, the SUT or the model-based oracle may eventually cut off infusion if the drug reservoir empties. While a filter may be a perfectly appropriate solution for static GUIs, the cumulative build-up of differences in complex systems, will likely render a filter ineffective on longer time scales.

As the output variables reflect current conditions for this system, mounting internal differences may be missed, and the filter may not be able to cope with larger behavior differences that result from this steady divergence. Steering is able to prevent these long-term divergences by *actually changing the state of the oracle* throughout the execution of the test. A filter simply overrides the oracle verdict. It does not change the state of the oracle, and as a result, a filter cannot predict or handle behavioral divergences once they build beyond the set of constraints that the filter applies.

We can illustrate this effect by adding a single internal variable to the set of variables considered when making filtering or steering conditions—a variable tracking the total drug volume infused. Adding this variable causes *no change* to the results of steering seen in Table 6. However, the addition

of this internal variable dramatically changes the results of filtering. The new results can be seen in Tables 9 and 10.

Because the total volume infused increases over the execution of the test, it will reflect any divergence between the model-based oracle and the SUT. As steering actually adjusts the execution of the model-based oracle, this volume counter also adjusts to reflect the changes induced by steering. Thus, steering is able to account for the growing difference in the volume infused by the model-based oracle and the volume infused by the SUT. However, as the filter makes no such adjustment, it is unable to handle the mounting difference in this variable (or any other considered variable that reflects change over time). The filter, even if initially effective, will fail to account for a large number of acceptable non-conformance events—ultimately resulting in a precision value no more effective than not doing anything at all (and a far lower recall).

Similar results can be seen for the Pacing system in Table 7. The output variables of the Pacing example include both immediate commands, but also scheduled times for the next pacing events in both heart chambers. As a result, the output variables reflect internally-growing divergences between the model and SUT far more quickly than they appear in Infusion_Mgr. Thus, the precision of filtering is far lower than that of steering for the Pacing system, as the filter struggles to keep up with the time-dependent changes that mount over the execution of the test case. When we moved the internal volume counter variable to the outputs of Infusion_Mgr, we saw precision fall and recall rise. Similarly, for Pacing, precision is far lower for the filter than for steering, but the loss in recall is quite small as a result. The filter does not allow many divergences to pass—legal or illegal. Thus, filtering actually has a slightly higher level of recall than steering. However, it comes at a far higher cost to precision.

These results are not intended to indicate that indirect actions like filtering are *never* useful. When state does not have a major, persistent impact, or when the sources of non-determinism are limited, then a filter may be appropriate—and will be easier to implement and use. However, in the system domains we are interested in, state is important, and non-determinism is common. Therefore, direct actions, such as steering, are better able to handle the complex divergences between model and SUT.

6.5 Impact of Tolerance Constraints

The tolerance constraints limit the choice of steering action. In essence, they define the specification of what non-determinism we will allow, bounding the variance between the model and SUT that can be corrected. As emphasized in Section 3.3, the selection of an appropriate set of constraints is likely a crucial factor in the success or failure of steering. If tolerance constraints are too strict, we hypothesize that they will be useless for correcting the allowable behavioral deviations; too loose, and dangerous faults may be masked. We saw hints of this in the initial experiment, which utilized strict tolerance constraints. We masked only a vanishingly small number of faults, but we also failed to account for a small number of tests that should have been handled. The choice of constraints

Initial Verdict	Pass (Post-Steering)				Fail (Post-Steering)				
	Strict	Medium	Minimal	No Input Constraints	Strict	Medium	Minimal	No Input Constraints	
Pass		1,270 (74.9%)				0 (0.0%)			
Fail (Due to Timing, Within Tolerance)	161 (9.4%)	156 (9.2%)	176 (10.4%)	180 (10.6%)	19 (1.1%)	24 (1.4%)	4 (0.2%)	0 (0.0%)	
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	30 (1.8%)		35 (2.1%)	35 (2.0%)	5 (0.3%)		0 (0.0%)	
Fail (Due to Fault)	0 (0.0%)		1 (0.1%)	92 (5.4%)	210 (12.4%)	209 (12.3%)		118 (7.0%)	

TABLE 11: Results for steering with various sets of constraints for the Infusion_Mgr system.

Initial Verdict	Pass (Post-Steering)			Fail (Post-Steering)		
	Strict	Medium/Minimal	No Input Constraints	Strict	Medium/Minimal	No Input Constraints
Pass	308 (28.0%)			0 (0.0%)		
Fail (Due to Timing, Within Tolerance)		284 (25.8%)	34 (3.1%)		22 (2.0%)	272 (24.7%)
Fail (Due to Timing, Not in Tolerance)	3 (0.2%)	70 (6.3%)	6 (0.5%)	81 (7.3%)	14 (1.2%)	78 (7.1%)
Fail (Due to Fault)	30 (2.7%)	40 (3.6%)	46 (4.1%)	372 (33.8%)	362 (32.9%)	356 (32.4%)

TABLE 12: Results for steering with various constraint sets for the Pacing system.

Technique	Precision	Recall	Accuracy
No Adjustment	0.58	1.00	0.73
Filtering	0.89	0.67	0.76
Steering - Strict	0.93	1.00	0.96
Steering - Medium	0.90	0.88	0.89
Steering - Minimal	0.98	0.85	0.91
Steering - No Input Constraints	1.00	0.48	0.65

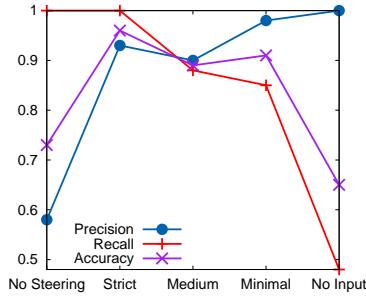


TABLE 13: Infusion_Mgr precision, recall, and accuracy for different tolerance constraint levels—as well as filtering and no adjustment.

Technique	Precision	Recall	Accuracy
No Adjustment	0.61	1.00	0.76
Filtering	0.72	0.93	0.81
Steering - Strict	0.95	0.93	0.94
Steering - Medium	0.94	0.77	0.85
Steering - Minimal	0.94	0.77	0.85
Steering - No Input Constraints	0.62	0.89	0.73

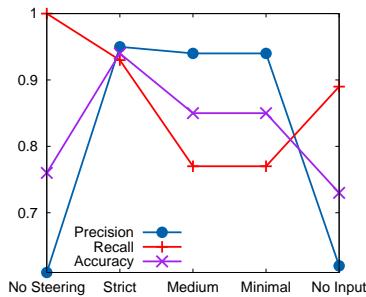


TABLE 14: Pacing precision, recall, and accuracy values for different tolerance levels—as well as filtering and no adjustment.

was a key factor in both the success of steering—not masking faults—and the limitations of the process—not handling all acceptable tests.

Given the apparent importance of the selection of tolerance constraints, we wished to determine the exact impact of the choice of tolerance constraints. One advantage of using steering over, say, directly modeling non-determinism is that, by utilizing a separate collection of rules to specify the bounds on acceptable non-deterministic deviations, we can easily *change* the constraints. By swapping in a new constraint file and re-executing the test suite, one can examine the effects of steering with the new limitations. For both Infusion_Mgr and Pacing, we took the implementations and five mutants for the original and each implementation and executed the test suite using four different sets of constraints. These are detailed in Section 5.4, and represent a steady loosening of the constraints from *strict* to *no constraints at all*.

Precision, recall, and accuracy results for Infusion_Mgr can be seen in Table 13. Exact values for accepting the initial verdict, filtering, and strict constraints differ slightly from those in Table 8, due to the lower number of mutants used, but the trends remain the same. As detailed in Table 11, steering with **strict** constraints improves precision by quite a bit by correcting almost all of the acceptable behavioral deviations, while masking no faults. Filtering, as in Section 6.4, improves precision, but at the cost of covering up many faults (resulting in a lower recall value).

As we loosen the constraints to the **medium** level—detailed in Table 11—we see a curious drop in precision. A small number of additional tests that fail due to acceptable non-determinism still fail after steering. It is likely that this is due to the sudden availability of additional steering actions. When presented with more choices, the search process chooses one of the several that minimizes the dissimilarity metric *now*, but causes side effects later on. We will revisit this when examining the results for the Pacing system. The recall also dips significantly. Inspecting Table 11 makes the reason for this clear, when given more freedom to adjust the timer values, the steering process will naturally cover up unacceptable timing differences. This underlines the importance of selecting constraints carefully.

When examining the results with **minimal** constraints and when there are **no constraints on the input variables** for Infusion_Mgr, shown in Table 11, two clear trends emerge. The first is that, as the constraints loosen, the precision rises. Naturally, given the freedom to make larger and larger adjustments to the selected steerable variables, the search process can handle more and more of the tests that fail due to acceptable deviations. Unsurprisingly, this increase in precision comes at a heavy cost to recall. With minimal constraints, we now not only can handle more of the acceptable deviations, but we also cover up many of the unacceptable deviations. Fortunately, we still effectively *do not mask code-based faults*. This is an encouraging result for steering. Although minimal constraints do cover the bad behaviors induced by non-determinism, we are still not masking issues within the code of the system.

That changes when we move to steering with **no input constraints** on the input variables of Infusion_Mgr. Now, not only can we handle all of the timing-based failures—acceptable or illegal—we also mask many of the induced faults as well. This is not unexpected. Given complete freedom to deviate from the original test inputs, guided only by the use of the dissimilarity metric, steering will mask many illegal behaviors. This is a clear illustration of the importance of selecting the correct constraints. Give too much freedom, and faults will be masked; too little freedom, and acceptable deviations will distract testers. It is important to experiment and strike the correct balance. Fortunately, our initial set of tolerances seems to have hit a reasonable balance point for Infusion_Mgr.

The precision, recall, and accuracy figures for the Pacing system appear in Table 14. Again, the results for no adjustment, filtering, and **strict** steering follow the same trends as the earlier experiment (see Table 8). Filtering and steering do equally well on recall, but steering achieves far higher precision. The filter is unable to keep up with the behavior divergences that build over time, while steering keeps up by adjusting execution each time behaviors diverge.

As we shift to the **medium**, **minimal**, and **no input constraint** results—detailed in Table 12—we see an interesting divergence from the results for Infusion_Mgr. Namely, that rather than improving, the precision actually significantly drops—from 0.95, to 0.94, and finally to 0.62. Steering the Pacing model with no input constraints is barely more accurate on the legal divergences than not steering at all.

By loosening the constraints, we gave the steering algorithm more freedom to manipulate the input values. On the Infusion_Mgr system, this resulted in us being able to handle more and more of the acceptable behavior differences, but at the cost of also covering up more and more of the unacceptable differences. On Pacing, we cover more faults as the constraints are loosened, but steering actually grows far *less capable* at accounting for the acceptable differences. This can be explained by examining the steerable variables for the Pacing system, detailed in Table 15.

Unless particularly specific conditions are met, changes to many of the steerable variables for Pacing will have a delayed impact on the behavior of the system. For example, altering the length of one of the refractory periods will only immediately

impact behavior if we are in a refractory period and decrease that period to be less than the current duration. To give a second example, enabling or altering ATR mode settings will only alter behavior if we are already in ATR mode, and even then, likely only after we have been in it for a longer period of time. Therefore, given very loose constraints (or worse, no constraints), it is incredibly easy for the steering algorithm to configure the immediately-effective variables to minimize the dissimilarity score, but to also alter one of the delayed variables in such a way that it eventually drives the model to diverge from the system. Infusion_Mgr, too, has prescription variables that can have delayed effects, but many of those could be adjusted again to “fix” the side effect. For Pacing, many of these side effects can only be “fixed” after they have damaged conformance.

This issue further highlights the importance of choosing good constraints, as many of the steering induced changes that can cause eventual side effects are also changes that *would not address hardware or time-based non-determinism*. Intuitively, changes to the majority of possible timing issues with Pacing would be restricted to a small number of those input variables and—even then—would only require small adjustments. You may want to correct a small delay in pacing, or slightly shift the end of a refractory period that has lasted too long, but it is unlikely that you would want to steer either of those factors by a significant amount, as the end result of a software fault could impact the health of a patient.

The possibility of choosing a steering action with an undesirable delayed side effect points to the need for further research on both tolerance constraints and dissimilarity metrics. We may want to add in a penalty factor on changes to certain variables to bias the search algorithm towards first trying the variables with immediate effects. We may also want to judge the impact of a steering action on both the immediate differences between the model and SUT and the impact on *eventual* differences. However, checking both current and future behavior is a difficult challenge, as the computational requirements to perform such a comparison may not be realistically obtainable.

Ultimately, what we see from both systems is that the choice of tolerance constraints is crucially important in determining the capabilities and limitations of steering. Relatively strict constraints seem to offer the best balance between accounting for acceptable deviations and masking fault-indicative behavior. As constraints loosen, we run a significantly increased risk of masking faults or choosing steering actions with undesirable long-term side effects. That said, the key to determining a reasonable set of steering constraints is in understanding the requirements of the system being built and the domain the device must operate within. Choosing the correct set of constraints is important, but it is a task that reasonably experienced developers should be capable of conducting. Our framework allows experimentation with different sets of constraints, allowing developers to find and tune the tolerance constraints. By using domain knowledge and the software specifications to build reasonable, well-considered constraints, we can use steering to enable the use of model-based oracles and focus the attention of the developers.

Variable Name	Explanation	When Behavior is Impacted
IN_V_EVENT	Sensed event indicator for ventricle chambers	Immediate
IN_A_EVENT	Sensed event indicator for atrial chambers	Immediate
IN_EVENT_TIME	Time of sensor poll	Immediate
IN_SYSTEM_MODE	Current system mode	Immediate
IN_LRL	Lower rate limit on paces	Likely Delayed
IN_URL	Upper rate limit on paces	Likely Delayed
IN_HYSTERESIS_RL	Optional adaptation of artificial pacing rate to natural pacing	Likely Delayed
IN_VRP	Ventricular refractory period following a ventricular event	Likely Delayed
IN_ARP	Atrial refractory period following an atrial event	Likely Delayed
IN_PVARP	Atrial refractory period following a ventricular event	Likely Delayed
IN_PVARP_EXTENSION	Optional extension on PVARP following certain events	Likely Delayed
IN_FIXED_AVD	Fixed timing window between atrial event and ventricular reaction	Likely Delayed
IN_DYNAMIC_AVD	Enables a dynamic timing window between atrial events and ventricular reactions	Likely Delayed
IN_DYNAMIC_AVD_MIN	Minimum dynamically-determined value for AVD window	Likely Delayed
IN_ATR_MODE	Enables special mode to ease patient out of pacemaker-induced atrial tachycardia	Delayed
IN_ATR_DURATION	Defines minimum period before entering ATR mode	Likely Delayed
IN_ATR_FALLBACK_TIME	Defines duration of ATR mode	Likely Delayed

TABLE 15: Inputs for the Pacing system, explanations of their utility, and when adjustments to those variables will impact observable system behavior.

6.6 Automatically Deriving Tolerance Constraints

As indicated in the previous section, tolerance constraints play an important role in the success of steering. Selecting the right constraints is clearly important; yet, one can imagine scenarios where the developers are uncertain of what boundaries to set or even what variables to loosen or constrain. Thus, we were interested in investigating whether constraints can be *learned* from steering against developer-classified test cases.

We took one of the time-delayed implementations of Infusion_Mgr (called “PBOLUS”) and the implementation of Pacing, steered using no input constraints for both dissimilarity metrics, and derived ten sets of tolerance constraints using the learning process described in Section 4. As the same learning process can also be applied to refine existing constraints, we repeated the same process for the strict, medium, and minimal constraint sets.

The results for PBOLUS are shown in Table 16, where the reported calculations for the learned constraints are the median of ten trials. As expected, making no adjustments to the verdicts when steering PBOLUS results in the lowest precision. As we did not include any of the implementations with seeded faults in this experiment, filtering performs very well—handling the timing fluctuations specific to this implementation with relative ease. Across the board, the results for the learned constraints are very positive, on average generally matching or exceeding filtering.

When learning from the strict, medium, or minimal constraints, the learned constraints can only be a *tightening* of the constraints being learned from. That is, if particular variables are already locked down, then those variables will not suddenly be loosened. Variables that have a tolerance window will only have that window remain the same size or have that window shrink—learning will not further open that window. Thus, a certain ceiling effect forms where, if a developer missed a variable that should have been steerable, then the tightening can only help a small amount with performance. Here, that performance ceiling for tightening seems to line up with the performance of a filter. This was hinted at for the Infusion_Mgr system in Section 6.4, where the filter performed

```
((real(IN_EVENT_TIME) = concrete_oracle_IN_EVENT_TIME)
or ((real(IN_EVENT_TIME) >=
concrete_oracle_IN_EVENT_TIME + 2.000000) and
(real(IN_EVENT_TIME) <= concrete_oracle_IN_EVENT_TIME
+ 3.000000)))
((real(IN_AVD_OFFSET) >= concrete_oracle_IN_AVD_OFFSET)
and (real(IN_AVD_OFFSET) <= concrete_oracle_IN_AVD_OFFSET
+ 1.000000))
(real(IN_ARP) = concrete_oracle_IN_ARP)
(real(IN_VRP) = concrete_oracle_IN_VRP)
...
(real(IN_URL) = concrete_oracle_IN_URL)
(real(IN_LRL) = concrete_oracle_IN_LRL)
```

Fig. 9: Sample constraints learned for Pacing.

as well as strict steering for the allowable deviations. Here, we see the same effect—when learning from existing constraints, we can only improve performance to a limited degree.

This tightening may still be useful if a developer is sure of the variables to steer, but unsure of the bounds to set on those variables. However, the results of learning from *no preexisting input constraints* are interesting because they do not have this performance limitation. Given just a set of classified tests with no input constraints, we are free to derive constraints on any variables and freely set those boundaries. As a result, for the PBOLUS system, the best results emerge when given this freedom, with median steering performance after learning from no input constraints beating steering after learning from strict tolerance constraints on precision by up to 17%. This suggests that the strict constraints may actually be stricter than they need to be, potentially missing variables that should be adjustable.

Note that we did see minor differences between the executions using the Manhattan dissimilarity metric and the Squared Euclidean metric. However, given their identical performance in prior experiments, we believe the differences noted here are due to the stochastic nature of the learning process, rather than a difference induced by the choice of metric.

At first, the results for the learned constraints for Pacing—shown in rows 3-10 of Table 16—appear very poor. The constraints learned for Pacing score a median precision of around 0.26 in almost all cases, and as low as 0.24. Filtering

Technique	Infusion_Mgr			Pacing		
	Precision	Recall	Accuracy	Precision	Recall	Accuracy
No Adjustment	0.18	1.00	0.30	0.24	1.00	0.39
Filtering	0.70	1.00	0.82	0.32	1.00	0.48
Learned from Strict (M)	0.70	1.00	0.82	0.26	1.00	0.42
Learned from Strict (SE)	0.65	1.00	0.77	0.26	1.00	0.42
Learned from Medium (M)	0.70	1.00	0.82	0.26	1.00	0.42
Learned from Medium (SE)	0.70	1.00	0.82	0.26	1.00	0.42
Learned from Minimal (M)	0.73	1.00	0.84	0.26	1.00	0.42
Learned from Minimal (SE)	0.76	1.00	0.86	0.26	1.00	0.42
Learned from No Tolerances (M)	0.82	1.00	0.89	0.24	1.00	0.39
Learned from No Tolerances (SE)	0.76	1.00	0.86	0.26	1.00	0.40
Learned, All Tolerances, After Widening (M/SE)				0.75	0.88	0.81

TABLE 16: Median precision, recall, and accuracy values for learned tolerance constraints for each system. M=Manhattan, SE=Squared Euclidean dissimilarity function.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Fail (Due to Timing, Within Tolerance)	9	67
Fail (Due to Timing, Not in Tolerance)	0	24

TABLE 17: Distribution of results for steering with tolerance constraints learned from strict constraints for the Pacing system. Raw number of test results.

does not do well, either, but does lead the pack with a precision of 0.32. We can see why the learning results are poor by examining the detailed test executions, listed in Table 17. The learned constraints, in almost all cases, are extremely strict. They never allow illegal behaviors to pass, but they also fail to compensate for the majority of the legal deviations.

Interestingly, if we look at the learned constraints (an example set is listed in Figure 9), we see that the learning process has actually picked up on the *correct variables* to set constraints on. In particular, it almost universally allowed the sensed event indicators to be free and allowed a small window of adjustment on the event time. However, it set *too strict* of a limit on how much those variables could be adjusted. This is actually an easy “issue” to correct. As mentioned a number of times, the use of a separate tolerance constraint file means that it is easy to experiment with different constraints. Given that we are using a set of classified test results, we can simply adjust the tolerances and re-execute the tests until the performance meets a desirable threshold.

Such an adjustment can be done automatically by systematically shrinking or widening the learned tolerances until this threshold is met. In this case, we took the constraints and increased the bounds by one on both ends. For example, the IN_EVENT_TIME tolerance listed in Figure 9 transforms from 2-3 seconds to allowing an adjustment anywhere from 1-4 seconds. Even this small adjustment leads to markedly improved results, as can be seen in the last row of Table 16. Across the board, this small adjustment led to a median accuracy result of 0.81—far higher than no adjustment, filtering, or the original learned constraints.

The results of learning tolerance constraints seem quite positive. Given a set of classified tests, we are able to extract a small, strict set of constraints that can be used—perhaps after a small amount of tuning—to successfully steer a model.

6.7 Summary of Results

The precision, recall, and F-measure for each method—accepting the initial verdict, steering (using two different dissimilarity metrics), and filtering—are shown in Table 8.

The default situation, accepting the initial verdict, results in the lowest precision value. Intuitively, not doing anything to account for allowed non-conformance will result in a large number of incorrect “fail” verdicts. However, the default practice does have the largest recall value. Again, not adjusting your results will prevent incorrect masking of faults. Filtering on a step-by-step basis results in higher precision than doing nothing, but due to the lack of reachability analysis and state adaptation—both of which used by the steering approach—the filter masks an unacceptably large number of faults for Infusion_Mgr. For Pacing, filtering is unable to keep up with the complex divergences that build over time. Although it is able to improve the level of precision over not adjusting the verdict, it fails to match the precision gains seen when steering.

Steering performs identically for both of the dissimilarity metrics used in this study. It is able to adapt the oracle to handle almost every situation where non-conforming behaviors are allowed by the system requirements, while masking only a few faults in a small number of tests. For both systems, steering results in a large increase in precision, with only a small cost in recall.

We find that steering results in the highest accuracy for the final test results for both systems. Steering demonstrates a higher overall accuracy—balance of precision and recall—than filtering or accepting the initial verdict, 0.96 to 0.64 and 0.78 for Infusion_Mgr and 0.95 to 0.86 and 0.82 for Pacing.

Tolerance constraints play a large role in determining the efficacy of steering, both limiting the ability of steering to mask faults and its ability to correct acceptable deviations. Relatively strict, well-considered constraints strike the best balance between enabling steering to focus developers and preventing steering from masking faults. As constraints are loosened, we observed that steering may be able to account for more and more acceptable deviations, but at the cost of also masking many more faults. Alternatively, loose constraints may also impair steering from performing its job by allowing the search process to choose a steering action that causes eventual side-effects.

Fortunately—even if developers are unsure of what variables to set constraints on—as long as they can classify the outcome of a set of tests, a set of constraints can automatically be learned. For our case examples, the derived set of constraints was small, strict, and able to successfully steer the model (albeit, for Pacing, with a small amount of tuning).

Steering is able to automatically adjust the execution of the oracle to handle non-deterministic, but acceptable, behavioral

divergence without covering up most fault-indicative behaviors. We, therefore, recommend the use of steering as a tool for focusing and streamlining the testing process.

7 THREATS TO VALIDITY

External Validity: Our study is limited to two case examples. Although we are actively working with domain experts to produce additional systems for future studies, we believe that the systems we are working with to be representative of the domains that we are interested in, and that our results will generalize to other systems in this domain.

We have used Stateflow, translated to Lustre, as our modeling language. Other modeling notations can be used to steer. We do not believe that the modeling language chosen has a significant impact on the ability to steer the model. Similarly, we have used Lustre as our implementation language, rather than more common languages such as C or C++. However, systems written in Lustre are similar in style to traditional imperative code produced by code generators used in embedded systems development. A simple syntactic transformation is sufficient to translate Lustre code to C code.

We have limited our study to fifty mutants for each version of the case example, resulting in a total of 150 mutants for the Infusion_Mgr system and 100 for the Pacing system. These values are chosen to yield a reasonable cost for the study, particularly given the length of each test. It is possible the number of mutants is too low. Nevertheless, we have found results using less than 250 mutants to be representative for similarly-sized systems [45], [42].

Internal Validity: Rather than develop full-featured system implementations for our study, we instead created alternative versions of the model—introducing various non-deterministic behaviors—and used these models and the versions with seeded faults as our “systems under test.” We believe that these models are representative approximations of the behavioral differences we would see in systems running on embedded hardware. In future work, we plan to generate code from these models and execute the software on actual hardware platforms.

In our experiments, we used a default testing scenario (accepting the oracle verdict) and stepwise filtering as baseline methods for comparison. There may be other techniques—particularly, other filters—that we could compare against. Still, we believe that the filter chosen was an acceptable comparison point, and was designed as such a filter would be in practice.

Construct Validity: We measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software. It is possible that using real faults would lead to different results. Nevertheless, Andrews et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults in similar fault finding experiments [43].

8 RELATED WORK

Model-based testing (MBT) is a formal method that uses models of software systems for the derivation of test suites [1]. Such techniques commonly take a model in the form of

a labeled transition system (for example, a finite state machine) [46], [4] and generate a series of tests to apply to the SUT. An attempt is made to establish conformance between the model and the system [47]. Much of the research on model-based testing is concerned explicitly with the generation of test input, although some have explored model-based oracle generation [48]. Such models serve implicitly as oracle on the generated tests, being the basis on which correctness is judged.

Several authors have examined the use of behavioral models as test-generation targets for real-time systems [49], [22], [3], [50], [51]. These models are designed to handle limited forms of non-deterministic behavior—allowing some flexibility in terms of the time that output events occur [22], [49], [50], [51]. Arcuri et al. also model the impact of non-deterministic hardware failures [49]. Many of these approaches make use of non-deterministic or special time modeling formalism, such as UPPAAL [52].

Oracle steering is conceptually similar to *dynamic* program steering, the automatic guidance of program execution [25], [8]. Much of the research in dynamic program steering is concerned with automatic adaptation to maintain consistent performance or a certain reliability level when faced with depleted computational resources [25]. Kannan et al. have proposed a framework to assure the correctness of software execution at runtime through corrective steering actions [53]. Although their framework bears similarities to what we are proposing, our goals are very different—rather than adjusting the behavior of the live system, we apply steering to the test oracle in order to better identify fault-indicative behaviors. The Spec Explorer [9] test generation framework explores the possible runs of the executable model by applying steering actions in order to guide the model through various execution scenarios. It can then use the model as an oracle for the generated test by checking whether the SUT produces the same behaviors. Although Spec Explorer also makes use of steering to guide the execution of behavioral models, their application and goals differ from ours. They use steering to create tests, but the final test cases are deterministic. Steering is not applied when checking conformance. As with the other approaches to model-based testing of real-time systems discussed in this section, Spec Explorer may be able to address some of the issues we are concerned with, but it also suffers from the same limitations of being locked into a particular model format and requiring that non-determinism be built into that model.

9 CONCLUSION AND FUTURE WORK

Specifying test oracles is still a major challenge for many domains, particularly those—such as real-time embedded systems—where issues related to timing, sensor inaccuracy, or the limited computation power of the embedded platform may result in non-deterministic behaviors for multiple applications of the same input. Behavioral models of systems, often built for analysis and simulation, are appealing for reuse as oracles. However, these models typically present an abstracted view of system execution that may not match the execution reality. Such models will struggle to differentiate unexpected—but still acceptable—behavior from behaviors indicative of a fault.

To address this challenge, we have proposed an automated *model-based oracle steering framework* that, upon detecting a behavioral difference, backtracks and selects—through a search-based process—a *steering action* that will bring the model in line with the execution of the system. To prevent the model from being forced into an illegal behavior—and masking a real fault—the search process must select an action that satisfies certain constraints and minimizes a dissimilarity metric. This framework allows non-deterministic, but bounded, behavior differences while preventing future mismatches by guiding the model, within limits, to match the execution of the SUT.

Experiments, conducted over complex real-time systems, have yielded promising results and indicate that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing development costs. The use of our steering framework can allow developers to focus on behavioral difference indicative of real faults, rather than spending time examining test failure verdicts that can be blamed on a rigid oracle model.

There is still much room for future work:

- We plan to further examine the impact of different dissimilarity metrics and tolerance constraints on oracle verdict accuracy;
- Policies defining when to attempt to steer could deeply impact the resulting testing process—we plan to define additional steering policies and explore their use;
- The need to invoke constraint solvers multiple times throughout execution limits the performance of the steering framework. We seek improvements to speed and scalability, and plan to experiment with the use of meta-heuristic optimization algorithms in place of multiple calls to an exhaustive solver;
- We would like to examine the use of steering and dissimilarity metrics as methods of quantifying non-conformance and their utility in fault identification and location;
- And, we plan to examine the use of steering to debug faulty or incomplete oracle models.

10 SOURCE CODE AND DATA

In the interest of allowing others to extend, reproduce, or otherwise make use of the work that we have conducted, the research prototype of our steering framework and experimental data—including the models, mutants, and tests—have been made freely available under the Mozilla Public License 2.0.

- 1) Experimental data for each system is available from the PROMISE repository [54]. This includes the original Stateflow models, the Lustre translation, fault-seeded mutants, and randomly-generated tests used in our experiments.
 - a) The Infusion_Mgr data can be found at <http://openscience.us/repo/test-generation/manager.html>
 - b) The Pacing data can be found at <http://openscience.us/repo/test-generation/pacing.html>.
- 2) The source code of the steering framework—and binaries of the required dependencies—can be obtained from <https://github.com/Greg4cr/Steering-Framework>.

REFERENCES

- [1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey on automated software test case generation,” *Journal of Systems and Software*, vol. 86, pp. 1978–2001, August 2013.
- [2] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, pp. 507–525, May 2015.
- [3] A. En-Nouaary, R. Dssouli, and F. Khendek, “Timed wp-method: testing real-time systems,” *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1023–1038, Nov.
- [4] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines—a survey,” *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.
- [5] “MathWorks Inc. Stateflow.” <http://www.mathworks.com/stateflow, 2015>.
- [6] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Polit, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, “Statemate: A working environment for the development of complex reactive systems,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 403–414, April 1990.
- [7] “IBM Rational Rhapsody.” [http://www.ibm.com/developerworks/rational/products/rhapsody/, 2014](http://www.ibm.com/developerworks/rational/products/rhapsody/).
- [8] D. Miller, J. Guo, E. Kraemer, and Y. Xiong, “On-the-fly calculation and verification of consistent steering transactions,” in *Supercomputing, ACM/IEEE 2001 Conf.*, pp. 8–8, 2001.
- [9] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, “Model-based testing of object-oriented reactive systems with spec explorer,” in *Formal Methods and Testing* (R. M. Hierons, J. P. Bowen, and M. Harman, eds.), vol. 4949 of *Lecture Notes in Computer Science*, pp. 39–76, Springer, 2008.
- [10] G. Gay, S. Rayadurgam, and M. P. Heimdahl, “Steering model-based oracles to admit real program behaviors,” in *Proceedings of the 36th International Conference on Software Engineering – NIER Track*, ICSE ’14, (New York, NY, USA), ACM, 2014.
- [11] G. Gay, S. Rayadurgam, and M. P. Heimdahl, “Improving the accuracy of oracle verdicts through automated model steering,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE ’14, (New York, NY, USA), pp. 527–538, ACM, 2014.
- [12] W. Howden, “Theoretical and empirical studies of program testing,” *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 293–298, 1978.
- [13] E. Weyuker, “The oracle assumption of program testing,” in *13th International Conference on System Sciences*, pp. 44–49, 1980.
- [14] M. Staats, G. Gay, and M. Heimdahl, “Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing,” in *Proceedings of the 2012 Int’l Conf. on Software Engineering*, pp. 870–880, IEEE Press, 2012.
- [15] D. Coppit and J. Haddox-Schatz, “On the use of specification-based assertions as test oracles,” in *Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, SEW ’05, (Washington, DC, USA), pp. 305–314, IEEE Computer Society, 2005.
- [16] M. Pezzé and M. Young, *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.
- [17] B. Scientific, “Pacemaker system specification,” in *Pacemaker Formal Methods Challenge*, Software Quality Research Lab, 2007.
- [18] T. M. Association, “Modelica - a unified object-oriented language for systems modeling,” tech. rep., 2012.
- [19] Q. Xie and A. M. Memon, “Designing and comparing automated test oracles for gui-based software applications,” *ACM Trans. Softw. Eng. Methodol.*, vol. 16, Feb. 2007.
- [20] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and E. Watanabe, “Constructive model-based analysis for safety assessment,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 673–702, 2012.
- [21] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl, “Proving the shalls: Early validation of requirements through formal methods,” *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 4, pp. 303–319, 2006.
- [22] K. G. Larsen, M. Mikucionis, and B. Nielsen, “Online testing of real-time systems using UPPAAL,” in *International workshop on formal approaches to testing of software (FATES 04)*, Springer, 2004.
- [23] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-based testing for embedded systems*. CRC press, 2011.

- [24] S. Mahajan and W. G. Halfond, "Finding HTML presentation failures using image comparison techniques," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), pp. 91–96, ACM, 2014.
- [25] W. Gu, J. Vetter, and K. Schwan, "An annotated bibliography of interactive program steering," *ACM SIGPLAN Notices*, vol. 29, 1994.
- [26] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *International Journal of Mathematical Models and Methods in Applied Sciences*, vol. 1, no. 4, pp. 300–307, 2007.
- [27] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, pp. 31–88, Mar. 2001.
- [28] G. Hagen, *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [29] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems, Second Edition*. Cambridge Press, 2006.
- [30] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185, Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands: IOS Press, 2009.
- [31] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.
- [32] G. Gay, *Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors*. PhD thesis, University of Minnesota, May 2015.
- [33] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Houndations of Machine Learning*. MIT Press, 2012.
- [34] T. Menzies and Y. Hu, "Data mining for very busy people," *Computer*, vol. 36, pp. 22–29, Nov. 2003.
- [35] C. P. T. M. Johann Schumann, Karen Gundy-Burlet and A. Barrett, "Software v&v support by parametric analysis of large software simulation systems," in *2009 IEEE Aerospace Conference*, 2009.
- [36] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies, "Parametric analysis of antares re-entry guidance algorithms using advanced test generation and data analysis," in *9th International Symposium on Artifical Intelligence, Robotics and Automation in Space*, 2007.
- [37] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies, "Parametric analysis of a hover test vehicle using advanced test generation and data analysis," in *AIAA Aerospace*, 2009.
- [38] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet, "Automatically finding the control variables for complex system behavior," *Automated Software Engineering*, vol. 17, pp. 439–468, Dec. 2010.
- [39] S. D. Bay and M. J. Pazzani, "Detecting change in categorical data: Mining contrast sets," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, (New York, NY, USA), pp. 302–306, ACM, 1999.
- [40] A. Murugesan, S. Rayadurgam, and M. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.
- [41] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [42] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," 2008.
- [43] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608–624, aug. 2006.
- [44] N. Li and J. Offutt, "An empirical analysis of test oracle strategies for model-based testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 363–372, March 2014.
- [45] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 161–170, ACM, 2008.
- [46] E. Brinksma and J. Tretmans, "Testing transition systems: An annotated bibliography," in *Modeling and Verification of Parallel Processes* (F. Cassez, C. Jard, B. Rozoy, and M. Ryan, eds.), vol. 2067 of *Lecture Notes in Computer Science*, pp. 187–195, Springer Berlin Heidelberg, 2001.
- [47] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, pp. 1–38, Springer, 2008.
- [48] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, "Model-based test oracle generation for automated unit testing of agent systems," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1230–1244, Sept 2013.
- [49] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing software and systems*, pp. 95–110, Springer-Verlag, 2010.
- [50] T. Savor and R. Seviora, "An approach to automatic detection of software failures in real-time systems," in *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pp. 136–146, 1997.
- [51] L. Briones and E. Brinksma, "A test generation framework for quiescent real-time systems," in *International workshop on formal approaches to testing of software (FATES 04)*, pp. 64–78, Springer-Verlag GmbH, 2004.
- [52] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets* (J. Desel, W. Reisig, and G. Rozenberg, eds.), vol. 3098 of *Lecture Notes in Computer Science*, pp. 87–124, Springer Berlin Heidelberg, 2004.
- [53] S. Kannan, M. Kim, I. Lee, O. Sokolsky, and M. Viswanathan, "Runtime monitoring and steering based on formal specifications," in *Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, 2000.
- [54] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data," 2015.



Gregory Gay is an Assistant Professor of Computer Science & Engineering at the University of South Carolina. His research interests include automated testing and analysis—with an emphasis on test oracle construction—and search-based software engineering. Greg received his Ph.D. from the University of Minnesota, working with the Critical Systems research group, and an M.S. from West Virginia University.



Sanjai Rayadurgam is a Research Staff Member at the University of Minnesota Software Engineering Center in the Department of Computer Science and Engineering. His research interests are in software testing, formal analysis and requirements modeling, with particular focus safety-critical systems development, where he has significant industrial experience. He earned a B.Sc. in Mathematics from the University of Madras at Chennai, and in Computer Science & Engineering, an M.E. from the Indian Institute of Science at Bangalore and a Ph.D. from the University of Minnesota at Twin Cities.



Mats P.E. Heimdahl is a Professor and Department Head in Computer Science and Engineering at the University of Minnesota. He earned an M.S. in Computer Science and Engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden and a Ph.D. in Information and Computer Science from the University of California at Irvine. His research interests are in software engineering, safety critical systems, software safety, testing, requirements engineering, formal specification languages, and automated analysis of specifications.

He is the recipient of the NSF CAREER award, a McKnight Land-Grant Professorship, the McKnight Presidential Fellow award, and the awards for Outstanding Contributions to Post-Baccalaureate, Graduate, and Professional Education at the University of Minnesota.

The Fitness Function for the Job: Search-Based Generation of Test Suites that Detect Real Faults

Gregory Gay

Department of Computer Science & Engineering

University of South Carolina, USA

greg@greggay.com

Abstract—Search-based test generation, if effective at fault detection, can lower the cost of testing. Such techniques rely on fitness functions to guide the search. Ultimately, such functions represent test goals that approximate—but do not ensure—fault detection. The need to rely on approximations leads to two questions—*can fitness functions produce effective tests and, if so, which should be used to generate tests?*

To answer these questions, we have assessed the fault-detection capabilities of the EvoSuite framework and eight of its fitness functions on 353 real faults from the Defects4J database. Our analysis has found that the strongest indicator of effectiveness is a high level of code coverage. Consequently, the branch coverage fitness function is the most effective. Our findings indicate that fitness functions that thoroughly explore system structure should be used as primary generation objectives—supported by secondary fitness functions that vary the scenarios explored.

I. INTRODUCTION

Proper verification practices are needed to ensure that developers deliver reliable software. *Testing* is an invaluable, widespread verification technique. However, testing is a notoriously expensive and difficult activity [32], and with exponential growth in the complexity of software, the cost of testing has risen accordingly. Means of lowering the cost of testing without sacrificing verification quality are needed.

Much of that cost can be traced directly to the human effort required to conduct most testing activities, such as producing test input and expected output. We believe the key to lowering such costs lies in the use of automation to ease this manual burden [3]. Automation has great potential in this respect, as much of the invested human effort is in service of tasks that can be framed as *search* problems [21].

Test case generation can naturally be seen as a search problem [3]. There are hundreds of thousands of test cases that could be generated for any particular class under test (CUT). Given a well-defined testing goal, and a numeric scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can systematically search the space of possible test inputs to locate those that meet that goal [28].

The effective use of search-based generation relies on the performance of two tasks—selecting a measurable test goal and selecting an effective fitness function for meeting that goal. Adequacy criteria offer checklists of measurable test goals, such as the execution of branches in the control-flow of the CUT [26], [34], [35]. Often, however, goals such as “coverage

of branches” are an approximation of a goal that is harder to quantify—we really want tests that will reveal faults [1]. “Finding faults” is not a goal that can be measured, and cannot be cleanly translated into a distance function.

To generate effective tests, we must identify criteria—and corresponding fitness functions—that are correlated with an increased probability of fault detection. If branch coverage is, in fact, correlated with fault detection, then—even if we do not care about the concept of branch coverage itself—we will end up with effective tests. However, the need to rely on approximations leads to two questions. First, *can common fitness functions produce effective tests?* If so, *which of the many available fitness functions should be used to generate tests?* Unfortunately, testers are faced with a bewildering number of options—an informal survey of two years of testing literature reveals 28 viable fitness functions—and there is little guidance on when to use one criterion over another [16].

While previous studies on the effectiveness of adequacy criteria have yielded inconclusive results [33], [30], [23], [16], two factors now allow us to more deeply examine this problem—particularly with respect to search-based generation. First, tools are now available that implement enough fitness functions to make unbiased comparisons. The EvoSuite framework offers over twenty options, and uses a combination of eight fitness functions in its default configuration [11]. Second, more realistic examples are available for use in assessment of suites. Much of the previous work on adequacy effectiveness has been assessed using mutants—synthetic faults created through source code transformation [24]. Whether mutants correspond to the types of faults found in real projects has not been firmly established [19]. However, the Defects4J project offers a large database of real faults extracted from open-source Java projects [25]. We can use these faults to assess the effectiveness of search-based generation on the complex faults found in real software.

We have used EvoSuite and eight of its fitness functions (as well as the default multi-objective configuration) to generate test suites for the five systems, and 353 of the faults, in the Defects4J database. In each case, we recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. By analyzing these factors, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite, but—through the use of learning algorithms—the

factors correlating with a high likelihood of fault detection. To summarize our findings:

- Collectively, 51.84% of the examined faults were detected by generated test suites.
- EvoSuite’s branch coverage criterion is the most effective—detecting more faults than any other criterion and demonstrating a 11.06-245.31% higher likelihood of detection for each fault than other criteria.
- There is evidence that combinations of criteria could be more effective than a single criterion—almost all criteria uniquely detect one or more faults, and in cases where the top-scoring criterion performs poorly, at least one other criterion would more capably detect the fault.
- Yet, while EvoSuite’s default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all criteria.
- High levels of coverage over the fixed version of the CUT and over patched lines of code are the factors that most strongly correlate with suite effectiveness.
- While others have found that test suite size correlates to mutant detection, we found that larger suites are not necessarily more effective at detecting real faults.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, line, and weak mutation coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Almost all of the criteria were useful in *some* case and could be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite. More research is needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites. However, our findings represent a step towards understanding the use, applicability, and combination of common fitness functions.

II. BACKGROUND

A. Search-Based Software Test Generation

Test case creation can naturally be seen as a search problem [21]. Of the thousands of test cases that could be generated for any SUT, we want to select—systematically and at a reasonable cost—those that meet our goals [28], [1]. Given a well-defined testing goal, and a scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*) [5]. Metaheuristics are often inspired by natural phenomena, such as swarm behavior [7] or evolution [22].

Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world

program is large and complex [1]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [27]. Such approaches have been applied to a wide variety of testing goals and scenarios [1].

B. Adequacy Metrics and Fitness Functions

When testing, developers must judge: (a) whether the produced tests are effective and (b) when they can stop writing additional tests. These two factors are linked. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*? The same question applies when adding new tests—if we have not observed new faults, have we not yet written *adequate* tests?

The concept of adequacy provides developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of tests. The most common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, branches of the software’s control flow, and complex boolean conditional statements [26], [34], [35]. Each adequacy criterion embodies a set of lessons about effective testing—requirements tests must fulfill to be considered adequate. If tests execute elements in the manner prescribed by the criterion, than testing is deemed “adequate” with respect to faults that manifest through such structures. Adequacy criteria have seen widespread use in software development, and is routinely measured as part of automated build processes [20]¹.

It is easy to understand the popularity of adequacy criteria. They offer clear checklists of testing goals that can be objectively evaluated and automatically measured [36]. These very same qualities make adequacy criteria ideal for use as automated test generation targets, as they can be straightforwardly transformed into distance functions that guide to the search to better solutions [4]. Search-based generation has even achieved higher coverage than developer-created tests [13].

III. STUDY

To generate tests that are effective at finding faults, we must identify criteria and corresponding fitness functions that increase the probability of fault detection. As we cannot know what faults exist before verification, such criteria are approximations—intended to increase the probability of fault detection, but offering no guarantees. Thus, it is important to turn a critical eye toward the choice of fitness function used in search-based test generation. We wish to know whether commonly-used fitness functions produce effective tests, and if so, why—and under what circumstances—do they do so?

More empirical evidence is needed to better understand the relationships between adequacy criteria, fitness functions and fault detection [20]. Many criteria exist, and there is little guidance on when to use one over another [16]. To better

¹For example, see <https://codecov.io/>.

understand the real-world effectiveness, use, and applicability of common fitness functions and the factors leading to a higher probability of fault detection, we have assessed the EvoSuite test generation framework and eight of its fitness functions (as well as the default multi-objective configuration) against 353 real faults, contained in the Defects4J database. In doing so, we wish to address the following research questions:

- 1) Are generated test suites able to detect real faults?
- 2) Which fitness functions have the highest likelihood of fault detection?
- 3) Does an increased search budget improve the effectiveness of the resulting test suites?

These three questions allow us to establish a basic understanding of the effectiveness of each fitness function—are *any* of the functions able to generate fault-detecting tests and, if so, are any of these functions more effective than others at the task? However, these questions presuppose that only one fitness function can be used to generate test suites. Many search-based generation algorithms can simultaneously target *multiple* fitness functions. Therefore, we also ask:

- 4) Are there situations where a combination of criteria could outperform a single criterion?
- 5) Does EvoSuite’s default configuration—a combination of eight criteria—outperform any single criterion?

Finally, across all criteria, we also would like to answer:

- 6) What factors correlate with a high likelihood of fault detection?

We have performed the following experiment:

- 1) **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section III-A).
- 2) **Generated Test Cases:** For each fault, we generated 10 suites per criterion, as well as EvoSuite’s default configuration, using the fixed version of each CUT. We performed with both a two-minute and a ten-minute search budget per CUT (Section III-B).
- 3) **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (Section III-B).
- 4) **Assessed Fault-finding Effectiveness:** For each fault, we measure the proportion of test suites that detect the fault to the number generated (Section III-C).
- 5) **Recorded Generation Statistics:** For each suite, fault, and budget, we measure factors that may influence suite effectiveness, related to coverage, suite size, and obligation satisfaction (Section III-C).

A. Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [25]². Currently, it consists of 357 faults from five projects: JFreeChart (26 faults), Closure compiler (133 faults), Apache Commons Lang (65 faults), Apache Commons Math (106 faults), and JodaTime (27 faults). Four

faults from the Math project were omitted due to complications encountered during suite generation, leaving 353 that we used in our study.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault.

B. Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [38]. In this study, we used EvoSuite version 1.0.3, and the following fitness functions:

Branch Coverage (BC): A test suite satisfies branch coverage if all control-flow branches are taken by at least one test case—the test suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one whose execution evaluates the predicate to `false`. To guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how “close” the targeted predicate is to being true, using a cost function based on the predicate formula [4].

Direct Branch Coverage (DBC): Branch coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. When a test covers a branch indirectly, it can be more difficult to understand how coverage was attained. Direct branch coverage requires each branch to be covered through a direct method call.

Line Coverage (LC): A test suite satisfies line coverage if it executes each non-comment source code line at least once. To cover each line of source code, EvoSuite tries to ensure that each basic code block is reached. The branch distance is computed for each branch that is a control dependency of any of the statements in the CUT. For each conditional statement that is a control dependency for some other line in the code, EvoSuite requires that the branch of the statement leading to the dependent code is executed.

Exception Coverage (EC): The goal of exception coverage is to build test suites that force the CUT to throw exceptions—either declared or undeclared. As the number of possible exceptions that a class can throw cannot be known ahead of time, the fitness function rewards suites that throw more exceptions. As this function is based on the number of

²Available from <http://defects4j.org>

Method	Budget	Total Obligations	% Obligations Satisfied	Suite Size	Suite Length	# Tests Removed	% LC (Fixed)	% LC (Faulty)	% BC (Fixed)	% BC (Faulty)	% Patch Coverage
Default	120	1834.43	50.50%	44.06	360.03	0.25	50.61%	49.82%	41.00%	40.00%	45.51%
	600		57.59%	58.50	563.74	0.47	55.41%	53.85%	47.00%	46.00%	49.94%
Branch Coverage (BC)	120	327.91	54.28%	36.35	225.33	0.32	56.44%	55.78%	48.00%	47.00%	50.31%
	600		61.58%	43.71	305.13	0.46	61.16%	60.30%	54.00%	53.00%	54.04%
Direct Branch (DBC)	120	327.91	49.97%	37.31	244.36	0.26	52.29%	51.80%	44.00%	43.00%	45.96%
	600		57.66%	47.10	347.13	0.43	56.11%	55.11%	49.00%	48.00%	50.02%
Exception Coverage (EC)	120	11.08	99.48%	11.03	28.18	0.07	20.89%	20.91%	11.00%	11.00%	16.41%
	600		99.40%	11.04	28.34	0.07	21.10%	21.08%	11.00%	11.00%	16.92%
Line Coverage (LC)	120	340.56	58.28%	30.64	186.78	0.25	56.91%	56.37%	44.00%	43.00%	50.25%
	600		63.75%	34.15	232.10	0.31	61.27%	60.12%	49.00%	48.00%	53.59%
Method Coverage (MC)	120	31.12	79.56%	21.82	72.33	0.06	36.09%	36.33%	21.00%	21.00%	29.39%
	600		84.48%	24.52	87.06	0.07	37.62%	37.81%	22.00%	22.00%	30.87%
Method, No Exception (MNEC)	120	31.12	77.96%	21.60	71.61	0.07	37.33%	37.41%	22.00%	22.00%	31.18%
	600		83.35%	24.00	89.32	0.07	39.29%	39.32%	23.00%	23.00%	32.89%
Output Coverage (OC)	120	205.71	46.66%	31.02	153.89	0.17	37.55%	37.15%	27.00%	27.00%	34.21%
	600		51.98%	36.86	197.23	0.20	39.85%	39.49%	29.00%	28.00%	36.18%
Weak Mutation Coverage (WMC)	120	560.04	52.57%	28.42	198.53	0.15	51.02%	50.76%	41.00%	41.00%	45.68%
	600		59.30%	35.32	300.84	0.27	56.25%	55.59%	48.00%	47.00%	50.93%

TABLE I

STATISTICS ON GENERATED TEST SUITES. VALUES ARE AVERAGED OVER ALL 353 FAULTS. LC=LINE COVERAGE, BC=BRANCH COVERAGE AS MEASURED BY COBERTURA. PATCH COVERAGE IS LINE COVERAGE OVER THE LINES ALTERED BY THE PATCH THAT FIXES THE FAULT.

discovered exceptions, the number of “test obligations” may change each time EvoSuite is executed on a CUT.

Method Coverage (MC): Method Coverage simply requires that all methods in the CUT are executed at least once, through direct or indirect calls. The fitness function for method coverage is discrete, as a method is either called or not called.

Method Coverage (Top-Level, No Exception) (MNEC): Generated test suites sometimes achieve high levels of method coverage by calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.

Output Coverage (OC): Output coverage rewards diversity in the method output by mapping return types to a list of abstract values [2]. A test suite satisfies output coverage if, for each public method in the CUT, at least one test yields a concrete return value characterized by each abstract value. For numeric data types, distance functions offer feedback using the difference between the chosen value and target abstract values.

Weak Mutation Coverage (WMC): Test effectiveness is often judged using mutants [24]. Suites that detect more mutants may be effective at detecting real faults as well. A test suite satisfies weak mutation coverage if, for each mutated statement, at least one test detects the mutation. The search is guided by the *infection distance*, a variant of branch distance tuned towards reaching and discovering mutated statements [12].

Rojas et al. provide a primer on each of these fitness functions [37]. We have also used EvoSuite’s default configuration—a combination of all of the above methods—to generate test suites. Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated from the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario, where tests guard against future issues.

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits each fitness function. To control experiment cost, we deactivated assertion filtering—all possible

regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget. This resulted in the generation of 63,540 test suites (two budgets, ten trials, nine configurations, 353 faults).

Generation tools may generate flaky (unstable) tests [38]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than one percent of the tests are removed from each suite (see Table I).

C. Data Collection

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **effectiveness** of each fitness function, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault.

To better understand the factors that influence effectiveness, we collected the following for each test suite:

Number of Test Obligations: Given a CUT, each fitness function will calculate a series of test obligations—goals—to cover. The number of obligations is informative of the difficulty of the generation, and impacts the size and formulation of tests [15].

Percentage of Obligations Satisfied: This factor indicates the ability of a fitness function to cover its goals. A suite that covers 10% of its goals is likely to be less effective than one that achieves 100% coverage.

Test Suite Size: We have recorded the number of tests in each test suite. Larger suites are often thought to be more effective [17], [23]. Even if two suites achieve the same coverage, the larger may be more effective simply because it exercises more combinations of input.

Test Suite Length: Each test consists of one or more method calls. Even if two suites have the same number of tests, one

	Budget	Chart	Closure	Lang	Math	Time	Total
Default	120	15	17	29	48	14	123
	600	17	20	35	57	14	143
	Total	18	22	36	59	16	151
BC	120	17	16	36	53	16	138
	600	20	19	35	54	17	145
	Total	21	21	41	57	18	158
DBC	120	14	16	32	48	15	125
	600	19	19	36	47	18	139
	Total	19	22	40	52	18	151
EC	120	8	7	12	13	6	46
	600	10	5	13	12	5	45
	Total	10	8	15	15	6	54
LC	120	15	12	31	50	15	123
	600	18	14	32	52	14	130
	Total	18	17	37	55	15	142
MC	120	10	6	9	25	5	55
	600	10	10	11	24	5	45
	Total	12	10	14	27	6	69
MNEC	120	9	8	10	29	5	61
	600	11	6	13	27	3	60
	Total	11	9	13	32	6	71
OC	120	9	7	13	36	5	70
	600	13	9	17	33	6	78
	Total	13	12	18	38	9	89
WMC	120	13	15	31	42	14	115
	600	18	19	32	48	14	131
	Total	18	22	37	51	16	143
Any criterion	120	18	23	44	61	16	162
	600	23	31	34	63	18	180
	Total	23	32	46	64	18	183

TABLE II

NUMBER OF FAULTS DETECTED BY EACH FITNESS FUNCTION. TOTALS ARE OUT OF 26 FAULTS (CHART), 133 (CLOSURE), 65 (LANG), 102 (MATH), 27 (TIME), AND 353 (OVERALL).

may have much *longer* tests—making more method calls. In assessing the effect of suite size, we must also consider the length of each test case.

Number of Tests Removed: Any tests that do not compile, or that return inconsistent results, are automatically removed. We track the number removed from each suite.

Code Coverage: As the premise of many adequacy criteria is that faults are more likely to be detected if structural elements of the code are thoroughly executed, the resulting coverage of the code may indicate the effectiveness of a test suite. Using the Cobertura tool³, we have measured the line and branch coverage achieved by each suite over both the faulty and fixed versions of each CUT.

Patch Coverage: A high level of coverage does not necessarily indicate that the lines relevant to the fault are covered. We also record line coverage over the program statements modified by the patch that fixes the fault—the lines of code that differ between the faulty and fixed version.

Table I records, for each fitness function and budget, the average values attained for each of these measurements over all faults. Note that the number of test obligations is dependent on the CUT, and does not differ between budgets.

IV. RESULTS & DISCUSSION

In Table II, we list the number of faults detected by each fitness function, broken down by system and search budget. We also list the number of faults detected by *any criterion*. Due to the stochastic search, a higher budget does not guarantee detection of the same faults found under a lower search budget.

³ Available from <http://cobertura.github.io/cobertura/>

Function	Number of Faults
Default	3
BC	6
DBC	3
EC	2
LC	0
MC	1
MNEC	1
OC	1
WC	3

TABLE III

NUMBER OF FAULTS UNIQUELY DETECTED BY EACH FITNESS FUNCTION.

Therefore, we also list the total number of faults detected by each fitness function.

The criteria are capable of detecting 183 (51.84%) of the 353 studied faults.

While there is clearly room for improvement, these results are encouraging. Generated tests are able to detect a variety of complex, real-world faults. In the following subsections, we will assess the results of our study with respect to each research question. In Section IV-A, we compare the capabilities of each fitness function. In Section IV-B, we explore combinations of criteria. Finally, in Section IV-C, we explore the factors that indicate effectiveness.

A. Comparing Fitness Functions

From Table II, we can see that suites differ in effectiveness between criteria. Overall, branch coverage produces the best suites, detecting 158 faults. Branch is closely followed by direct branch (151 faults), weak mutation (143), and line coverage (142). These four fitness functions are trailed by the other four, with exception coverage showing the weakest results (54 faults). These rankings do not differ much on a per-system basis. At times, ranks may shift—for example, direct branch coverage occasionally outperforms branch coverage—but we can see two clusters among the fitness functions. The first cluster contains branch, direct branch, line, and weak mutation coverage—with branch and direct branch leading the other two. The second cluster contains exception, method, method (no-exception), and output coverage—with output coverage producing the best results and exception coverage producing the worst.

Table III depicts the number of faults uniquely detected by each fitness function. A total of twenty faults can only be detected by a single criterion. Branch coverage is again the most effective in this regard, uniquely detecting six faults. Direct branch coverage and weak mutation also perform well, each detecting three faults that no other criterion can detect.

Due to the stochastic nature of the search, one suite generated by EvoSuite may not always detect a fault detected by another suite—even if the same criterion is used. To more clearly understand the effectiveness of each fitness function, we must not track only whether a fault was detected, but the likelihood of detection. To do so, we record the proportion of detecting suites to the total number of suites generated for

	Budget	Chart	Closure	Lang	Math	Time	Total
<i>Default</i>	120	47.31%	4.51%	23.85%	25.78%	25.93%	19.01%
	600	48.08%	7.07%	32.66%	32.84%	33.33%	24.25%
	% Change	1.62%	56.67%	36.77%	27.38%	28.57%	27.57%
BC	120	45.00%	4.66%	34.00%	27.94%	34.82%	22.07%
	600	48.46%	5.79%	40.15%	32.75%	39.26%	25.61%
	% Change	7.69%	24.19%	18.10%	17.19%	12.77%	16.05%
DBC	120	34.23%	5.11%	30.00%	24.51%	31.11%	19.43%
	600	40.77%	6.09%	38.77%	28.63%	40.37%	23.80%
	% Change	19.10%	19.12%	29.23%	16.80%	29.76%	22.45%
EC	120	22.31%	1.35%	7.54%	6.37%	9.26%	6.09%
	600	21.54%	0.98%	9.23%	7.06%	9.63%	6.43%
	% Change	-3.45%	-27.78%	22.45%	10.77%	4.00%	5.58%
LC	120	38.85%	4.14%	31.23%	25.78%	30.00%	19.92%
	600	46.15%	4.81%	34.31%	29.22%	36.67%	22.78%
	% Change	18.81%	16.36%	9.85%	13.31%	22.22%	14.37%
MC	120	30.77%	1.58%	7.54%	10.98%	8.15%	8.05%
	600	30.77%	2.26%	7.69%	10.88%	8.15%	8.30%
	% Change	0.00%	42.86%	2.04%	-0.89%	0.00%	3.17%
MNEC	120	23.46%	2.18%	6.62%	12.16%	6.67%	7.79%
	600	30.77%	1.88%	7.54%	12.06%	5.19%	8.24%
	% Change	31.15%	-13.79%	13.95%	-0.81%	-22.22%	5.82%
OC	120	21.15%	2.03%	7.85%	16.57%	9.63%	9.29%
	600	23.85%	2.56%	10.92%	16.76%	12.22%	10.51%
	% Change	12.73%	25.93%	39.22%	1.18%	26.92%	13.11%
WMC	120	38.08%	4.44%	24.15%	23.04%	25.19%	17.51%
	600	46.15%	5.56%	32.15%	27.45%	27.04%	21.42%
	% Change	21.21%	25.42%	33.12%	19.15%	7.35%	22.33%

TABLE IV

AVERAGE LIKELIHOOD OF FAULT DETECTION, BROKEN DOWN BY FITNESS FUNCTION, BUDGET, AND SYSTEM.

that fault. The average likelihood of fault detection is listed for each criterion, by system and budget, in Table IV.

We largely observe the same trends as above. Branch coverage has the highest overall likelihood of fault detection, with 22.07% of suites detecting faults given a two-minute search budget and 25.61% of suites detecting faults given a ten-minute budget. Line and direct branch coverage follow with a 19.92-22.78% and 19.43-23.80% success rate, respectively. While the effectiveness of each criterion varies between system—direct branch outperforms all other criteria for Closure, for example—the two clusters noted above remain intact. Branch, line, direct branch, and weak mutation coverage all perform well, with the edge generally going to branch coverage. On the lower side of the scale, output, method, method (no exception), and exception coverage perform similarly, with a slight edge to output coverage.

Branch coverage is the most effective criterion, detecting 158 faults—six of which were only detected by this criterion. Branch coverage suites have, on average, a 22.07-25.61% likelihood of fault detection.

From Table IV, we can see that almost all criteria benefit from an increased search budget. Direct branch coverage and weak mutation benefit the most, with average improvements of 22.45% and 22.33% in effectiveness. In general, all distance-driven criteria—branch, direct branch, line, weak mutation, and, partially, output coverage—improve given more time. Discrete fitness functions benefit less from the budget increase.

Distance-based functions benefit from increased budget, particularly direct branch and weak mutation.

We can perform statistical analysis to assess our observa-

	Default	BC	DBC	LC	WM
Default	-	0.89	0.55	0.54	0.25
BC	0.11	-	0.13	0.13	0.03
DBC	0.45	0.87	-	0.50	0.22
LC	0.46	0.87	0.50	-	0.22
WC	0.75	0.97	0.78	0.78	-

TABLE V
P-VALUES FOR MANN-WHITNEY-WILCOXON COMPARISONS OF “TOP-CLUSTER” CRITERIA (TWO-MINUTE SEARCH BUDGET).

tions. For each pair of criteria, we formulate hypothesis H and its null hypothesis, H_0 :

- H : Given a fixed search budget, test suites generated using criterion A will have a higher likelihood of fault detection than suites generated using criterion B .
- H_0 : Observations of fault detection likelihood for both criteria are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H_0 without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [40], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. Due to the limited number of faults for the Chart and Time systems, we have analyzed results across the combination of all systems. We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

The tests further validate the “two clusters” observation. For the four criteria in the top cluster—branch, direct branch, line, and weak mutation coverage—we can always reject the null hypothesis with regard to the remaining four criteria in the bottom cluster. Within each cluster, we generally cannot reject the null hypothesis. P-values within the top cluster, when a two-minute search budget is used, are shown in Table V. We can use these results to infer a partial ordering—Branch coverage outperforms weak mutation coverage with significance. While there were no other cases where H_0 can be rejected, p-values are much lower for branch coverage against the other criterion than in any other pairing⁴. This suggests a slight advantage in using branch coverage, consistent with previous results. Similarly, in the lower cluster, we can reject H_0 for output coverage against exception coverage (two-minute budget) and against exception, method, and method (no exception) with a ten-minute budget. We cannot reject H_0 in the other comparisons.

Branch coverage outperforms, with statistical significance, weak mutation (2m budget), and method, MNEC, output, and exception coverage (both budgets).

B. Combinations of Fitness Functions

The analysis above presupposes that only one fitness function can be used to generate test suites. However, many search-based generation algorithms can simultaneously target

⁴P-values for a ten-minute budget are omitted due to space constraints, but suggest similar conclusions.

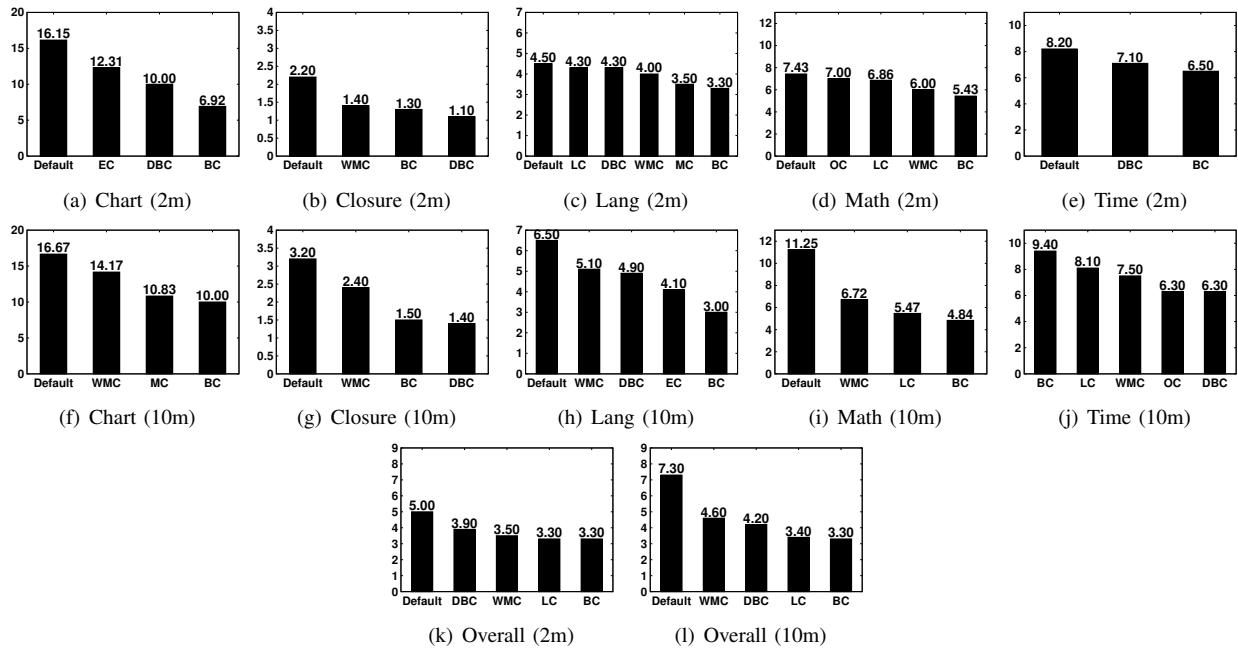


Fig. 1. Average % likelihood of fault detection for fitness functions once data is filtered for faults where the most effective function for that system has < 30% chance of detection.

multiple fitness functions. EvoSuite’s default configuration, in fact, attempts to satisfy all eight of the fitness functions examined in this study [37]. In theory, suites generated through a combination of fitness functions could be more effective than suites generated through any one objective. For example, combining exception and branch coverage may result in a suite that both thoroughly explores the structure of the system (due to the branch obligations) and rewards tests that throw a larger number of exceptions. Such a suite may be more effective than a suite generated using branch or exception coverage alone. To better understand the potential of combined criteria, we answer two questions. First, are there situations where the most effective criterion is outperformed by another, secondary, criterion? Second, does EvoSuite’s default combination outperform the individual criteria?

From Table III, we can see that there are a total of twenty faults only detected by a single configuration. Thus, it is clear that no one fitness function can detect all faults. Almost all criteria—regardless of their overall effectiveness—can detect something the others cannot. This does not automatically mean that combinations of criteria can detect these faults either—the default configuration does not detect the 17 faults detected by the other individual criteria. It does, however, detect three faults not found otherwise. Still, combinations of criteria, either explored over multiple, independent generations—each for a single criterion—or through a single multi-objective generation, may be able to detect faults missed by a single criterion.

To further understand the situations where combinations of criteria may be useful, we filter the set of faults for those that the top-scoring criterion is ineffective at detecting. In Figure 1, we have taken the faults for each system, isolated any where the “best” criterion for that system (generally

branch coverage, see Table IV) has < 30% likelihood of detection, and calculated the likelihood of fault detection for each criterion for that subset of the faults. In each subplot, we display the likelihood of fault detection for any criterion that outperforms the best from the full set.

From these plots, we can see that there are always 2-5 criterion that are more effective in these situations. The exact criteria depend strongly on the system, and likely, on the types of faults examined. Interestingly, despite the similarity in distance functions and testing intent, direct branch and line coverage are often more effective than branch coverage in situations where it has a low chance of detection. In these cases, the criteria drive tests to interact in such a way with the CUT that they are better able to detect the fault. Despite its poor overall performance, exception coverage also often does well in these situations—demonstrating the wisdom of favoring suites that throw more exceptions.

These results imply that a combination of criteria could outperform a single criterion. Indeed, from Figure 1, we can see that EvoSuite’s default configuration outperforms all other criteria for the examined subsets of faults. In situations where the single “best” criterion performs poorly, a multi-objective solution achieves improved results.

In situations where the criterion that is the most effective overall has < 30% likelihood of fault detection, other criteria and combinations of criteria more effectively detect the fault. The effective secondary criteria vary by system.

Overall, EvoSuite’s default configuration performs well, but fails to outperform all individual criteria. Table II shows that

the default configuration detects 151 faults—fewer than branch coverage, but a tie with direct branch coverage. As previously mentioned, it also uniquely detects three faults (see Table III). Again, this is fewer than branch coverage, but tied with direct branch and weak mutation coverage. According to Table IV, the default configuration’s average overall likelihood of fault detection is 19.01% (2m budget)-24.25% (10m budget). At the two-minute level, this places it below branch, line, and direct branch coverage. At the ten-minute level, it falls below branch coverage, but above all other criteria. This places the default configuration in the top cluster—an observation confirmed by further statistical tests.

In theory, a combination of criteria could detect more faults than any single criterion. In practice, combining *all* criteria results in suites that are quite effective, but fail to reliably outperform individual criterion. From Table IV, we can see that the performance of the default configuration also varies quite a bit between systems, and by search budget. For Chart and Math, the default configuration performs almost as well as branch coverage. With a higher budget, it performs as well or better than Branch for Math and Closure. However, for the Lang and Time systems, the default configuration is less effective than branch, line, and direct branch coverage even with a larger budget.

The major reason for the less reliable performance of this configuration is the difficulty in attempting to satisfy so many obligations at once. As noted in Table I, the default configuration must attempt to satisfy, on average, 1,834 obligations. The individual criteria only need to satisfy a fraction of that total. As a result, the default configuration also benefits more than any individual criterion from an increased search budget—a 27.57% improvement in efficacy.

EvoSuite’s default configuration has an average 19.01-24.25% likelihood of fault detection—in the top cluster, but failing to outperform all individual criteria.

Our observations imply that combinations of criteria could be more effective than individual criteria. However, a combination of all eight criteria results in unstable performance—especially if search budget is limited. Instead, testers may wish to identify a smaller subset of criteria to combine during test generation. More research is needed to understand which combinations are most effective, and whether system-specific combinations may yield further improvements.

C. Understanding the Factors Leading to Fault Detection

As discussed in Section III-C—to better understand the combination of factors correlating with effective fault detection—we have collected the following statistics for each generated test suite: the number of obligations, the percent satisfied, suite size, suite length, number of tests removed, branch (BC) and line coverage (LC) over the fixed and faulty versions of the CUT, and coverage of the lines changed to fix the fault (patch coverage, or PC).

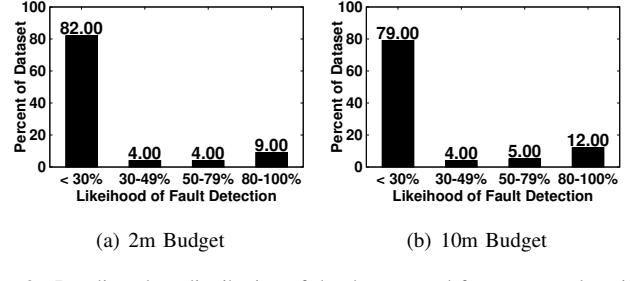


Fig. 2. Baseline class distribution of the dataset used for treatment learning. Likelihood of fault detection is discretized into four classes—our target is the 80-100% class.

This collection of factors forms a dataset in which, for each fault, we have recorded the average of each statistic for the suites generated to detect that fault. We can then use the likelihood of fault detection (D) as the class variable—discretized into four values: $D < 30\%$, $30 \leq D < 50$, $50 \leq D < 80$, $D \geq 80$. We have created two datasets, dividing data by search budget. The class distribution of each dataset is shown in Figure 2.

A standard practice in machine learning is to *classify data*—to use previous experience to categorize new observations [31]. We are instead interested in the reverse scenario. Rather than attempting to categorize new data, we want to work backwards from classifications to discover *which factors correspond most strongly to a class of interest*—a process known as treatment learning [29]. Treatment learning approaches take the classification of an observation and reverse engineer the evidence that led to that categorization. Such learners produce a *treatment*—a small set of attributes and value ranges that, if imposed, will identify a subset of the original data matching skewed towards the target classification. In this case, a treatment notes the factors—and their values—that contribute to a high likelihood of fault detection. Using the TAR3 treatment learner [14], we have generated five treatments from each dataset. The treatments are scored according to their impact on class distribution, and top-scoring treatments are presented first. We extracted the following treatments:

Two-Minute Dataset:

- 1) BC (fixed) > 67.70%, PC > 66.67%
- 2) LC (fixed) > 82.19%, PC > 66.67%, BC (fixed) > 67.70%
- 3) PC > 66.67%, LC (fixed) > 82.19%
- 4) 69.40 ≥ Length ≤ 169.80, BC (fixed) > 67.70%
- 5) BC (fixed) > 67.70%, % of obligations satisfied > 86.38%

Ten-Minute Dataset:

- 1) BC (fixed) > 76.08%, PC > 70.00%
- 2) PC > 70.00%, LC (fixed) > 85.92%
- 3) BC (fixed) > 76.08%, % of obligations satisfied > 93.30%
- 4) BC (fixed) > 76.08%, LC (fixed) > 85.92%, % of obligations satisfied > 93.30%
- 5) PC > 70.00%, LC (fixed) > 85.92%, BC (fixed) > 76.08%

In Figure 3(a), we plot the class distribution of the subset fitting the highest-ranked treatment learned from the two-minute dataset. In Figure 4, we do the same for the top treatment from the ten-minute dataset. Comparing the plots in Figure 2 to the subsets in Figures 3-4, we can see that the treatments

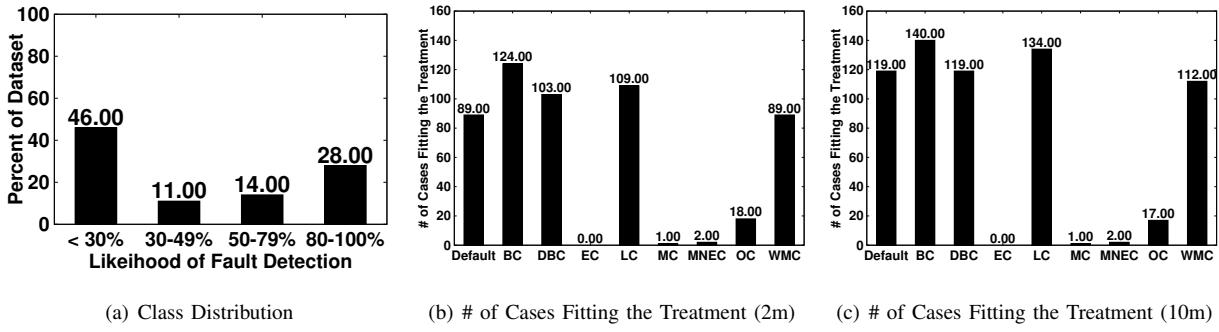


Fig. 3. For the top treatment learned from the 2m dataset: Class distribution of the data subset fitting the treatment, the number of cases that fit the treatment for each fitness function from the 2m dataset, and the number of cases that fit the treatment for each fitness function from the 10m dataset.

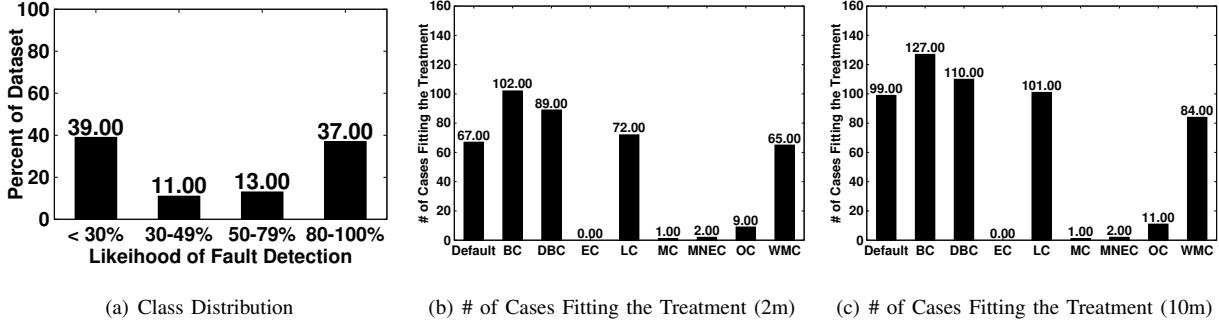


Fig. 4. For the top treatment learned from the 10m dataset: Class distribution of the data subset fitting the treatment, the number of cases that fit the treatment for each fitness function from the 2m dataset, and the number of cases that fit the treatment for each fitness function from the 10m dataset.

do impose a large change in the class distribution—a lower percentage of cases have the < 30% classification, and more have the other classifications, particularly 80 – 100%. This shows that the treatments do reasonably well in predicting for success. Test suites fitting these treatments are not guaranteed to be successful, but are more likely to be.

We can make several observations. First, the most common factors selected as indicative of efficacy are all coverage-related factors. For both datasets, the top-ranked treatment specifically indicates that branch coverage over the fixed version of the CUT and patch coverage are the most important factors in determining suite efficacy. Even if their goal is not to attain coverage, successful suites thoroughly explore the structure of the CUT. The fact that coverage is important is not, in itself, entirely surprising—if patched code is not well covered, the fault is unlikely to be discovered.

More surprising is how much weight is given to coverage. Coverage is recommended by all of the selected treatments—generally over the fixed version of the CUT. While patch coverage is important, overall branch and line coverage over the faulty version is less important than coverage over the fixed version. It seems to be important that the suite thoroughly explores the program it is generated on, and that it still covers lines patched in the faulty version.

Suite size has been a focus in recent work, with Inozemtseva et al. (and others) finding that the size has a stronger correlation to efficacy than coverage level [23]. However, a size attribute (suite length) only appears in one of the treatments produced for either dataset. This seems to indicate

that, unlike with mutants, larger test suites are not necessarily more effective at detecting real faults.

The other factor noted as indicative of efficacy is the percent of obligations satisfied. For coverage-based fitness functions like branch and line coverage, a high level of satisfied obligations likely correlates with a high level of branch or line coverage. For other fitness functions, the correlation may not be as strong, but it is likely that suites satisfying more of their obligations also explore more of the structure of the CUT.

Factors that strongly indicate efficacy include a high level of branch coverage over the fixed CUT and patch coverage. Coverage is favored over factors related to suite size or test obligations.

We have recorded how effective each fitness function is at producing suites that meet the conditions indicated by the top-ranked treatments learned from each dataset in Figures 3(b) and (c) and 4(b) and (c). From these plots, we can immediately see that the top cluster of fitness functions met those conditions for a large number of faults. The bottom cluster, on the other hand, rarely meets those conditions.

Note, however, that we still do not entirely understand the factors that indicate a high probability of fault detection. From Figures 3-4, we can see that the treatments radically alter the class distribution from the baseline in Figure 2. Still, from Figure 4, we can see that suites fitting the top-ranked treatments are ineffective as often as they are highly effective. From this, we can conclude that factors predicted by

treatments are a necessary precondition for a high likelihood of fault detection, but are not sufficient to ensure that faults are detected. Unless code is executed, faults are unlikely to be found. However, how code is executed matters, and execution alone does not guarantee that faults are triggered and observed as failures. The fitness function determines how code is executed. It may be that fitness functions based on stronger adequacy criteria (such as complex condition-based criteria [39]) or combinations of fitness functions will better guide such a search. Further research is needed to better understand how to ensure a high probability of fault detection.

While coverage increases the likelihood of fault detection, it does not ensure that suites are effective.

V. RELATED WORK

Those who advocate the use of adequacy criteria hypothesize that criteria fulfillment will result in test suites more likely to detect faults—at least with regard to the structures targeted by that criterion. If this is the case, we should see a correlation between higher attainment of a criterion and the chance of fault detection for a test suite [20]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [33], [30], [6], [9], [10], [8], [18], [23], [16]. Inozemtseva et al. provide a good overview of work in this area [23]. Our focus differs—our goal is to examine the relationship between fitness function and fault detection efficacy for search-based test generation. However, fitness functions are largely based on, and intended to fulfill, adequacy criteria. Therefore, there is a close relationship between the fitness functions that guide test generation and adequacy criteria intended to judge the resulting test suites.

EvoSuite has previously been used to generate test suites for the systems in the Defects4J database. Shamshiri et al. applied EvoSuite, Randoop, and Agitar to each fault in the Defects4J database to assess the general fault-detection capabilities of automated test generation [38]. They found that the combination of all three tools could identify 55.7% of the studied faults. Their work identifies several reasons why faults were not detected, including low levels of coverage, heavy use of private methods and variables, and issues with simulation of the execution environment. In their work, they only used the branch coverage fitness function when using EvoSuite. In our study, we have expanded the number of EvoSuite configurations to better understand the role of the fitness function in determining suite efficacy.

VI. THREATS TO VALIDITY

External Validity: Our study has focused on a relatively small number of systems. Nevertheless, we believe that such systems are representative of—at minimum—other small to medium-sized open-source Java systems. We also believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar projects.

We have used a single test generation framework. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same number and variety of fitness functions. Therefore, a more thorough comparison of tool performance cannot be made at this time.

To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, this still yielded 64,360 test suites to use in analysis. We believe that this is a sufficient number to draw stable conclusions.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.

VII. CONCLUSIONS

We have examined the role of the fitness function in determining the ability of search-based test generators to produce suites that detect complex, real faults. From the eight fitness functions and 353 faults studied, we can conclude:

- EvoSuite’s branch coverage fitness function is the most effective—detecting more faults than any other criterion, and having a higher likelihood of detection for each fault. Line, direct branch, and weak mutation coverage also perform well in both regards.
- There is evidence that multi-objective suite generation could be more effective than single-objective generation, and EvoSuite’s default combination of eight functions performs relatively well. However, the difficulty of simultaneously balancing all eight functions decreases the average performance of this configuration, and it fails to outperform branch coverage.
- High levels of coverage over the fixed version of the CUT and over patched lines of code are the factors that most strongly predict suite effectiveness.
- While others have found that test suite size correlates to mutant detection, we found that larger suites are not necessarily more effective at detecting real faults.

Our findings represent a step towards understanding the use, applicability, and combination of common fitness functions. Our observations indicate that, while coverage seems to be a prerequisite to effective fault detection, it is not sufficient to ensure it. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. We hypothesize that lightweight, perhaps system-specific, combinations of fitness functions may be more effective than single metrics in improving the probability of fault detection—both exploring the structure of the CUT and sufficiently varying input in a manner that improves the probability of fault detection. However, more research is needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation

algorithm, and CUT in determining the efficacy of test suites. In future work, we plan to further explore these topics.

REFERENCES

- [1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010.
- [2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, New York, NY, USA, 2014. ACM.
- [3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
- [4] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.
- [5] L. Bianchi, M. Dorigo, L. Gambardella, and W. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.
- [6] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST ’05, pages 1–7, New York, NY, USA, 2005. ACM.
- [7] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.
- [8] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT ’98/FSE-6, pages 153–162, New York, NY, USA, 1998. ACM.
- [9] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 154–164, New York, NY, USA, 1991. ACM.
- [10] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, Aug 1993.
- [11] G. Fraser and A. Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, Feb 2013.
- [12] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.
- [13] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.
- [14] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Automated Software Engineering*, 17(4):439–468, Dec. 2010.
- [15] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Trans. Softw. Eng. Methodol.*, 25(3):25:1–25:34, July 2016.
- [16] G. Gay, M. Staats, M. Whalen, and M. Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, PP(99), 2015.
- [17] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.
- [18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.
- [19] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014.
- [20] A. Groce, M. A. Alipour, and R. Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 255–268, New York, NY, USA, 2014. ACM.
- [21] M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
- [22] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.
- [23] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.
- [24] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.
- [25] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.
- [26] E. Kit and S. Finzi. *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
- [27] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE ’11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.
- [28] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.
- [29] T. Menzies and Y. Hu. Data mining for very busy people. *Computer*, 36(11):22–29, Nov. 2003.
- [30] A. Mockus, N. Nagappan, and T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301, Oct 2009.
- [31] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2012.
- [32] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
- [33] A. Namin and J. Andrews. The influence of size and coverage on test suite effectiveness, 2009.
- [34] W. Perry. *Effective Methods for Software Testing, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2006.
- [35] M. Pezzé and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.
- [36] S. Rayadurgam and M. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int’l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.
- [37] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.
- [38] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.
- [39] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 Int’l Conf. on Software Engineering*. ACM, May 2013.
- [40] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.

Using Search-Based Test Generation to Discover Real Faults in Guava

Hussein Almulla, Alireza Salahirad, Gregory Gay

University of South Carolina, Columbia, SC, USA,^{**}
[halmulla, alireza]@email.sc.edu, greg@greggay.com

Abstract. Testing costs can be reduced through automated unit test generation. An important benchmark for such tools is their ability to detect *real faults*. Fault databases, such as Defects4J, assist in this task. The Guava project—a collection of Java libraries from Google—offers an opportunity to expand such databases with additional complex faults. We have identified 11 faults in the Guava project, added them to Defects4J, and assessed the ability of the EvoSuite framework to detect these faults. Ultimately, EvoSuite was able to detect three faults. Analysis of the remaining faults offers lessons in how to improve generation tools. We offer these faults to the community to assist future benchmarking efforts.

Keywords: Search-based test generation, automated test generation, software faults

1 Introduction

With the growing complexity of software, the cost of testing has grown as well. Automation of tasks such as unit test creation can assist in controlling that cost. One promising form of automated test generation is *search-based* generation. Given a measurable testing goal, and a fitness function capable of guiding the search towards that goal, powerful optimization algorithms can select test inputs able to meet that goal [3].

When testing, developers ultimately wish to detect faults. Therefore, to impact testing practice, automated generation techniques must be effective at detecting the complex faults that manifest in real-world software projects [7]. By offering examples of such faults, fault databases—such as Defects4J [6]—allow us to benchmark generation tools against realistic case examples. Importantly, Defects4J can be expanded to include additional systems and example faults.

The Guava project ¹ offers an excellent expansion opportunity. Guava is an open-source set of core libraries for Java, developed by Google, that include collection types, graph libraries, functional types, in-memory caching, and numerous other utilities. Guava is an essential tool of modern development, and is one of the most used libraries [8].

Guava serves as an interesting benchmark subject for two reasons. First, much of its functionality is, naturally, related to the creation and manipulation of complex objects. Guava defines a variety of new data structures, and functionality related to those structures. Generation and initialization of complex input is an outstanding challenge area

^{**} This work is supported by National Science Foundation grant CCF-1657299.

¹ <https://github.com/google/guava>

for automated generation [1]. Second, Guava is a mature project. Faults in Guava—particularly recent faults—are unlikely to resemble the simple syntactic mistakes modeled by mutations. Rather, we expect to see faults that require specific, difficult to trigger, combinations of input and method calls. Generation tools that can detect such faults are likely to be effective on other real-world projects. If not, then by studying these faults, we may be able to learn lessons that will improve these tools.

We have identified 11 real faults in the Guava project, and added them to Defects4J. We generated test suites using the EvoSuite framework [3], and assessed the ability of these suites to detect nine of the faults². Ultimately, EvoSuite is able to detect three of the nine studied faults. Some of the issues preventing fault detection include the need for specific input values, data types, or sequences of method calls—generally factors that cannot be addressed through code coverage alone. We have made these faults available to provide data and examples that could benefit future test generation research.

2 Study

In this study, we have extracted faults from the Guava project. We have generated tests for the fixed version of each class using the EvoSuite framework [3], and applied those tests to the faulty version in order to assess the efficacy of generated suites. In doing so, we wish to answer the following research questions: (1) *can EvoSuite detect the extracted faults?*, and (2), *what factors prevented fault detection?*

In order to answer these questions, we have performed the following experiment:

1. **Extracted Faults:** We have identified 11 real faults in the Guava project, and added them to the Defects4J fault database (See Section 2.1).
2. **Generated Test Cases:** For nine of the faults, we generated 10 suites per fault using the fixed version of each class-under-test (CUT). We repeat this process with a two-minute and a ten-minute search budget per CUT (See Section 2.2).
3. **Removed Non-Compiling Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (See Section 2.2).
4. **Assessed Fault-finding Efficacy:** For each budget and fault, we measure the likelihood of fault detection. For each undetected fault, we examined the report and source code to identify possible detection-preventing factors.

2.1 Fault Extraction

Defects4J is an extensible database of real faults extracted from Java projects [6]³. Currently, it consists of 395 faults from six projects. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose each fault, and a list of classes and lines of code modified to fix the fault.

We have added Guava to Defects4J. This consisted of developing build files that work across project versions, extracting candidate faults using Guava’s version control and issue tracking systems, ensuring that each candidate could be reliably reproduced, and minimizing the “patch” used to distinguish fixed and faulty classes.

² Two faults were omitted from the case study as they require the use of JDK 7 (see Section 2).

³ Available from <http://defects4j.org>

For inclusion in the final dataset, each fault is required to meet three properties. First, the fault must be related to the source code. For each reported issue, we attempted to identify a pair of code versions that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix to the fault must be isolated from unrelated code changes such as refactorings.

One property of all Defects4J faults is that the commit message for the “fixed” version reference a reported issue in the project’s tracking system (i.e., “fixes #2345”). Fulfilling this property has not been a problem for the six existing projects, as the developers of those projects have used a standard commit message format. However, Guava commits do not follow a standard format—many “fixes” do not reference a reported issue. To maintain continuity with the other Defects4J projects, we restricted our search to fixes that do make an explicit reference. Following this process, we extracted 11 faults from a pool of 63 candidate faults that reference an explicit issue. In the future, we may allow commits without explicit references in order to mine additional faults.

One additional limiting factor is that particular Java Development Kit versions must be installed and used to build certain versions of Guava. Due to language changes, JDK 7 must be used to build faults 10 and 11. Faults 1-9 can be built using JDK 8. Recently, the decision was made to require that all new additions to Defects4J be compatible with JDK 8. Faults 10 and 11 will still be made available, but will not be included in the core Defects4J database or used in our case study.

The faults used in this study can be accessed by cloning the `bug-mining` branch of <https://github.com/Greg4cr/defects4j>. Additional data about each fault can be found at <http://greggay.com/data/guava/guavafaults.csv>, including commit IDs, fault descriptions, and a list of triggering tests. Later, these faults will be migrated into the `master` branch at <http://defects4j.org>. We plan to add additional faults and improvements in the future.

2.2 Test Generation and Removal

EvoSuite applies a genetic algorithm in order to evolve test suites over several generations, forming a new population by retaining, mutating, and combining the strongest solutions [7]. In this study, we used EvoSuite version 1.0.5 with a combination of three fitness functions—Branch, Exception, and Method Coverage—a combination recently found to be generally effective at detecting faults [5].

Tests are generated from the fixed version of the system and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario. Given the potential difficulty in achieving coverage over Guava classes, two search budgets were used—two and ten minutes, a typical and an extended budget [4]. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget.

Generation tools may generate flaky (unstable) tests [7]. For example, a test case that makes assertions about the system time will only pass during generation. We auto-

Fault	Budget	Fault Detected	Likelihood of Detection	Branch Coverage (Covered/Total Goals)	Suite Size	Suite Length	Number of Tests Removed
1	2 min	X	0.00%	11.83% (22.60/191.00)	6.90	26.10	0.90
	10 min	X	0.00%	73.25% (139.90/191.00)	38.70	180.80	11.10
2	2 min	X	0.00%	92.47% (82.30/89.00)	26.10	65.60	0.00
	10 min	X	0.00%	93.60% (83.30/89.00)	26.90	70.00	0.00
3	2 min	✓	90.00%	96.64% (132.40/137.00)	65.90	114.90	0.00
	10 min	✓	90.00%	97.52% (133.60/137.00)	67.70	117.00	0.00
4	2 min	✓	60.00%	67.56% (83.10/123.00)	91.40	270.40	1.20
	10 min	✓	100.00%	92.03% (113.20/123.00)	130.40	424.20	1.60
5	2 min	X	0.00%	32.38% (13.60/42.00)	4.70	49.40	0.00
	10 min	X	0.00%	76.31% (30.70/40.20)	14.50	96.90	0.00
6	2 min	X	0.00%	3.10% (30.90/1008.00)	3.00	11.60	0.00
	10 min	X	0.00%	3.10% (31.10/1008.00)	3.40	13.00	0.10
7	2 min	X	0.00%	2.32% (23.30/1005.00)	3.40	20.50	0.00
	10 min	X	0.00%	1.91% (19.20/1005.00)	2.20	15.50	0.30
8	2 min	✓	10.00%	21.51% (11.40/53.00)	7.30	35.10	0.00
	10 min	✓	60.00%	91.89% (48.70/53.00)	42.40	214.90	1.20
9	2 min	X	0.00%	3.54% (5.60/158.00)	3.40	8.40	0.00
	10 min	X	0.00%	57.47% (90.80/158.00)	74.30	175.30	0.20

Table 1. Test generation results for each fault and search budget—likelihood of fault detection, average achieved branch coverage (covered/total branches), average number of tests, average suite length, and average number of tests removed.

matically remove flaky tests⁴. First, non-compiling test cases are removed. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, 0.92 tests are removed from each suite.

3 Results and Discussion

In Table 1, we list—for each search budget—whether EvoSuite was able to detect each fault and, if so, the likelihood of detection (the proportion of suites that detected the fault). We also list the average Branch Coverage attained over the ten trials, the average number of tests in the generated suites, the average suite length (number of test steps), and the average number of tests removed.

From Table 1, we can see the three of the nine faults were detected (Faults 3, 4, and 8). Of these, Fault 3 was detected the most reliably (90% likelihood for both budgets). This fault, dealing with incorrect rounding⁵, is a classic example of the types of faults that automated generation excels at. Branch Coverage drives the search towards the affected code and towards differing output between versions.

Fault 4⁶ was also detected reliably. The faulty version uses a non-standard ASCII character in the `toString()` function for class `Range`. This is a relatively easy fault to catch—any call to `toString()` with a valid `Range` object will result in differing output between faulty and fixed versions. With the shorter search budget, EvoSuite is somewhat less likely to call the function and somewhat more likely to set up an invalid range. However, the longer budget ensures that the fault is caught by all suites.

Fault 8 involves the computation of the intersection of `RegularContiguousSet` objects when one is a singleton⁷. One of the changes made to fix the fault is a shift from

⁴ This process is documented in more detail in [7] and [4].

⁵ <https://github.com/google/guava/commit/1b1163b7e2c121d4a5b25b8966714201551976c4>

⁶ <https://github.com/google/guava/commit/c6e21a35f3113a7a952a9615a0e92dcf1dd4bfb3>

⁷ <https://github.com/google/guava/commit/44a2592b04490ad26d2bc874f9dbd4c1146cc5de>

< in the return statement of the `isEmpty()` method to `<=`. Any test case where the two compared variables are the same will now detect the fault. A longer search budget increases the number of suites that detect the fault (10% to 30%), but this is still a clear case of a fault that requires not just coverage, but *picking specific input*.

EvoSuite failed to detect the other six faults. Therefore, our next step was to examine these faults to identify factors preventing detection. These factors include:

Specific Input Values are Required: As seen in Fault 8, it is not enough to simply cover a line. At times, specific input values are required to trigger and detect a fault. In the case of Fault 8, the generator is able to stumble on these inputs given enough time. In other cases, such as with Fault 2⁸, not enough context is offered to the generator. Because of this fault, splitting a string with a zero-width regular expression pattern would result in single-character strings on either end of the split being dropped. The fix to the code changes a `>=` to a `>`, but unless input matching this particular corner case is used, the fault will not be discovered.

Specific Data Types are Required for Input: Guava includes functionality for iterating over lists that is intended to function regardless of the *type* of list used. Fault 9⁹ illustrates the difficulty of verifying such functionality. Unlike sets, lists typically allow duplicate elements. This is not universally true, however. Therefore, if a list type is used that does not allow duplicates, then the affected code in Guava will throw an exception. This is another case that coverage cannot handle, as coverage can be obtained using any type of list. Detecting the fault requires choosing a specialized data type.

Inputs are Instances of Complex Data Types: Generating input for complex data types is still an open challenge for automated generation [1]. If the generator cannot produce and manipulate input of such types, it may not be able to cover code, reducing the possibility of triggering faults. Fault 6¹⁰ is one such example. This fault revolves around the wrong cause of removal being listed for items in a cache. To discover this fault, EvoSuite must generate and initialize an instance of the class `LocalCache`. In addition, this class is a generic type, further complicating automated generation [2].

A Specific Series of Method Calls Must Be Generated: Each unit test consists of a series of one or more calls to methods in the CUT. Rather than specific input, at times, triggering a fault requires a specific sequence of calls. Fault 5¹¹ is one such example. In this case, a long sequence of nested `Futures.transform(...)` calls on the same object will indefinitely hang because a `StackOverflowException` is thrown and swallowed. Detecting this fault requires not only input that triggers an exception, but a sequence of transformation calls on that input.

Fault 1¹² offers a second example of this factor. `MinMaxPriorityQueue` fails to remove the correct object after a sequence of multiple `add` and `remove` calls—specifically, certain elements may be iterated more than once if elements are removed during iteration. It would not be unusual to see a sequence of calls in a generated test case. However, the example tests created by humans to reproduce this fault include

⁸ <https://github.com/google/guava/commit/55524c66de8db4c2e44727b69421c7d0e4f30be0>

⁹ <https://github.com/google/guava/commit/1a1b97ee1f065d0bc52c91eeeb6407bfaa6cbea1>

¹⁰ <https://github.com/google/guava/commit/0a686a644ca5cefb9e7bf4a38b34bf4ede9e75aa>

¹¹ <https://github.com/google/guava/commit/52b5ee640da780e0fd2502ec995436fcfdc93e03e>

¹² <https://github.com/google/guava/commit/2ef955163b3d43e7849c1929ef4e5d714b93da96>

relatively long sequences of calls. The suite minimization and bloat control mechanisms used to control suite size in automated generation are designed to avoid a long series of calls that do not contribute to code coverage—actively discouraging the generation of the very type of test cases that would detect this fault.

Many of these factors cannot be solved through increasing code coverage. Rather, they require context from the project. Methods of gleaned that context, either through seeding from existing test cases or data mining of project elements, may assist in improving the efficacy of test generation.

4 Conclusion

We have identified 11 real faults in the Guava project, and added them to the Defects4J fault database. To study the capabilities of modern test generation tools, we generated test suites using the EvoSuite framework. Ultimately, EvoSuite is able to detect three of the nine studied faults. Some of the issues preventing fault detection include the need for specific input values, data types, or sequences of method calls—generally factors that cannot be addressed through code coverage alone. We have made these faults available to provide data and examples that could benefit future test generation research.

References

1. Feldt, R., Pouling, S.: Finding test data with specific properties via metaheuristic search. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 350–359 (Nov 2013)
2. Fraser, G., Arcuri, A.: Automated Test Generation for Java Generics, pp. 185–198. Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-03602-1_12
3. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 291–301. ISSTA, ACM, New York, NY, USA (2013), <http://doi.acm.org/10.1145/2483760.2483774>
4. Gay, G.: The fitness function for the job: Search-based generation of test suites that detect real faults. In: Proceedings of the International Conference on Software Testing. ICST 2017, IEEE (2017)
5. Gay, G.: Generating effective test suites by combining coverage criteria. In: Proceedings of the Symposium on Search-Based Software Engineering. SSBSE 2017, Springer Verlag (2017)
6. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), <http://doi.acm.org/10.1145/2610384.2628055>
7. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). ASE 2015, ACM, New York, NY, USA (2015)
8. Weiss, T.: We analyzed 30,000 GitHub projects - here are the top 100 libraries in Java, JS and Ruby (2013), <http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-javascript-and-ruby/>

Generating Effective Test Suites by Combining Coverage Criteria

Gregory Gay

University of South Carolina, Columbia, SC, USA,
greg@greggay.com

Abstract. A number of criteria have been proposed to judge test suite adequacy. While search-based test generation has improved greatly at criteria coverage, the produced suites are still often ineffective at detecting faults. Efficacy may be limited by the single-minded application of one criterion at a time when generating suites—a sharp contrast to human testers, who simultaneously explore multiple testing strategies. We hypothesize that automated generation can be improved by selecting and simultaneously exploring multiple criteria.

To address this hypothesis, we have generated multi-criteria test suites, measuring efficacy against the Defects4J fault database. We have found that multi-criteria suites can be up to 31.15% more effective at detecting complex, real-world faults than suites generated to satisfy a single criterion and 70.17% more effective than the default combination of all eight criteria. Given a fixed search budget, we recommend pairing a criterion focused on structural exploration—such as Branch Coverage—with targeted supplemental strategies aimed at the type of faults expected from the system under test. Our findings offer lessons to consider when selecting such combinations.

Keywords: Search-based Test Generation, Automated Test Generation, Adequacy Criteria, Search-based Software Engineering

1 Introduction

With the exponential growth in the complexity of software, the cost of testing has risen accordingly. One way to lower costs without sacrificing quality may lie in automating the generation of test input [1]. Consider search-based generation—given a testing goal, and a scoring function denoting *closeness to attainment of that goal*, optimization algorithms can search for input that achieves that goal [12].

As we cannot know what faults exist *a priori*, dozens of criteria—ranging from the measurement of structural coverage to the detection of synthetic faults [14]—have been proposed to judge testing *adequacy*. In theory, if such criteria are fulfilled, tests should be *adequate* at detecting faults. Adequacy criteria are important for search-based generation, as they can guide the search [12].

Search techniques have improved greatly in terms of achieved coverage [2]. However, the primary goal of testing is not coverage, but fault detection. In this regard, automated generation often does not produce human competitive results [2, 3, 5, 15].

If automation is to impact testing practice, it must match—or, ideally, outperform—manual testing in terms of fault-detection efficacy.

The current use of adequacy criteria in automated generation sharply contrasts how such criteria are used by humans. For a human, coverage typically serves an advisory role—as a way to point out gaps in existing efforts. Human testers build suites in which adequacy criteria contribute to a *multifaceted* combination of testing strategies. Previous research has found that effectiveness of a criterion can depend on factors such as how expressions are written [4] and the types of faults that appear in the system [6]. Humans understand such concepts. They build and vary their testing strategy based on the needs of their current target. Yet, in automated generation, coverage is typically *the* goal, and a single criterion is applied at a time.

However, search-based techniques need not be restricted to one criterion at a time. The test obligations of multiple criteria can be combined into a single score or simultaneously satisfied by multi-objective optimization algorithms. We hypothesize that the efficacy of automated generation can be improved by applying a targeted, multifaceted approach—where multiple testing strategies are selected and simultaneously explored.

In order to examine the efficacy of suites generated by combining criteria, we have used EvoSuite and eight coverage criteria to generate multi-criteria test suites—as suggested by three selection strategies—with efficacy judged against the Defects4J fault database [10]. Based on experimental observations, we added additional configurations centered around the use of two criteria, Exception and Method Coverage, that performed poorly on their own, but were effective in combination with other criteria.

By examining the proportion of suites that detect each fault for each configuration, we can examine the effect of combining coverage criteria on the efficacy of search-based test generation, identify the configurations that are more effective than single-criterion generation, and explore situations where particular adequacy criteria can effectively cooperate to detect faults. To summarize our findings:

- For all systems, at least one combination is more effective than a single criterion, offering efficacy improvements of 14.84-31.15% over the best single criterion.
- The most effective combinations pair a structure-focused criterion—such as Branch Coverage—with supplemental strategies targeted at the class under test.
 - Across the board, effective combinations include Exception Coverage. As it can be added to a configuration with minimal effect on generation complexity, we recommend it as part of any generation strategy.
 - Method Coverage can offer an additional low-cost efficacy boost.
 - Additional targeted criteria—such as Output Coverage for code that manipulates numeric values or Weak Mutation Coverage for code with complex logical expressions—offer further efficacy improvements.

Our findings offer lessons to consider when selecting such combinations, and a starting point in discovering the best combination for a given system.

2 Background

As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation

must be used to measure the adequacy of testing efforts. Common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, points where control can branch, and complex boolean expressions [7].

The idea of measuring adequacy through coverage is simple, but compelling: unless code is executed, many faults are unlikely to be found. If tests execute elements in the manner prescribed by the criterion, than testing is deemed “adequate” with respect to faults that manifest through such structures. Adequacy criteria have seen widespread use in software development, as they offer clear checklists of testing goals that can be objectively evaluated and automatically measured [7]. Importantly, they offer *stopping criteria*, advising on when testing can conclude. These very same qualities make adequacy criteria ideal for use as automated test generation targets.

Of the thousands of test cases that could be generated for any SUT, we want to select—systematically and at a reasonable cost—those that meet our goals [12]. Given scoring functions denoting *closeness to the attainment of those goals*—called *fitness functions*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*). Metaheuristics are often inspired by natural phenomena. For example, genetic algorithms evolve a group of candidate solutions by filtering out bad “genes” and promoting fit solutions [2].

Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world program is large and complex. Metaheuristic search—by strategically sampling from that space—can scale effectively to large problems. Such approaches have been applied to a wide variety of testing scenarios [1].

While adequacy has been used in almost all generation methods, it is particularly relevant to metaheuristic search-based generation. In search-based generation, the fitness function must capture the testing objective and provide feedback to guide the search. Through this guidance, the fitness function has a major impact on the quality of the solutions generated. Adequacy criteria are common optimization targets for automated test case generation, as they can be straightforwardly transformed into distance functions that lead to the search to better solutions [12].

3 Study

We hypothesize that the efficacy of automated generation can be improved by selecting and simultaneously exploring a combination of testing strategies. In particular—in this project—we are focused on combinations of common adequacy criteria.

Rojas et al. previously found that multiple fitness functions could be combined with minimal loss in coverage of any single criterion and with a reasonable increase in test suite size [14]. Indeed, recent versions of the EvoSuite framework¹ now, by default, combine eight coverage criteria when generating tests. However, their work did not assess the effect of combining criteria on the fault-detection efficacy of the generated suites. We intend to focus on the performance of suites generated using a combination of criteria. In particular, we wish to address the following research questions:

1. Do test suites generated using a combination of two or more coverage criteria have a higher likelihood of fault detection than suites generated using a single criterion?

¹ Available from <http://evosuite.org>

2. For each system, and across all systems, which combinations are most effective?
3. What effect does an increased search budget have on the studied combinations?
4. Which criteria best pair together to increase the likelihood of fault detection?

The first two questions establish a basic understanding of the effectiveness of criteria combinations—given fixed search budgets, are *any* of the combinations more effective at fault detection than suites generated to satisfy a single criterion? Further, we hypothesize that the most effective combination will vary across systems. We wish to understand the degree to which results differ across the studied systems, and whether the search budget plays a strong role in determining the resulting efficacy of a combination. In each of these cases, we would also like to better understand *why* and *when* particular combinations are effective.

In order to examine the efficacy of suites generated using such combinations, we have first applied EvoSuite and eight coverage criteria to generate test suites for the systems in the Defects4J fault database [10]. We have performed the following:

1. **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 3.1).
2. **Generated Test Cases:** For each fault, we generated 10 suites per criterion using the fixed version of each class-under-test (CUT). We use both a two-minute and a ten-minute search budget per CUT (Section 3.2).
3. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (Section 3.2).
4. **Assessed Fault-finding Effectiveness:** For each fault, we measure the proportion of test suites that detect the fault to the number generated.

Following single-criterion generation, we applied three different selection strategies to build sets of multi-criteria configurations for each system (described in Section 3.3). We generated suites, following the steps above, using each of the configurations suggested by these strategies, as well as EvoSuite’s default eight-criteria configuration. Based on our initial observations, we added additional configurations centered around two criteria—Exception and Method Coverage—that performed poorly on their own, but were effective in combination with other criteria (See Section 4).

3.1 Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [10]². Currently, it consists of 357 faults from five projects: JFreeChart (26 faults), Closure compiler (133 faults), Apache Commons Lang (65 faults), Apache Commons Math (106 faults), and JodaTime (27 faults). Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 353.

3.2 Test Suite Generation

EvoSuite uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions.

² Available from <http://defects4j.org>

It is actively maintained and has been successfully applied to a variety of projects [15]. We used the following fitness functions, corresponding to common coverage criteria³:

Branch Coverage (BC): A test suite satisfies BC if all control-flow branches are taken by at least one test case—the test suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one whose execution evaluates to `false`. To guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how “close” the targeted predicate is to being `true`, using a cost function based on the predicate formula [14].

Direct Branch Coverage (DBC): Branch Coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. DBC requires each branch to be covered through a direct call.

Line Coverage (LC): A test suite satisfies LC if it executes each non-comment line of code at least once. To cover each line, EvoSuite tries to ensure that each basic code block is reached. The branch distance is computed for each branch that is a control dependency of any of the statements in the CUT. For each conditional statement that is a control dependency for some other line, EvoSuite requires that the branch of the statement leading to the dependent code is executed.

Exception Coverage (EC): EC rewards test suites that force the CUT to throw more exceptions—either declared or undeclared. As the number of possible exceptions that a class can throw cannot be known ahead of time, the fitness function rewards suites that throw the largest observed number of exceptions.

Method Coverage (MC): MC simply requires that all methods in the CUT be executed at least once, either directly or indirectly.

Method Coverage (Top-Level, No Exception) (MNEC): Test suites sometimes achieve MC while calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.

Output Coverage (OC): OC rewards diversity in method output by mapping return types to abstract values. A test suite satisfies OC if, for each public method, at least one test yields a concrete value characterized by each abstract value. For numeric data types, distance functions guide the search by comparing concrete and abstract values.

Weak Mutation Coverage (WMC): Suites that detect more mutants may be effective at detecting real faults as well. A test suite satisfies WMC if, for each mutated statement, at least one test detects the mutation. The search is guided by the *infection distance*, a variant of branch distance tuned towards detecting mutated statements.

To generate for multiple criteria, EvoSuite calculates the fitness score as a linear combination of the objectives for all of the criteria [14]. No ordering is imposed on the criteria when generating—combinations such as BC-LC and LC-BC are equivalent.

Test suites are generated for each class reported as faulty, using the fixed version of the CUT. They are applied to the faulty version in order to eliminate the oracle problem. This translates to a regression testing scenario, where tests guard against future issues.

³ Rojas et. al provide a primer on each fitness function [14].

Two search budgets were used—two minutes and ten minutes per class—allowing us to examine the effect of increasing the search budget. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget, generating an initial pool of 56,480 test suites.

Generation tools may generate flaky (unstable) tests [15]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, non-compiling test cases are removed. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, less than one percent of the tests are removed from each suite.

3.3 Selecting Criteria Combinations

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	45.00%	4.66%	34.00%	27.94%	34.82%	22.07%
	600	48.46%	5.79%	40.15%	32.75%	39.26%	25.61%
DBC	120	34.23%	5.11%	30.00%	24.51%	31.11%	19.43%
	600	40.77%	6.09%	38.77%	28.63%	40.37%	23.80%
EC	120	22.31%	1.35%	7.54%	6.37%	9.26%	6.09%
	600	21.54%	0.98%	9.23%	7.06%	9.63%	6.43%
LC	120	38.85%	4.14%	31.23%	25.78%	30.00%	19.92%
	600	46.15%	4.81%	34.31%	29.22%	36.67%	22.78%
MC	120	30.77%	1.58%	7.54%	10.98%	8.15%	8.05%
	600	30.77%	2.26%	7.69%	10.88%	8.15%	8.30%
MNEC	120	23.46%	2.18%	6.62%	12.16%	6.67%	7.79%
	600	30.77%	1.88%	7.54%	12.06%	5.19%	8.24%
OC	120	21.15%	2.03%	7.85%	16.57%	9.63%	9.29%
	600	23.85%	2.56%	10.92%	16.76%	12.22%	10.51%
WMC	120	38.08%	4.44%	24.15%	23.04%	25.19%	17.51%
	600	46.15%	5.56%	32.15%	27.45%	27.04%	21.42%

Table 1. Average likelihood of fault detection for single-criterion generation, divided by budget and system.

Branch Coverage follow with a 19.92-22.78% and 19.43-23.80% likelihood of detection. DBC and WMC benefit the most from an increased search budget, with average improvements of 22.45% and 22.33%. Criteria with distance-driven fitness functions—BC, DBC, LC, WMC, and, partially, OC—improve the most given more time.

We seek to identify whether combinations of criteria can outperform the top single criterion for each system—either BC or DBC. Studying all possible combinations is infeasible—even restricting to combinations of four criteria would result in 1,680 configurations. To control experiment cost, we have employed three strategies to suggest combinations. We have also used Evosuite’s default configuration of all eight criteria. We perform selection using each strategy for each system and search budget, then build combinations using the entire pool of faults in order to suggest general combinations. The suggested combinations are then used to generate suites. All combinations are applied ten times per search budget. To converse space, we do not list all combinations. However, the top combinations are named and discussed in Section 4. The three selection strategies include:

“Top-Ranked” Strategy: This simple strategy build combinations by selecting the top two to four coverage criteria, as seen in Table 1, and combining them.

Overall, suites generated to satisfy a single criterion detect 180 (50.99%) of the 353 studied faults. The average likelihood of fault detection is listed for each single criterion, by system and budget, in Table 1.

BC is the most effective single criterion, detecting 158 of the 353 faults. BC suites have an average likelihood of fault detection of 22.07% given a two-minute search budget and 25.61% given a ten-minute budget. Line and Direct

“Boost” Strategy: This strategy aims to select secondary criteria that “back up” a central criterion—the criterion that performed best on its own—in situations where that central criterion performs poorly. To select the additional criteria, the pool of faults is filtered for examples where that central criterion had $\leq 30\%$ likelihood of fault detection. Then, two to four top-ranked criteria from that filtered pool are selected for the combination. Criteria are only selected if they are more effective than the central criterion post-filtering.

“Unique Faults” Strategy: This greedy strategy favors the criteria that detect the highest number of unique faults. Combinations of criteria are selected by choosing the criterion that produced suites that detected the most faults, removing those faults from consideration, and choosing from the remaining faults and criteria. Ties are broken at random. Selection stops once four criteria are chosen, or if no faults remain.

Together, these three strategies (and EvoSuite’s default) yielded 12 combinations for Chart, 12 for Closure, 17 for Lang, 14 for Math, and 11 for Time—resulting in the generation of 94,760 test suites.

4 Results & Discussion

Tables 2–3 show—for each system, and across all systems—the combinations that are more effective than the top single criterion for the two-minute and ten-minute search budgets. We also list the performance of EvoSuite’s default combination of eight criteria. Combinations outperformed by the top criterion, except for the default, are omitted. For the overall results, only combinations generated for all systems are considered. The abbreviations for each criterion are listed in Section 3.2. For each combination, we note the strategies that suggested the combination and the average efficacy. In Table 3, we also note the improvement from the increased budget.

For all systems and both budgets, at least one combination outperforms the top single criterion. This validates our core hypothesis—the likelihood of fault detection can be increased by combining criteria. As can be seen in Tables 2 and 3, EvoSuite’s default combination of all eight criteria rarely manages to outperform the top single criterion. As the number of criteria expands, the difficulty of the search process also grows. While criteria can work together to produce more effective suites, there is a point where the generation task becomes too difficult to achieve within the selected budget. Our observations, instead, point towards the wisdom of choosing a *targeted* set of criteria for the CUT.

We proposed three strategies to suggest combinations—the Top-Ranked, Boosting, and Unique Faults strategies. Examining the results of this experiment, the Unique Faults strategy seems to produce the best overall results, suggesting nine of the top strategies for the two-minute budget and 24 for the ten-minute budget. The Boosting strategy suggests only three for the two-minute budget and 17 for the ten-minute budget, and the Top Ranked strategy suggests seven for the two-minute budget and 14 for the ten-minute budget.

However, while these strategies have yielded effective combinations, we cannot recommend any of them as a general-purpose strategy for testing new systems. Each also suggested a large number of ineffective combinations. With a two-minute budget, only

System	Combination	Strategy	Efficacy
Chart	Default	-	47.30%
	BC	-	45.00%
	BC-LC	TR, UF	6.00%
Closure	DBC	-	5.10%
	Default	-	4.50%
	BC-EC-LC-MC	UF	40.00%
Lang	BC-EC	UF	39.40%
	BC-LC	TR, BS, UF	36.50%
	BC-EC-LC	UF	35.70%
	BC-DBC	TR, BS, UF	35.50%
	BC-LC-DBC	TR, BS	34.00%
	BC	-	34.00%
	Default	-	23.80%
	BC-LC-OC-EC	UF	32.40%
	BC-LC	TR, UF	31.90%
Math	BC	-	27.90%
	Default	-	25.80%
	DBC-BC-LC	TR, BS	35.20%
Time	BC	-	34.80%
	Default	-	25.90%
	BC-LC	TR, UF	24.00%
Overall	BC	-	22.10%
	Default	-	19.00%

Table 2. Efficacy (average likelihood of fault detection) for the initial set of combinations, when generated with a two-minute search budget. TR = Top-Ranked Strategy, BS = Boosting Strategy, UF = Unique Faults Strategy.

System	Combination	Strategy	Efficacy	Improvement From Budget
Chart	BC-LC-WMC-EC	UF	57.30%	35.46%
	BC-EC-DBC	BS	55.80%	28.28%
	BC-EC	BS	54.60%	23.53%
	BC-DBC-WMC-OC	UF	49.20%	54.23%
	BC-LC-WMC	TR, UF	48.90%	49.71%
	BC-WMC-MC	BS	48.90%	27.01%
	BC-WMC	BS, UF	48.80%	39.43%
	BC	-	48.50%	7.78%
Closure	Default	-	48.10%	1.69%
	BC-LC	TR, UF	7.6%	26.67%
	BC-LC-DBC	TR	7.30%	48.98%
	BC-LC-WMC-EC	UF	7.20%	94.59%
	BC-DBC	TR, BS, UF	7.10%	51.05%
	Default	-	7.10%	57.78%
	BC-WMC-DBC-LC	TR, BS	7.00%	42.86%
	DBC-WMC	BS, UF	6.80%	61.90%
	BC-WMC	BS	6.50%	54.76%
	DBC-WMC-BC	TR, BS, UF	6.40%	60.00%
Lang	DBC	-	6.10%	19.61%
	BC-EC	UF	47.50%	20.56%
	BC-EC-LC-MC	UF	45.70%	14.25%
	BC-EC-LC	UF	45.70%	28.00%
	BC-LC-DBC	TR, BS	42.30%	24.41%
	BC-LC-WMC-EC	UF	41.70%	44.79%
	BC-DBC	TR, BS, UF	41.10%	12.60%
	BC-LC	TR, BS, UF	40.90%	12.05%
	BC	-	40.20%	18.24%
	Default	-	32.60%	36.97%
Math	BC-LC-OC-EC	UF	38.00%	17.28%
	BC-OC-LC	BS, UF	35.80%	31.62%
	BC-OC	BS	34.70%	24.82%
	BC-OC-LC-WMC	BS, UF	33.70%	22.99%
	BC-LC	TR, UF	31.90%	3.76%
	BC-LC-WMC-EC	UF	33.00%	21.32%
	BC	-	32.80%	17.56%
	Default	-	32.80%	27.13%
Time	DBC-BC	TR, BS, UF	43.30%	39.22%
	DBC-BC-LC	TR, BS	41.50%	17.90%
	DBC	-	40.40%	29.90%
	Default	-	33.33%	28.68%
	BC-LC-WMC-EC	UF	26.90%	33.83%
Overall	BC-LC	TR, UF	26.40%	10.00%
	BC-DBC	TR, BS, UF	25.80%	20.00%
	BC-LC-DBC	TR	25.60%	28.00%
	BC	-	25.60%	15.84%
	Default	-	24.20%	27.39%

Table 3. Efficacy for the initial set of combinations (ten-minute search budget).

28% of the the Top Ranked strategy combinations were effective. For the Boosting strategy, the total was only 8%, and for the Unique Faults strategy, the total was only 19%. With a ten-minute budget, 56% of the combinations for the Top Ranked strategy were effective. For the Boosting strategy, this was 45%, and for the Unique Faults strategy, the total was 50%. Therefore, while the basic hypothesis—that combinations *can* outperform a single criterion—seems to be valid, more research is needed in how to determine the best combination.

4.1 Additional Configurations

Following test generation, we noticed that (a) Exception Coverage was frequently selected as part of combinations, despite performing poorly on its own, and (2), that these combinations are often highly effective. For example, the top combinations for Chart, Lang, and Math in Tables 2-3 all contain EC. Of the studied criteria, EC is unique in that it does not prescribe static test obligations. Rather, it simply rewards suites that

System	Combination	Efficacy
Chart	<i>Default</i>	47.30%
	<i>BC</i>	45.00%
	<i>BC-LC</i>	6.00%
	<i>BC-LC-EC</i>	5.70%
	<i>DBC-EC</i>	5.10%
	<i>DBC</i>	5.10%
	<i>Default</i>	4.50%
Closure	<i>BC-EC-LC-MC</i>	40.00%
	<i>BC-EC*</i>	39.40%
	<i>BC-LC-EC*</i>	35.70%
	<i>BC</i>	34.00%
	<i>Default</i>	23.80%
	<i>BC-LC-OC-EC</i>	32.40%
	<i>BC-EC</i>	31.70%
Lang	<i>BC</i>	27.90%
	<i>Default</i>	25.80%
	<i>BC-EC</i>	39.60%
	<i>DBC-BC-LC-EC</i>	39.30%
	<i>DBC-EC</i>	37.80%
	<i>DBC-BC-LC</i>	35.20%
	<i>BC</i>	34.80%
Math	<i>Default</i>	25.90%
	<i>BC-EC</i>	24.50%
	<i>BC-LC</i>	24.00%
	<i>BC-LC-EC</i>	22.40%
	<i>BC</i>	22.10%
	<i>Default</i>	19.00%
	<i>BC-EC-LC-MC</i>	40.00%
Time	<i>BC-EC*</i>	39.40%
	<i>BC-LC-EC*</i>	35.70%
	<i>BC</i>	34.00%
	<i>Default</i>	23.80%
	<i>BC-EC</i>	39.60%
	<i>DBC-BC-LC-EC</i>	39.30%
	<i>DBC-EC</i>	37.80%
Overall	<i>DBC-BC-LC</i>	35.20%
	<i>BC</i>	34.80%
	<i>Default</i>	25.90%
	<i>BC-EC</i>	24.50%
	<i>BC-LC</i>	24.00%
	<i>BC-LC-EC</i>	22.40%
	<i>BC</i>	22.10%
<i>Default</i>	<i>Default</i>	19.00%

Table 4. Efficacy of Exception Coverage-based combinations (two-minute budget). The top combination from Table 2, top single criterion, and EvoSuite’s default combination are shown for context. A * means that the combination was also suggested by a previous strategy.

System	Combination	Efficacy	Improvement From Budget
Chart	<i>BC-LC-WMC-EC</i>	57.30%	35.46%
	<i>BC-EC*</i>	54.60%	23.53%
	<i>BC-LC-EC</i>	50.40%	14.81%
	<i>BC</i>	48.50%	7.78%
	<i>Default</i>	48.10%	1.69%
	<i>BC-LC</i>	7.6%	26.67%
	<i>BC-LC-EC</i>	7.40%	29.82%
Closure	<i>BC-EC</i>	7.10%	47.92%
	<i>Default</i>	7.10%	57.78%
	<i>DBC</i>	6.10%	19.61%
	<i>BC-EC*</i>	47.50%	20.56%
	<i>BC-LC-EC*</i>	45.70%	28.00%
	<i>BC</i>	40.20%	18.24%
	<i>Default</i>	32.60%	36.97%
Lang	<i>BC-LC-OC-EC</i>	38.00%	17.28%
	<i>BC-EC</i>	34.00%	7.25%
	<i>BC</i>	32.80%	17.56%
	<i>Default</i>	32.80%	27.13%
	<i>BC-EC</i>	44.80%	13.13%
	<i>DBC-BC-EC</i>	44.10%	38.25%
	<i>DBC-BC</i>	43.30%	39.22%
Time	<i>DBC-BC-LC-EC</i>	42.60%	8.40%
	<i>BC-LC-EC</i>	41.10%	18.10%
	<i>DBC-EC</i>	41.10%	8.73%
	<i>DBC</i>	40.40%	29.90%
	<i>Default</i>	33.33%	28.68%
	<i>BC-EC</i>	28.70%	17.14%
	<i>BC-LC-EC</i>	27.50%	22.77%
Overall	<i>BC-LC-WMC-EC</i>	26.90%	33.83%
	<i>BC</i>	25.60%	15.84%
	<i>Default</i>	24.20%	27.39%

Table 5. Efficacy of EC-based combinations (ten-minute budget). The top combination from Table 3, top single criterion, and EvoSuite’s default combination are shown for context.

cause more exceptions to be thrown. This means that it can be added to a combination with little increase in search complexity.

To further study the potential of EC as a “low-cost” addition to combinations, we added a set of additional combinations to our study. Specifically, we generated tests for all systems using the BC-EC and BC-LC-EC combinations—the combination of EC with the overall best single criterion, and the combination of EC with the best overall combination seen to this point. As DBC outperformed BC for the Closure and Time systems, we also generated tests for the DBC-EC combination in those two cases. Finally, as the top-ranked combination Time lacked EC, we added the DBC-BC-LC-EC and DBC-BC-EC combinations for that system. In total, this adds one new configuration for Chart, three for Closure, zero for Lang, two for Math, and five for Time—resulting in the generation of an additional 15,280 test suites.

Results can be seen in Tables 4 and 5 for the two budgets. From these results, we can see that—almost universally—the best observed combination of criteria includes Exception Coverage. In fact, the best overall configuration—up to this point—is a simple combination of BC and EC. The simplicity of EC explains its poor performance as the *primary* criterion. It lacks a feedback mechanism to drive generation towards exceptions. However, EC appears to be effective *when paired with criteria that effectively explore the structure of the CUT*, such as Branch or Line Coverage. Exception Cover-

System	Combination	Efficacy
Chart	<i>Default</i>	47.30%
	<i>BC</i>	45.00%
Closure	<i>BC-LC</i>	6.00%
	<i>BC-LC-EC</i>	5.70%
	<i>BC-EC-MC</i>	5.60%
	<i>BC-LC-MC</i>	5.30%
	<i>DBC</i>	5.10%
	<i>Default</i>	4.50%
Lang	<i>BC-EC-MC</i>	40.50%
	<i>BC-EC-LC-MC*</i>	40.00%
	<i>BC-EC</i>	39.40%
	<i>BC-LC-MC</i>	38.00%
	<i>BC</i>	34.00%
	<i>Default</i>	23.80%
Math	<i>BC-LC-OC-EC-MC</i>	32.90%
	<i>BC-LC-OC-EC</i>	32.40%
	<i>BC-EC</i>	31.70%
	<i>BC-EC-MC</i>	30.40%
	<i>BC-EC-LC-MC</i>	30.20%
	<i>BC</i>	27.90%
Time	<i>Default</i>	25.80%
	<i>BC-EC</i>	39.60%
	<i>BC-EC-MC</i>	35.90%
	<i>DBC-BC-LC</i>	35.20%
	<i>BC</i>	34.80%
	<i>Default</i>	25.90%
Overall	<i>BC-EC</i>	24.50%
	<i>BC-EC-MC</i>	24.30%
	<i>BC-LC</i>	24.00%
	<i>BC-EC-LC-MC</i>	23.60%
	<i>BC-LC-MC</i>	22.20%
	<i>BC</i>	22.10%
	<i>Default</i>	19.00%

Table 6. Efficacy of Method Coverage-based combinations (two-minute budget). The top combinations from Tables 2 and 4, top single criterion, and EvoSuite’s default combination are shown for context. A * means that the combination was also suggested by a previous strategy.

age adds little cost in terms of generation difficulty, and almost universally outperforms the use of Branch Coverage alone.

An example of effective combination can be seen in fault 60 for Lang⁴—a case where two methods can look beyond the end of a string. No single criterion is effective, with a maximum of 10% chance of detection given a two-minute budget and 20% with a ten-minute budget. However, combining BC and EC boosts the likelihood of detection to 40% and 90% for the two budgets. In this case, if the fault is triggered, the incorrect string access will cause an exception to be thrown. However, this only occurs under particular circumstances. Therefore, EC alone never detects the fault. BC provides the necessary means to drive program execution to the correct location. However, two suites with an equal coverage score are considered equal. BC alone may prioritize suites with slightly higher (or different) coverage, missing the fault. By combining

System	Combination	Efficacy	Improvement From Budget
Chart	<i>BC-LC-WMC-EC</i>	57.30%	35.46%
	<i>BC-EC</i>	54.60%	23.53%
	<i>BC-EC-MC</i>	53.90%	25.06%
	<i>BC-LC-EMC-EC-MC</i>	53.50%	19.96%
	<i>BC</i>	48.50%	7.78%
	<i>Default</i>	48.10%	1.69%
Closure	<i>BC-EC-MC</i>	8.00%	42.86%
	<i>BC-EC-LC-MC</i>	7.70%	54.00%
	<i>BC-LC</i>	7.6%	26.67%
	<i>BC-LC-EC</i>	7.40%	29.82%
	<i>Default</i>	7.10%	57.78%
	<i>DBC</i>	6.10%	19.61%
Lang	<i>BC-EC-MC</i>	48.20%	19.01%
	<i>BC-EC</i>	47.50%	20.56%
	<i>BC-EC-LC-MC*</i>	45.70%	14.25%
	<i>BC-LC-MC</i>	43.70%	15.00%
	<i>BC</i>	40.20%	18.24%
	<i>Default</i>	32.60%	36.97%
Math	<i>BC-LC-OC-EC-MC</i>	39.00%	18.54%
	<i>BC-LC-OC-EC</i>	38.00%	17.28%
	<i>BC-EC-MC</i>	34.30%	12.83%
	<i>BC-EC-LC-MC</i>	34.10%	12.91%
	<i>BC-EC</i>	34.00%	7.25%
	<i>BC</i>	32.80%	17.56%
Time	<i>Default</i>	32.80%	27.13%
	<i>BC-EC-MC</i>	47.00%	30.92%
	<i>BC-EC</i>	44.80%	13.13%
	<i>DBC-BC</i>	43.30%	39.22%
	<i>BC-EC-LC-MC</i>	41.10%	18.10%
	<i>DBC</i>	40.40%	29.90%
Overall	<i>Default</i>	33.33%	28.68%
	<i>BC-EC-MC</i>	29.40%	20.99%
	<i>BC-EC</i>	28.70%	17.14%
	<i>BC-EC-LC-MC</i>	27.80%	17.80%
	<i>BC-LC-WMC-EC</i>	26.90%	33.83%
	<i>BC-LC-MC</i>	25.60%	15.31%
	<i>BC</i>	25.60%	15.84%
	<i>Default</i>	24.20%	27.39%

Table 7. Efficacy of MC-based combinations (ten-minute budget). Top combinations from Tables 3 and 5, top single criterion, and EvoSuite’s default combination are shown for context.

⁴ <https://github.com/apache/commons-lang/commit/a8203b65261110c4a30ff69fe0da7a2390d82757>.

the two, exception-throwing tests are prioritized and retained, succeeding where either criterion would fail alone.

Given that EC can boost the likelihood of fault detection without a substantial cost increase, it seems reasonable to look for other “low-cost” criteria that could provide a similar effect. The two forms of Method Coverage used in this project are ideal candidates. In general, a class will not have a large number of methods, and methods are either covered or not covered. Additionally, MC also appears in some of the top combinations—such as those for Lang—despite poor performance on its own.

Therefore, we have also generated tests for the BC-EC-MC, BC-LC-MC, and BC-EC-LC-MC combinations for all systems. We have also added MC to the top combination for any system that did not already have one of the above as the resulting combination, adding BC-LC-WMC-EC-MC for Chart and BC-LC-OC-EC-MC for Math. In total, this adds four new combinations for Chart, three for Closure, two for Lang, four for Math, and three for Time—yielding 22,400 additional test suites.

The results of these combinations can be seen in Tables 6-7. With a two-minute budget, the addition of Method Coverage can improve results—as seen in Lang, where BC-EC-MC outperforms BC-EC, and Math, where BC-LC-OC-EC-MC outperforms BC-LC-OC-EC. However, in other cases—such as with Closure and Time—the addition of MC decreases efficacy. Results improve across the board with a ten-minute budget, where the top combinations for Closure, Lang, Math, and Time all contain MC. Overall, with a ten-minute budget, the combination of BC-EC-MC outperforms any other blanket policy. It seems that MC can improve a combination, but does not have the same impact as EC. Given a high enough search budget, we do recommend its inclusion. An example where the addition of MC could boost efficacy can be seen in Lang fault 34⁵. This fault resides in two small (1-2 line) methods. Calling either method will reveal the fault, but BC can easily overlook them.

4.2 Observations and Recommendations

We can address each research question in turn. First:

For all systems and search budgets, at least one combination of criteria is more effective than a single criterion, with the top combination offering a 5.11-31.15% improvement in the likelihood of fault detection over the best single criterion and up to 70.17% improvement over the default combination of all eight criteria.

For each budget B , combination C , and individual criterion I , we formulate hypothesis H and null hypothesis H_0 :

- H : With budget B , test suites generated using X will have a higher likelihood of fault detection than suites generated using I .
- H_0 : Observations of efficacy for C and I are drawn from the same distribution.

Due to the limited number of faults for Chart and Time, we have focused on overall results. Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H_0 without

⁵ <https://github.com/apache/commons-lang/commit/496525b0d626dd5049528cdef61d71681154b660>

any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test, a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

To save space, we focus on the top combinations—BC-EC for the two-minute budget and BC-EC-MC for the ten-minute budget. At the two-minute search budget, we can reject the null hypothesis for MC, MNEC, EC, OC (all < 0.001), and WMC (0.030). We cannot reject the null hypothesis for DBC (0.055), LC (0.057), or BC (0.304). At the ten-minute level, we can reject the null hypothesis for MC, MNEC, EC, OC, WMC, LC (all < 0.001), DBC (0.017), and BC (0.040). Therefore, the BC-EC-MC combination significantly outperforms all individual criteria, given sufficient search budget.

From Tables 3, 5, and 7, we can see that the search budget affects the efficacy of combinations. At a higher budget, more combinations outperform individual criteria, and the performance gap between combinations and individual criteria widens. While combinations *can* outperform individual criteria at the two-minute budget, a larger budget clearly benefits combinations.

As more criteria are added, the generation task becomes more complex. There is a trade-off to be made in terms of the required search budget and the efficacy of the results. The default eight-way combination of criteria, even with a ten-minute budget, is ineffective in the majority of cases. While an even higher budget may help, we have seen that simple, *targeted* combinations can perform very well, even with a tight budget.

This leads, naturally, to the next question—which combinations are effective, in practice? At the two minute budget, a combination of all eight criteria is the most effective for Chart. BC-LC is best for Closure. For Lang, it is CB-EC-MC. For Math, it is BC-LC-OC-EC-MC. Finally, for Time, it is BC-EC. The best general policy, at that budget, is the BC-EC combination. More consensus is seen at the ten-minute budget level, where the BC-EC-MC combination is the best observed for Closure, Lang, and Time (and is the best general policy). For Chart, the top combination is BC-LC-WMC-EC. For Math, it is BC-LC-OC-EC-MC.

We do not wish to advocate these as the best *possible* combinations. Even for the studied systems, we did not exhaustively try all possibilities. Further, while performance gains are reasonably significant, better performance is likely possible. Rather, we wish to use this study to derive a starting point for those who wish to generate effective tests.

Either Branch or DBC was found to be the most effective single criterion. Tests that fail to execute faulty lines of code are highly unlikely to reveal a fault, so a criterion intended to achieve code coverage should form the core of a combination. However, code coverage is not sufficient on its own. Merely executing code does not ensure a failure—*how* that code is executed is important. From our results, we can observe that the most effective combinations pair a structure-focused criterion with a small number of supplemental strategies that can guide the structure-based criterion towards the *correct* input for the CUT.

Across the board, effective combinations include Exception Coverage. As EC can be added to a combination with minimal effect on generation complexity, we recommend it as part of any generation strategy. Although Method Coverage does not have the clear

symbiotic relationship with BC that EC has, it offers a slight boost to efficacy at a low cost. We recommend its inclusion in combinations with a longer search budget.

We recommend a combination of Branch or Direct Branch Coverage with Exception and Method Coverage as a *base* approach to test generation. Additional criteria, targeted towards the CUT, may further improve efficacy.

We observed several situations where the central structure-based criterion is boosted by secondary criteria. First, Output Coverage often assists in revealing faults for Math. OC divides the data type of the method output into a series of abstract values, then rewards suites that cover each of those classes. In particular, OC offers the search feedback for numeric data types [14], explaining its utility for Math. For example, consider Fault 53⁶. The patch removes a misbehaving check for *Nan*. As the fixed version *removes* code, BC does not reveal the fault. However, Output Coverage ensures that method calls return a variety of values—raising the likelihood of fault detection.

Weak Mutation Coverage can also boost BC. Consider Lang fault 28⁷. BC alone fails to detect the fault, while WMC alone has a 40% chance of detection. A BC-WMC combination has a 90% chance of detection. The patched code includes an *if*-condition that can be mutated in several ways. BC assists in mutation detection by driving execution to, and into, the *if*-block. This combination is effective for other similar faults.

Combining structure-focused criteria seems potentially redundant. However, BC-LC and BC-DBC combinations can be effective (see Table 7). Consider Closure fault 94⁸. No single criterion detects the fault. However, at the ten-minute budget, the BC-LC combination has a 30% detection likelihood. The BC-LC combination is not only more effective, but also achieves higher levels of coverage. BC suites attain an average of 54.91% LC and 37.46% BC. LC-based suites attain 58.94% LC and 33.99% BC. Suites generated using the combination achieve 59.09% LC and 42.45% BC. By attaining higher coverage, the combination is more likely to execute the faulty code.

While more research is needed to identify situations where criteria work well together, developers should be able to produce more effective test cases using automated generation by considering the CUT and choosing criteria accordingly.

5 Related Work

Advocates of adequacy criteria hypothesize that there should exist a correlation between higher attainment of a criterion and fault detection efficacy [7]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [13, 5, 8]. Inozemtseva et al. provide a good overview [8].

Shamshiri et al. applied EvoSuite (Branch Coverage only), Randoop, and Agitar to each fault in Defects4J to assess the fault-detection capabilities of automated generation [15]. They found that the combination of tools could detect 55.70% of the faults.

⁶ <https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/53.src.patch>

⁷ <https://github.com/apache/commons-lang/commit/3e1afec200d7e3be9537c95b7cf52a7c5031300>

⁸ <https://github.com/rjust/defects4j/blob/master/framework/projects/Closure/patches/94.src.patch>

Their work identifies several reasons why faults were not detected, including low levels of coverage, heavy use of private methods and variables, and issues with simulation of the execution environment. Our recent experiments expand on this work, comparing fitness functions from EvoSuite in terms of fault detection efficacy [3].

Rojas et al. previously found that, given a fixed generation budget, multiple fitness functions could be combined with minimal loss in coverage of any single criterion and with a reasonable increase in test suite size [14]. Others have explored combinations of coverage criteria with non-functional criteria, such as memory consumption [11] or execution time [16]. Few have studied the effect of such combinations on fault detection. Jeffrey et al. found that combinations are effective following suite reduction [9].

6 Threats to Validity

External Validity: We have focused on five systems. We believe such systems are representative of small to medium-sized open-source Java systems, and that we have examined a sufficient number of faults to offer generalizable results.

We have used only one test generation framework. While other techniques may yield different results, no other framework offers the same variety of coverage criteria. Therefore, a more thorough comparison of tool performance cannot be made. While exact results may differ, we believe that general trends will remain the same, as the underlying criteria follow the same philosophy.

To control costs, we have only performed ten trials per combination of fault, budget, and configuration. Additional trials may yield different results. However, we believe that 134,300 suites is a sufficient number to draw conclusions.

Conclusion Validity: When using statistical analysis, we ensure base assumptions are met. We use non-parametric methods, as distribution characteristics are not known.

7 Conclusions

In this work, we have examined the effect of combining coverage criteria on the efficacy of search-based test generation, identified effective combinations, and explored situations where criteria can cooperate to detect faults. For all systems, we have found that at least one combination is more effective than individual criteria, with the top combinations offering up to a 31.15% improvements in efficacy over top individual criteria. The most effective combinations pair a criterion focused on structure exploration—such as Branch Coverage—with a small number of targeted supplemental strategies suited to the CUT. Our findings offer lessons to consider when selecting such combinations.

Although we recommend the combination of Branch, Exception, and Method Coverage as a starting point, further research is needed to determine how to select the best combination for a system. In future work, we plan to focus on automated means of selecting combinations, perhaps using hyperheuristic search.

8 Acknowledgements

This work is supported by National Science Foundation grant CCF-1657299.

References

1. S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.
2. G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA, pages 291–301, New York, NY, USA, 2013. ACM.
3. G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the 2017 International Conference on Software Testing*, ICST 2017. IEEE, 2017.
4. G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Trans. Softw. Eng. Methodol.*, 25(3):25:1–25:34, July 2016.
5. G. Gay, M. Staats, M. Whalen, and M. Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, PP(99), 2015.
6. R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *25th International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014.
7. A. Groce, M. A. Alipour, and R. Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'14, pages 255–268, New York, NY, USA, 2014. ACM.
8. L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.
9. D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, Feb 2007.
10. R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.
11. K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, New York, NY, USA, 2007. ACM.
12. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.
13. A. Mockus, N. Nagappan, and T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301, Oct 2009.
14. J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.
15. S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.
16. S. Yoo and M. Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689 – 701, 2010.

RESEARCH ARTICLE

Choosing The Fitness Function for the Job: Automated Generation of Test Suites that Detect Real Faults

Alireza Salahirad, Hussein Almulla, Gregory Gay

¹Department of Computer Science & Engineering,
University of South Carolina, Columbia, SC,
United States of America

Correspondence

*Email: alireza@email.sc.edu,
halmulla@email.sc.edu, greg@greggay.com

Summary

Search-based unit test generation, if effective at fault detection, can lower the cost of testing. Such techniques rely on fitness functions to guide the search. Ultimately, such functions represent test goals that approximate—but do not ensure—fault detection. The need to rely on approximations leads to two questions—*can fitness functions produce effective tests and, if so, which should be used to generate tests?* To answer these questions, we have assessed the fault-detection capabilities of unit test suites generated to satisfy eight white-box fitness functions on 597 real faults from the Defects4J database. Our analysis has found that the strongest indicators of effectiveness are a high level of code coverage over the targeted class and high satisfaction of a criterion’s obligations. Consequently, the branch coverage fitness function is the most effective. Our findings indicate that fitness functions that thoroughly explore system structure should be used as primary generation objectives—supported by secondary fitness functions that explore orthogonal, supporting scenarios. Our results also provide further evidence that future approaches to test generation should focus on attaining higher coverage of private code and better initialization and manipulation of class dependencies.

KEYWORDS:

Search-based Test Generation, Automated Test Generation, Unit Testing, Adequacy Criteria, Search-based Software Engineering

1 | INTRODUCTION

Proper verification practices are needed to ensure that developers deliver reliable software. *Testing* is an invaluable, widespread verification technique. However, testing is a notoriously expensive and difficult activity [51], and with exponential growth in the complexity of software, the cost of testing has risen accordingly. Means of lowering the cost of testing without sacrificing verification quality are needed.

Much of that cost can be traced directly to the human effort required to conduct most testing activities, such as producing test input and expected output. One way of lowering such costs may lie in the use of automation to ease this manual burden [5]. Automation has great potential in this respect, as much of the invested human effort is in service of tasks that can be framed as *search* problems [34]. For example, unit test case generation can naturally be seen as a search problem [5]. There are hundreds of thousands of test cases that could be generated for any particular class under test (CUT). Given a well-defined testing goal, and a numeric scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can systematically search the space of possible test inputs to locate those that meet that goal [45].

The effective use of search-based generation relies on the performance of two tasks—selecting a measurable test goal and selecting an effective fitness function for meeting that goal. Adequacy criteria offer checklists of measurable test goals based on the program source code, such as the execution of branches in the control-flow of the CUT [40, 55, 56]. Because such criteria are based on source code elements, we refer to them as “white-box” test selection criteria. Often, however, goals such as “coverage of branches” are an approximation of a goal that is harder to quantify—we really want tests that will reveal faults [2]. “Finding faults” is not a goal that can be measured, and cannot be translated into a distance function.

To generate effective tests, we must identify criteria—and corresponding fitness functions—that are correlated with an increased probability of fault detection. If branch coverage is, in fact, correlated with fault detection, then—even if we do not care about the concept of branch coverage itself—we will end up with effective tests. However, the need to rely on approximations leads to two questions. First, *can common fitness functions produce effective tests?* If so, *which of the many available fitness functions should be used to generate tests?* Unfortunately, testers are faced with a bewildering number of options—an informal survey of two years of testing literature reveals 28 viable white-box fitness functions—and there is little guidance on when to use one criterion over another [29].

While previous studies on the effectiveness of adequacy criteria in test generation have yielded inconclusive results [52, 48, 36, 29], two factors allow us to more deeply examine this problem—particularly with respect to search-based generation. First, tools are now available that implement enough fitness functions to make unbiased comparisons. The EvoSuite framework offers over twenty options, and uses a combination of eight fitness functions as its default configuration [20]. Second, more realistic examples are available for use in assessment of suites. Much of the previous work on adequacy effectiveness has been assessed using mutants—synthetic faults created through source code transformation [38]. Whether mutants correspond to the types of faults found in real projects has not been firmly established [31]. However, the Defects4J project offers a large database of real faults extracted from open-source Java projects [39]. We can use these faults to assess the effectiveness of search-based generation on the complex faults found in real software.

In this study, we have used EvoSuite and eight of its white-box fitness functions (as well as the default multi-objective configuration and a combination of branch, exception, and method coverage) to generate test suites for the fifteen systems, and 593 of the faults, in the Defects4J database. In each case, we seek to understand *when and why* generated test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques, and could inspire new approaches. Thus, in each case, we have recorded the proportion of suites that detect the fault and a number of factors—related to suite size, obligation satisfaction, and attained coverage. We have recorded a set of traditional *source code metrics*—sixty metrics related to cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics—for each class associated with a fault the Defects4J dataset. By analyzing these generation factors and metrics, we can begin to understand not only the real-world applicability of the fitness options in EvoSuite, but—through the use of machine learning algorithms—the factors correlating with a high or low likelihood of fault detection. To summarize our findings:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60-25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.
- While EvoSuite’s default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria.
- However, a combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion’s test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Criteria such as exception, output, and weak mutation coverage are situationally useful, and should be

applied as secondary testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

This work extends a prior conference publication [24]. The earlier paper looked at the same core research questions. However, in order to undergo a deeper investigation into the topic, we have contributed an additional 240 faults, from fifteen new systems, to Defects4J—almost doubling the size of the database. Our updated study includes suites generated over those new case examples, adding further observations and points of discussion. We have also used the findings of our separate research into combinations of fitness functions [25] to reformulate and extend our experiments and discussion of the effects of combining criteria. In addition, we have changed how we build and classify data in our treatment learning analysis, added the source code metric analysis, and have included a far deeper examination of the factors indicating success or lack thereof in test generation. Our observations provide evidence for the anecdotal findings of other researchers [8, 24, 23, 60, 7] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. While more research is still needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites, our findings in this revised and extended case study offer lessons in understanding the use, applicability, and combination of common fitness functions.

2 | BACKGROUND

2.1 | Search-Based Software Test Generation

Test case creation can naturally be seen as a search problem [34]. Of the thousands of test cases that could be generated for any SUT, we want to select—systematically and at a reasonable cost—those that meet our goals [45, 2]. Given a well-defined testing goal, and a scoring function denoting closeness to the attainment of that goal—called a *fitness function*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*) [10]. Metaheuristics are often inspired by natural phenomena, such as swarm behavior [15] or evolution [35].

While the particular details vary between algorithms, the general process employed by a metaheuristic is as follows: (1) One or more solutions are generated, (2), The solutions are scored according to the fitness function, and (3), this score is used to reformulate the solutions for the next round of evolution. This process continues over multiple generations, ultimately returning the best-seen solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic impacts the quality and efficiency of the search process [16].

Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world program is large and complex [2]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [43]. Such approaches have been applied to a wide variety of testing goals and scenarios [2].

2.2 | Adequacy Metrics and Fitness Functions

When testing, developers must judge: (a) whether the produced tests are effective and (b) when they can stop writing additional tests. These two factors are linked. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*? The same question applies when adding new tests—if we have not observed new faults, have we not yet written *adequate* tests?

The concept of adequacy provides developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of tests. The most common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, branches of the software’s control flow, and complex boolean conditional statements [40, 55, 56].

Each adequacy criterion embodies a set of lessons about effective testing—requirements tests must fulfill to be considered adequate. These requirements are expressed as a series of *test obligations*—properties that must be met by the corresponding test suite. For example, the branch coverage criterion requires that each program expression that can cause control flow to diverge—i.e., loop conditions, switch statements, and if-conditions—evaluate to each possible outcome. In this case, a test obligation would indicate a particular expression and a targeted outcome for the evaluation of that expression. If tests fulfill the list of obligations prescribed by the criterion, than testing is deemed “adequate” with respect to faults that manifest through the structures of interest to the criterion.

Adequacy criteria have seen widespread use in software development, and is routinely measured as part of automated build processes [32]¹. It is easy to understand the popularity of adequacy criteria. They offer clear checklists of testing goals that can be objectively evaluated and automatically measured [57]. These very same qualities make adequacy criteria ideal for use as automated test generation targets. In search-based testing, the fitness function needs to capture the testing objective and guide the search. Through this guidance, the fitness function has a major impact on

¹For example, see <https://codecov.io/>.

the quality of the solutions generated. Functions must be efficient to execute, as they will be calculated thousands of times over a search. Yet, they also must provide enough detail to differentiate candidate solutions and guide the selection of optimal candidates. Adequacy criteria are ideal optimization targets for automated test case generation as they can be straightforwardly transformed into efficient, informative fitness functions [6]. Search-based generation often can achieve higher coverage than developer-created tests [22].

3 | STUDY

To generate unit tests that are effective at finding faults, we must identify criteria and corresponding fitness functions that increase the probability of fault detection. As we cannot know what faults exist before verification, such criteria are approximations—intended to increase the probability of fault detection, but offering no guarantees. Thus, it is important to turn a critical eye toward the choice of fitness function used in search-based test generation. We wish to know whether commonly-used fitness functions produce effective tests, and if so, why—and under what circumstances—do they do so?

More empirical evidence is needed to better understand the relationships between adequacy criteria, fitness functions and fault detection [32]. Many criteria exist, and there is little guidance on when to use one over another [29]. To better understand the real-world effectiveness, use, and applicability of common fitness functions and the factors leading to a higher probability of fault detection, we have assessed the EvoSuite test generation framework and eight of its fitness functions (as well as the default multi-objective configuration) against 593 real faults, contained in the Defects4J database. In doing so, we wish to address the following research questions:

1. How capable are generated test suites at detecting real faults?
2. Which fitness functions have the highest likelihood of fault detection?
3. Does an increased search budget improve the effectiveness of the resulting test suites?
4. Under what situations can a combination of criteria outperform a single criterion?
5. What factors correlate with a high likelihood of fault detection?

The first three questions allow us to establish a basic understanding of the effectiveness of each fitness function—are *any* of the functions able to generate fault-detecting tests and, if so, are any of these functions more effective than others at the task? However, these questions presuppose that only one fitness function can be used to generate test suites. Many search-based generation algorithms can simultaneously target *multiple* fitness functions [25]. Therefore, we also ask question 4—when does it make sense to employ a set of fitness functions instead of a single function?

Finally, across all criteria, we also would like to gain insight into the factors that influence the likelihood of detection. To inspire new research advances, we desired a deeper understanding of when generated suites are likely to detect a fault, and when they will fail. We have made use of *treatment learning*—a machine learning technique designed to take classified data and identify sets of attributes, along with bounded values of such attributes, that are highly correlated with particular outcomes. In our case, these “outcomes” are associated with whether generated suites from each fitness function detect a fault or not. We have gathered factors from two broad sets:

- **Test Generation Factors** are related to the test suites produced—identifying coverage attained, suite size, and obligation satisfaction.
- **Source Code Metrics** examine the faulty classes being targeted, and ask whether factors related to the classes themselves—i.e., the number of private methods or cloned code—can impact the test generation process.

We have created datasets based off of both sets of factors and applied treatment learning to assess which factors strongly affected the outcome of test generation. We make these datasets, as well as the new Defects4J case examples, available to other researchers to aid in future advances (see Sections 3.1 and 3.5.3).

In order to investigate these questions, we have performed the following experiment:

1. **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 3.1).
2. **Developed New Case Examples:** We have also mined an additional 240 faults from ten new projects, and added these faults to the Defects4J database (Section 3.1).
3. **Recorded Source Code Metrics:** For each affected class (both faulty and fixed versions), we measure a series of sixty source code metrics, related to cloning, cohesion, coupling, documentation, inheritance, and class size (Section 3.2).
4. **Generated Test Suites:** For each fault, we generated 10 suites per criterion using the fixed version of each CUT. We performed with both a two-minute and a ten-minute search budget per CUT (Section 3.3).
5. **Generated Test Suites for Combinations of Criteria:** We perform the same process for EvoSuite’s default configuration—a combination of eight criteria—and a combination of branch, exception, and method coverage (Section 3.3).

-
6. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are removed (Section 3.3).
 7. **Assessed Fault-finding Effectiveness:** We measure the proportion of test suites that detect each fault to the number generated (Section 3.4).
 8. **Recorded Generation Statistics:** For each suite, fault, and budget, we measure factors that may influence suite effectiveness, related to coverage, suite size, and obligation satisfaction (Section 3.4).
 9. **Prepare Datasets:** Datasets were prepared for learning purposes by adding classifications based on fault detection to each entry in the generation factor and code metric datasets. Separate datasets were prepared for each generation budget and fitness function, as well as sets based on overall fault detection (across all fitness functions and function combinations) for each budget (Section 3.5).
 10. **Performed Treatment Learning:** We apply the TAR3 learner to identify factors correlated to each classification for each dataset (Section 3.5).

3.1 | Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [39]². The original dataset consisted of 357 faults from five projects: Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), and Time (27 faults). For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the fault, and a list of classes and lines of code modified by the patch that fixes the fault.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings.

In order to expand our study to a larger set of case examples, we have added an additional ten systems to Defects4J. The process of adding new faults is semi-automated, and requires the development of build files that work across project versions. The commit messages of the project’s version control system are scanned for references to issue identifiers. These versions are considered to be candidate “fixes” to the referenced issues. The human-developed test suite for that version is then applied to previous project versions. If one or more test cases pass on the “fixed” version and fail on the earlier version, then that version is retained as the “faulty” variant. The code differences between versions are captured as a patch file, which must then be manually minimized to remove any differences that are not required to reproduce the fault.

Following this process, we added 240 faults—bringing the total to 597 faults from 15 projects. The new projects include: CommonsCLI (24 faults), CommonsCSV (12), CommonsCodec (22), CommonsJXPath (14), Guava (9), JacksonCore (13), JacksonDatabind (39), JacksonXML (5), Jsoup (64), and Mockito (38)³. The ten new systems were chosen because they are popular projects and have reached a reasonable level of maturity—meaning that the detected faults are often relatively complicated. Two of these systems, Guava and Mockito, were the subjects of recent research challenges at the Symposium on Search-Based Software Engineering [23, 3]. Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 593 faults that we used in our study.

3.2 | Code Metric-based Characterization

When assessing the results of our study, we wish to gain understanding of *when and why* our test suites were able to detect—or not detect—faults. Such understanding could lead to a deeper understanding of the strengths and limitations of current test generation techniques, and could inspire new approaches. Gaining such understanding requires fine-grained information about the faults being targeted—and, more specifically, the classes being targeted for test generation. To assist in gaining this understanding, we have turned to traditional *source code metrics*. Using the SourceMeter framework⁴, we have gathered a set of 60 cloning, complexity, cohesion, coupling, documentation, inheritance, and size metrics for each class associated with a fault included in the Defects4J dataset.

Such metrics, commonly used as part of research on effort estimation [50] and defect prediction [64], are considered to have substantial predictive power. By characterizing the classes that host the Defects4J faults using these code metrics, we can better understand the results of our research. Using the SourceMeter framework, we have measured 60 source code metrics for each class related to a fault in the Defects4J dataset. These metrics are recorded for both the faulty and fixed versions of each affected class. The metrics may be divided into the following categories:

- **Clone Metrics** are used to measure the occurrence of Type-2 clones in the class—code fragments that are structurally identical, but may differ in variable names, literals, and identifiers [59].

²Available from <http://defects4j.org>

³The new faults have been submitted to Defects4J as a pull request. Until they are accepted, they can be found at the *additional-faults-1.4* branch of <https://github.com/Greg4cr/defects4j>.

⁴Available from <https://www.sourceme.com>.

Category	Abbreviation	Metric	Median	Standard Deviation
Clone	CC	Clone Coverage	0.00	0.16
	CCL	Clone Classes	0.00	10.80
	CCO	Clone Complexity	0.00	453.26
	CI	Clone Instances	0.00	27.86
	CLC	Clone Line Coverage	0.00	0.09
	CLLC	Clone Logical Line Coverage	0.00	0.14
	LDC	Lines of Duplicated Code	0.00	161.06
Cohesion	LLDC	Logical Lines of Duplicated Code	0.00	147.78
	LCOM5	Lack of Cohesion in Methods 5	1.00	7.31
Complexity	NL	Nesting Level	4.00	2.94
	NLE	Nesting Level Else-If	3.00	1.93
	WMC	Weighted Methods per Class	55.00	149.58
Coupling	CBO	Coupling Between Object Classes	8.00	11.79
	CBOI	Coupling Between Object Classes Inverse	5.00	61.38
	NII	Number of Incoming Invocations	16.00	172.67
	NOI	Number of Outgoing Invocations	14.00	31.31
	RFC	Response Set For Class	37.50	56.43
Documentation	AD	API Documentation	1.00	0.32
	CD	Comment Density	0.38	0.18
	CLOC	Comment Lines of Code	127.00	460.47
	DLOC	Documentation Lines of Code	102.50	443.12
	PDA	Public Documented API	7.00	28.31
	PUA	Public Undocumented API	0.00	8.77
	TCD	Total Comment Density	0.36	0.17
Inheritance	TCLOC	Total Comment Lines of Code	144.50	465.29
	DIT	Depth of Inheritance Tree	1.00	1.15
	NOA	Number of Ancestors	1.00	1.71
	NOC	Number of Children	0.00	2.07
	NOD	Number of Descendants	0.00	3.39
Size	NOP	Number of Parents	1.00	0.86
	LLOC	Logical Lines of Code	208.50	462.19
	LOC	Lines of Code	382.50	879.00
	NA	Number of Attributes	8.00	14.33
	NG	Number of Getters	3.00	14.96
	NLA	Number of Local Attributes	6.00	10.24
	NLG	Number of Local Getters	2.00	8.58
	NLM	Number of Local Methods	21.00	35.03
	NLPA	Number of Local Public Attributes	0.00	4.35
	NLPM	Number of Local Public Methods	9.00	29.50
	NLS	Number of Local Setters	0.00	5.23
	NM	Number of Methods	28.50	50.96
	NOS	Number of Statements	109.50	279.94
	NPA	Number of Public Attributes	0.00	6.17
	NPM	Number of Public Methods	14.00	44.25
	NS	Number of Setters	0.00	11.33
	TLLOC	Total Logical Lines of Code	275.50	503.28
	TLOC	Total Lines of Code	495.00	919.51
	TNA	Total Number of Attributes	10.00	17.34
	TNG	Total Number of Getters	4.00	20.96
	TNLA	Total Number of Local Attributes	8.00	14.20
	TNLG	Total Number of Local Getters	2.00	10.62
	TNLM	Total Number of Local Methods	27.00	44.33
	TNLPA	Total Number of Local Public Attributes	0.00	4.67
	TNLPM	Total Number of Local Public Methods	12.00	34.82
	TNLS	Total Number of Local Setters	0.00	6.20
	TNM	Total Number of Methods	37.00	79.02
	TNOS	Total Number of Statements	143.00	293.37
	TNPA	Total Number of Public Attributes	0.00	6.36
	TNPM	Total Number of Public Methods	19.00	62.78
	TNS	Total Number of Setters	0.00	14.01

TABLE 1 List of metrics gathered for each class, separated by category. The median and standard deviation are listed for each.

- **Cohesion Metrics** assess the level of cohesion in a class—whether attributes and the operations that use them are organized into one class, and whether there are methods that are unrelated to the other methods and attributes in the class [33].
- **Complexity Metrics** assess the complexity of the class, using information such as the depth of nesting and the number of control-flow paths through methods [13]. Complexity is often used as part of defect prediction, as more complex methods are expected to contain more defects than simpler methods.

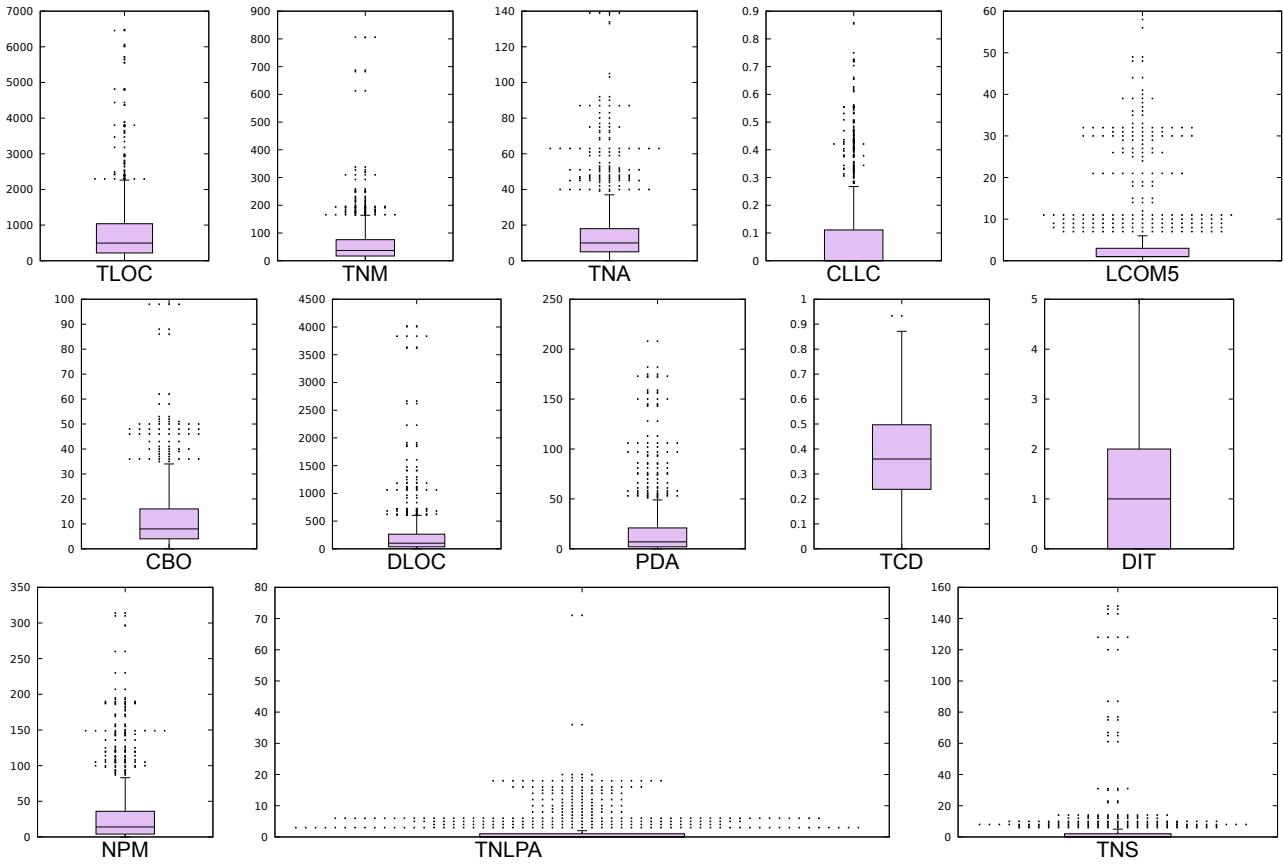


FIGURE 1 Boxplots illustrating the median, first, and third quartile values for select metrics from the dataset.

- **Coupling Metrics** assess the level of dependency between classes [9]. When designing a system, developers are cautioned to minimize the level of coupling—to make each class as independent as possible. Coupling metrics are used to identify design weaknesses and architectural bottlenecks.
- **Documentation Metrics** measure the degree that a class is documented by its developers [42]. Well-documented code is often thought to be higher quality code, and these metrics can identify classes that may have received less attention.
- **Inheritance Metrics** tracks the relationships between parent and child classes along the class hierarchy [9]. As inheritance defines a form of dependence, such metrics are useful for identifying how code changes can propagate through a system.
- **Size Metrics** characterize the size and complexity of a class based on structural elements such as the number of lines of code, methods, attributes, setters, and getters [1]. Such metrics can be used, at a glance, to identify some of the more complex classes in a system.

Table 1 lists the gathered metrics. Detailed definitions may be found on the SourceMeter documentation [63]. Table 1 notes the median and standard deviation for all metrics.

3.2.1 | Characterizing the “Average” Class

To help illustrate the “average” class from Defects4J, we have included boxplots for several of the measured metrics in Figure 1. Each box depicts the first and third quartiles, as well as the median value. Outliers—points more than 1.5 times the interquartile range—are depicted as well. Rather than depict all 60 metrics, we show a subset indicated as important in our case study to help characterize the studied classes. First, to set context:

- **TLOC (Total Lines of Code)** indicates the amount of code in a class, including comments and whitespace. TLOC include lines in anonymous, nested, and local classes. The median TLOC is 495, but a large standard deviation (919.51) indicates that classes have a wide range of sizes. Studied classes tend to cluster between 0-1,000 TLOC, and the largest class has 6,481 TLOC.

-
- **TNM (Total Number of Methods)** is the number of methods in a class, including those in anonymous, nested, and local classes, as well as those inherited from a parent. The median TNM is 37, with the maximum being 806. Classes in Defects4J tend to have less than 100 methods. Again, however, there are a number of outliers.
 - **TNA (Total Number of Attributes)** is the number of attributes in a class, including those in anonymous, nested, and local classes, as well as those inherited from a parent. The median TNA is 10—with values tending to cluster between 5-20—and the maximum is 139.

The following metrics were indicated in our case study as being able to explain why generated test suites are able—or not able—to detect faults. We will discuss the implications of these findings in Section 4.5. Here, we use these metrics to further characterize the “average” class in Defects4J.

- **CLLC (Clone Logical Line Coverage)** is the ratio of code covered by code duplications in the class to the size of the class, expressed in terms of logical lines of code (non-empty, non-comment lines). **Clone Coverage** is the same measurement, except that it includes comments and whitespace. The CLLC and CC are both largely concentrated towards the low end of the scale, with a median of 0 for both—no code being duplicated—and a relatively low standard deviation (0.14).
- **LCOM5 (Lack of Cohesion in Methods 5)** measures the lack of cohesion and computes how many coherent classes the class could be split into. Although there are many outliers, the LCOM5 is largely concentrated towards the low end of the scale—with a median of 1—indicating that classes tend to be highly cohesive.
- **CBO (Coupling Between Objects)** indicates the number of classes that serve as dependencies of the target class (by inheritance, method call, type reference, or attribute reference). Classes dependent on many other classes are very sensitive to the changes in the system, and can be harder to test or evolve. The median CBO is 8 and standard deviation is 11.79, indicating that—while classes are connected—the average class does not overly depend on the rest of the system.
- **DLOC (Documented Lines of Code)** simply measures the number of line of code that are comments (in-line or standalone). The median number of documented lines is 102. However, there are a large range of values, with the maximum DLOC at a staggering 4,017.
- **PDA (Public Documented API)** is the number of public methods with documentation. The median PDA is 7. Again, however, there is a large variance in PDA, with a standard deviation of 28.31 and a large number of outlying values.
- **TCD (Total Comment Density)** is the ratio of the comment lines to the sum of its comment and logical lines of code, including nested, anonymous, and local classes. The **CD (Comment Density)**, also noted as important, is the same measurement excluding nested, anonymous, and local classes. The median TCD is 0.36 and the median CD is 0.38, indicating that the “average” Defects4J class is approximately one-third documentation.
- **DIT (Depth of Inheritance Tree)** measures the length of the path from the class to its furthest ancestor in the inheritance tree. The median DIT is 1—many classes have a parent. The maximum DIT is 5, but this is an extreme outlier—the standard deviation is 1.15 and almost all classes have 0-2 levels of ancestors.
- **NPM (Number of Public Methods)** indicates the number of methods that are publicly-accessible in the class, including inherited methods. Closely related is the **TNLPM (Total Number of Local Public Methods)**, which includes nested, anonymous, and local classes, but excludes inherited methods. The median NPM is 14, with the majority concentrated below 50. The median TNLPM is slightly lower—12. Both are lower than the TNM, indicating that many classes have a large number of private methods.
- **NLPA (Number of Local Public Attributes)**, similarly, indicates the number of publicly-accessible attributes in the class, excluding inherited attributes. The **TNLPA (Total Number of Local Public Attributes)** includes nested, anonymous, and local classes. Both metrics are less consistent than many of the others, with a large number of outlying values. However, the median for both is 0—indicating that there are often no public attributes in a class. This is notably lower than the TNA, indicating that class attributes are generally private.
- **TNS (Total Number of Setters)** records the number of setter methods in the class, including inherited methods. Our study also indicated the **NS (Number of Setters)**—excluding nested, anonymous, and local classes—and **TNLS (Total Number of Local Setters)**—excluding inherited setters—as important. This set of attributes also has a relatively large number of outliers, but is concentrated towards the low end of the scale. All three of these metrics have a median value of 0, indicating that most classes in Defects4J have no setter methods.

3.3 | Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [60]. In this study, we used EvoSuite version 1.0.5, and the following fitness functions:

Branch Coverage (BC): A test suite satisfies branch coverage if all control-flow branches are taken during test execution—the test suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one whose execution evaluates the predicate to `false`. To

guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how “close” the targeted predicate is to being true, using a cost function based on the predicate formula [6].

Direct Branch Coverage (DBC): Branch coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. When a test covers a branch indirectly, it can be more difficult to understand how coverage was attained. Direct branch coverage requires each branch to be covered through a direct method call.

Line Coverage (LC): A test suite satisfies line coverage if it executes each non-comment source code line at least once. To cover each line of source code, EvoSuite tries to ensure that each basic code block is reached. The branch distance is computed for each branch that is a control dependency of any of the statements in the CUT. For each conditional statement that is a control dependency for some other line in the code, EvoSuite requires that the branch of the statement leading to the dependent code is executed.

Exception Coverage (EC): The goal of exception coverage is to build test suites that force the CUT to throw exceptions—either declared or undeclared. As the number of possible exceptions that a class can throw cannot be known ahead of time, the fitness function rewards suites that throw more exceptions. As this function is based on the number of discovered exceptions, the number of “test obligations” may change each time EvoSuite is executed on a CUT.

Method Coverage (MC): Method Coverage simply requires that all methods in the CUT are executed at least once, through direct or indirect calls. The fitness function for method coverage is discrete, as a method is either called or not called.

Method Coverage (Top-Level, No Exception) (MNEC): Generated test suites sometimes achieve high levels of method coverage by calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.

Output Coverage (OC): Output coverage rewards diversity in the method output by mapping return types to a list of abstract values [4]. A test suite satisfies output coverage if, for each public method in the CUT, at least one test yields a concrete return value characterized by each abstract value. For numeric data types, distance functions offer feedback using the difference between the chosen value and target abstract values.

Weak Mutation Coverage (WMC): Test effectiveness is often judged using mutants [38]. Suites that detect more mutants may be effective at detecting real faults as well. A test suite satisfies weak mutation coverage if, for each mutated statement, at least one test detects the mutation. The search is guided by the *infection distance*, a variant of branch distance tuned towards reaching and discovering mutated statements [21].

Rojas et al. provide a primer on each of these fitness functions [58]. In order to study the effect of combining fitness functions, we also generate test suites using two combinations. The first is EvoSuite’s default configuration—a combination of all of the above methods (called the “**Default Combination**”). The second is a combination of branch, exception, and method coverage (called the “**BC-EC-MC Combination**”). This combination was identified as an effective baseline in our prior work studying combination efficacy on the five original systems from Defects4J [25].

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. EvoSuite generates assertion-based oracles. Generating oracles based on the fixed version of the class means that we can confirm that the fault is actually detected, and not just that there are coincidental differences in program output. In practice, this translates to a regression testing scenario, where tests are generated using a version of the system understood to be “correct” in order to guard against future issues. Tests that fail on the faulty version, then, detect behavioral differences between the two versions.

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits each fitness function. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget. This resulted in the generation of 118,600 test suites (two budgets, ten trials, ten configurations, 593 faults).

Generation tools may generate flaky (unstable) tests [60]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, fewer than one test tends to be removed from each suite (see Table 2).

3.4 | Test Generation Data Collection

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **effectiveness** of each fitness function, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault. We refer to this as the likelihood of fault detection.

To better understand the generation factors that influence effectiveness, we also collected the following for each test suite:

Method	Budget	Total Obligations	% Obligations Satisfied	Suite Size	Suite Length	# Tests Removed	% Line Coverage (Fixed)	% Line Coverage (Faulty)	% Branch Coverage (Fixed)	% Branch Coverage (Faulty)
Branch Coverage (BC)	120	295.15	58.32%	36.33	195.96	0.38	61.98%	62.04%	58.96%	58.87%
	600		65.00%	44.27	268.95	0.75	67.35%	67.23%	65.44%	65.19%
Direct Branch (DBC)	120	295.15	54.60%	38.32	217.55	0.35	60.01%	59.59%	56.37%	55.93%
	600		61.79%	48.27	310.52	0.73	65.53%	64.96%	63.32%	62.65%
Exception Coverage (EC)	120	12.47	99.41%	11.99	35.54	0.23	21.35%	21.36%	15.82%	15.99%
	600		99.38%	12.12	36.09	0.26	21.60%	21.63%	15.97%	16.15%
Line Coverage (LC)	120	329.90	61.29%	30.32	162.08	0.28	62.27%	61.69%	53.60%	53.09%
	600		66.79%	34.73	207.65	0.45	67.53%	66.90%	59.11%	58.51%
Method Coverage (MC)	120	31.92	78.99%	22.00	73.78	0.05	37.51%	37.40%	29.18%	29.26%
	600		83.30%	24.32	86.62	0.09	38.91%	38.93%	30.36%	30.52%
Method, No Exception (MNEC)	120	31.92	77.59%	21.68	74.54	0.05	39.06%	38.99%	30.39%	30.48%
	600		82.19%	23.79	88.88	0.10	40.84%	40.78%	31.75%	31.91%
Output Coverage (OC)	120	185.89	42.78%	29.04	133.04	0.12	39.00%	38.82%	32.90%	32.88%
	600		46.17%	32.68	161.32	0.26	40.81%	40.67%	34.47%	34.47%
Weak Mutation (WMC)	120	508.38	56.20%	26.48	164.51	0.16	56.02%	55.87%	49.73%	49.50%
	600		62.94%	32.60	246.57	0.42	61.58%	61.31%	56.19%	55.80%
Default Combination	120	1673.19	53.92%	48.71	358.93	0.55	58.25%	58.05%	53.15%	52.95%
	600		60.72%	64.57	550.03	1.08	64.65%	64.07%	60.91%	60.30%
BC-EC-MC Combination	120	345.59	63.81%	50.73	262.03	1.06	62.91%	62.22%	59.95%	59.12%
	600		69.69%	65.10	368.83	2.04	68.00%	67.12%	66.30%	65.24%

TABLE 2 Statistics on generated test suites (each statistic is explained in Section 3.4). Values are averaged over all faults (i.e., the average number of obligations, average number of tests removed, etc.). “Default Combination” is a combination of the eight individual fitness functions. “BC-EC-MC Combination” combines the branch, exception, and method coverage fitness functions.

Number of Test Obligations: Given a CUT, each fitness function will calculate a series of test obligations to cover (as defined in Section 2). The number of obligations is informative of the difficulty of the generation, and impacts the size and formulation of tests [28]. Note that the number of test obligations is dependent on the CUT, and does not differ between budgets except in the case of exception coverage. As exception coverage simply counts the number of observed exceptions, it does not have a consistent set of obligations each time generation is performed.

Percentage of Obligations Satisfied: This factor indicates the ability of a fitness function to cover its goals. A suite that covers 10% of its goals is likely to be less effective than one that achieves 100% coverage.

Test Suite Size: We have recorded the number of tests in each test suite. Larger suites are often thought to be more effective [30, 36]. Even if two suites achieve the same coverage, the larger may be more effective simply because it exercises more combinations of input.

Test Suite Length: Each test consists of one or more method calls. Even if two suites have the same number of tests, one may have much *longer* tests—making more method calls. In assessing the effect of suite size, we must also consider the length of each test case.

Number of Tests Removed: Any tests that do not compile, or that return inconsistent results, are automatically removed. We track the number removed from each suite.

Code Coverage: As the premise of many adequacy criteria is that faults are more likely to be detected if structural elements of the code are thoroughly executed, the resulting coverage of the code may indicate the effectiveness of a test suite. Using EvoSuite’s coverage measurement capabilities, we have measured the line and branch coverage achieved by each suite when executed over both the faulty and fixed versions of each CUT. Due to instrumentation issues, we were unable to measure coverage over two systems—JacksonDatabind and Mockito. However, we were able to measure coverage over the remaining 516 faults.

Table 2 records, for each fitness function and budget, the average values attained for each of these measurements over all faults for which we were able to take measurements. In Figure 2, we show boxplots of the total obligations and % of obligations satisfied for suites generated for each fitness configuration and search budget. Branch and direct branch coverage will always have the same number of obligations. Line coverage tends to operate in the same approximate range. Exception, method, and MNEC have the fewest obligations, while weak mutation coverage tends to have the most obligations of the individual fitness functions. Naturally, the two combinations have more obligations—the combination of their member functions. In terms of satisfaction, the three fitness functions with the fewest obligations—exception, method, and MNEC—all also have the highest satisfaction rate. Output coverage has the lowest average, and generally lower, satisfaction rates. For all functions other than Exception Coverage, there tends to be large variance in the satisfaction rate.

In Figure 3, we show boxplots of the suite size, length, and line coverage of suites generated for each fitness configuration and search budget. Most fitness functions yield similar median suite size and variance in results. Exception coverage tends to yield the smallest suites, owing to its small number of test obligations. Method coverage and MNEC yield smaller suites than other fitness functions, but not significantly so. Output coverage

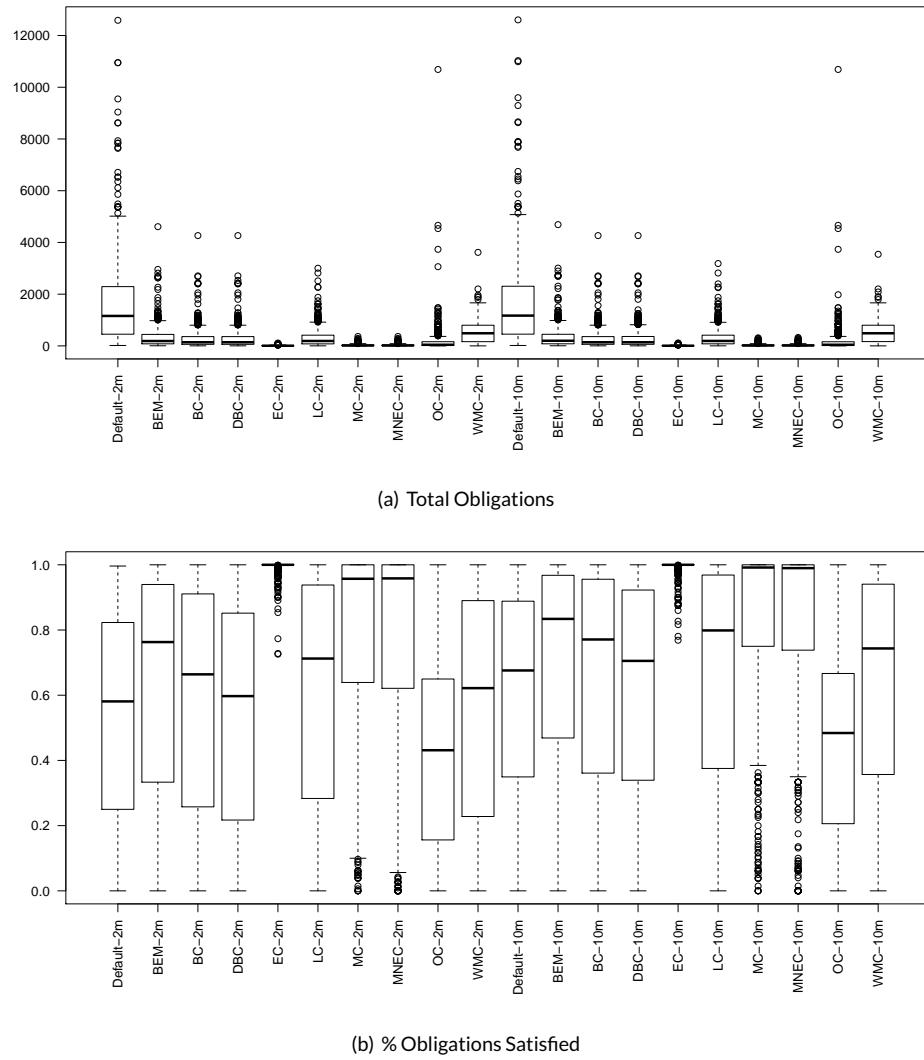


FIGURE 2 Boxplots of the total obligations and % of obligations satisfied for suites generated for each fitness configuration and search budget.

tends to have relatively large test suites—comparable in size to branch and direct branch coverage. Again, the two combinations tend to yield larger test suites, but not significantly larger than those for branch, direct branch, and weak mutation coverage. Test suite length largely offers similar observations. However, we do note that the “default combination” tends to yield very *long* test suites, composed of more method calls than suites for other fitness configurations. This is not the case for the BC-EC-MC combination.

Like with obligation satisfaction, there is a large variance in the line coverage attained by test suites, regardless of fitness function. Exception coverage tends to achieve both the lowest coverage and the least variance in coverage. This is reasonable, as the fitness function for exception coverage has no mechanism to encourage class exploration. Naturally, branch, direct branch, line, and weak mutation coverage tend to attain high coverage rates over classes, as all four use coverage-based fitness mechanisms. Exception, method, MNEC, and output coverage are based on source code elements, but all have fitness representations that are not based on control flow. As a result, they tend to attain lower coverage levels.

3.5 | Dataset Preparation for Treatment Learning

To understand the factors leading to detection—or lack of detection—of a fault, we have collected two basic sets of data for each fault: **test generation factors** related to the test suites produced and **source code metrics** examining the classes being targeted for unit test generation.

A standard practice in machine learning is to *classify data*—to use previous experience to categorize new observations [49]. We are instead interested in the reverse scenario. Rather than attempting to categorize new data, we want to work backwards from classifications to discover which

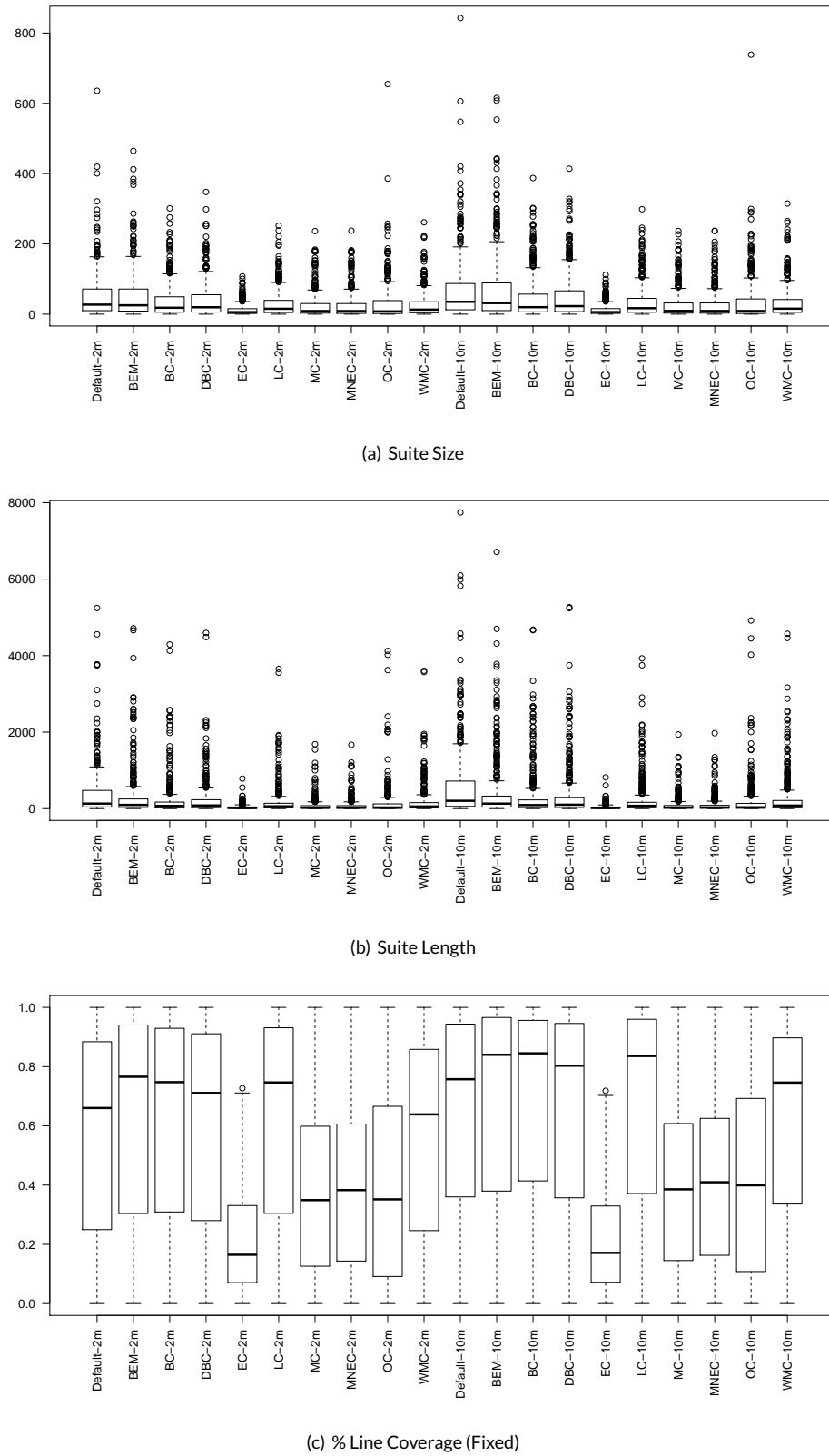


FIGURE 3 Boxplots of the suite size, length, and line coverage of suites generated for each fitness configuration and search budget.

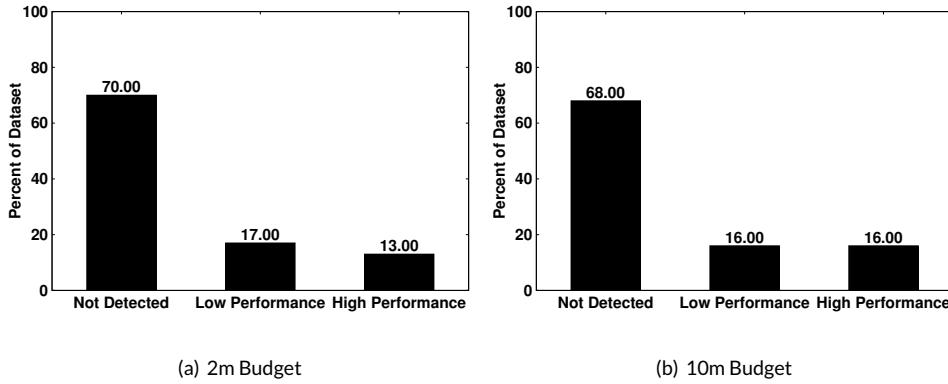


FIGURE 4 Baseline class distribution of the generation factors datasets used for treatment learning.

factors correspond most strongly to a class of interest—a process known as treatment learning [47]. Treatment learning approaches take the classification of an observation and reverse engineers the evidence that led to that categorization. Such learners produce a *treatment*—a small set of attributes and value ranges for each that, if imposed, will identify a subset of the original data skewed towards the target classification. In this case, a treatment notes the metrics—and their values—that indicate that generated test suites will detect a fault.

For example, the treatment [$NS = [0.00..1.00]$, $TCD = [0.52..0.93]$]—derived from the code metric-based dataset—indicates that the subset of the data where the Number of Setters is less than 1 and the Total Comment Density is between 52–93% has a higher percentage of “Yes” classifications (“Yes” implying that the fault is detected, while a “No” classification indicates that the fault was not detected) than the base dataset. Within this subset, “Yes” classifications account for 72.00% of the class distribution of the subset—compared to 47.44% of the base distribution.

Using the TAR3 treatment learner [27], we have generated five treatments from each dataset that target each of the applied verdicts. A user can specify the minimum number of examples that may make up the data subset matching a treatment in order to ensure minimum support for a treatment. We require the subset to contain at least 20% of the total dataset. In addition, a limit can be placed on the number of attributes chosen for a treatment. It is thought that large treatments—those recommending more than five attribute-range pairs—may not have more explanatory power than smaller treatments [27]. Therefore, we also limit the treatment size to five attribute-value pairings.

3.5.1 | Generation-Based Datasets

As discussed in Section 3.4—to better understand the combination of factors correlating with effective fault detection—we have collected the following statistics for each generated test suite: the number of obligations, the percent satisfied, suite size, suite length, number of tests removed, as well as branch (BC) and line coverage (LC) over the fixed and faulty versions of the CUT.

This collection of factors forms a dataset that can be used to analyze the impact of these factors on efficacy. Each record in the dataset corresponds to the values of these attributes for each fault, and for each criterion or combination of criteria. In total, this dataset contains 5160 records (516 faults for which we could collect all statistics, times the ten fitness configurations). We build separate datasets for each search budget. We can then use the likelihood of fault detection (D) as the basis for a class variable—discretized into three values: “not detected” ($D = 0$), “low performance” ($D < 70\%$), and “high performance” ($D \geq 70\%$). The class distribution of each dataset is shown in Figure 4.

3.5.2 | Code Metric-Based Datasets

As with the test generation factors, we have created separate source code metric datasets split by the search budget used for test generation (two minutes per class or ten minutes per class). Source code metrics do not differ based on the fitness function used in test generation. Therefore, rather than merging all fitness configurations into a single dataset, we have produced separate datasets for each fitness function. We also produced “overall” datasets, classified by whether *any* fitness function or combination detected a fault. In total, this process produced a set of 18 datasets for treatment learning: two based on overall results—split by search budget—and two for each of the eight studied coverage criteria.

In order to learn which metrics predict whether or not a fault is detected, we have added classifications to each characterization dataset. In this case, we have used two classification values—yes or no, based on whether or not the fault was detected by the generated test suites⁵ For each fault,

⁵Initially, we used a three-option classification like with the generation factors dataset. However, this did not yield enough examples to yield detailed treatments in many cases. Instead, we elected to use a two-class outcome.

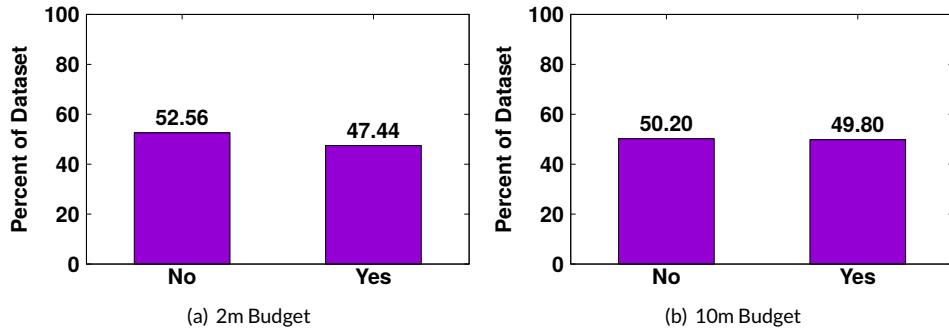


FIGURE 5 Baseline class distributions for the “overall” code metrics datasets used for treatment learning.

the characterization dataset has an entry for each class fixed as part of patching the fault. We apply a “yes” classification if *any* test suite generated targeting that class, under that search budget, detects the fault. If no test suites detected that fault under that search budget, we apply a “no” classification. The class distributions for the two overall datasets are shown in Figure 5. In both cases, the two classes each make up roughly half of the dataset, with a slight edge to the number of “no” classifications.

3.5.3 | Accessing Datasets

We have made all datasets used in this study openly available as community resources. They can be downloaded from:

<https://github.com/Greg4cr/coverage-exp-data>

4 | RESULTS & DISCUSSION

In Section 4.1, we will outline the basic fault detection capabilities of the generated test suites. Section 4.2 examines the efficacy of each individual fitness function, while Section 4.3 outlines the effect of combining fitness functions. In Section 4.4, we examine the generation factors contributing to fault detection. Finally, in Section 4.5, we examine the source code metrics that impact detection efficacy.

4.1 | Overall Fault-Detection Capability

In Table 3, we list the number of faults detected by each fitness function, broken down by system and search budget. We also list the number of faults detected by *any criterion*, including and excluding the combinations (which we will discuss further in Section 4.3). Due to the stochastic search, a higher budget does not guarantee detection of the same faults found under a lower search budget. Therefore, we list the number of faults found under either budget, as well as the total number of faults detected by each fitness function. These results offer a baseline for further discussion. In our experiments:

The individual criteria are capable of detecting 304 (51.26%) of the 593 faults. Combinations detect a further 17 faults.

While there is clearly room for improvement, these results are encouraging. The studied faults are actual faults, reported by the users of real-world software projects. The generated tests are able to detect a variety of complex programming issues. Ultimately, our results are consistent with previous studies involving Defects4J—for instance, Shamshiri et al. found that a combination of test generation tools—including suites generated using EvoSuite’s branch fitness function—could detect 55.7% of the faults from the original five systems from Defects4J [60].

Shamshiri’s work—as well as our studies on Guava [3] and Mockito [23]—offer explanation of the broad reasons why test generation fails to detect particular faults. Some of these reasons include a general inability to gain coverage—particularly over private methods—challenges with initialization of complex data types, and a general lack of the context needed to set up sophisticated series of method and class interactions.

In this work, we are focused on the capabilities and applicability of common fitness functions. In the following subsections, we will assess the results of our study with respect to each research question. In Section 4.2, we compare the capabilities of each fitness function. In Section 4.3, we

	Budget	Chart	Closure	CommonsCLI	CommonsCodec	CommonsCSV	CommonsJXPath	Guava	JacksonCore	JacksonDatabind	JacksonXML	Jsoup	Lang	Math	Mockito	Time	Total
BC	2min	17	16	10	12	10	8	3	10	8	3	21	36	53	4	16	227
	10min	20	19	12	12	10	7	2	9	8	3	23	35	54	4	17	235
	Total	21	21	13	13	11	9	3	10	9	3	25	41	57	4	17	257
DBC	2min	14	16	12	12	11	5	2	9	8	1	20	32	48	4	15	209
	10min	19	19	11	13	10	6	2	10	10	2	22	36	47	4	18	229
	Total	19	22	13	13	11	7	2	10	10	2	26	40	52	4	18	249
EC	2min	8	7	5	4	2	0	0	5	4	1	9	12	13	3	6	79
	10min	10	5	4	3	2	2	0	6	3	1	8	13	12	3	5	77
	Total	10	8	6	4	2	2	0	6	4	1	9	16	15	3	6	92
LC	2min	15	12	7	11	11	4	2	8	9	3	14	31	50	3	15	196
	10min	18	14	10	11	9	8	2	8	10	3	23	32	52	3	14	217
	Total	18	17	11	13	11	8	2	8	11	3	24	37	55	3	15	236
MC	2min	10	6	5	7	5	0	1	5	2	1	7	9	25	2	5	90
	10min	10	10	4	6	4	0	2	4	2	1	7	11	24	2	5	92
	Total	12	10	6	7	5	0	2	5	2	1	8	14	27	2	6	107
MNEC	2min	9	8	5	7	3	1	2	5	5	1	6	10	29	2	5	98
	10min	11	6	2	5	4	2	2	4	4	0	9	13	27	2	3	94
	Total	11	9	5	7	4	2	2	5	5	1	9	13	21	2	6	113
OC	2min	9	7	6	8	2	1	2	4	5	2	7	13	36	2	5	109
	10min	13	9	5	9	2	2	2	3	4	2	8	17	33	2	6	117
	Total	13	12	6	9	2	2	2	4	5	2	9	18	38	2	8	132
WMC	2min	13	15	6	13	8	4	2	10	7	3	19	31	42	3	14	190
	10min	18	19	9	15	8	6	3	9	7	2	22	32	48	3	14	215
	Total	18	22	9	15	8	6	3	10	9	3	25	37	51	3	15	234
Default	2min	15	17	11	14	10	4	2	9	9	1	17	29	48	5	14	205
	10min	17	20	11	14	11	7	2	10	11	1	22	35	57	5	14	237
	Total	18	22	11	14	11	7	2	10	11	1	23	36	59	5	15	245
BC-EC-MC	2min	16	19	10	13	11	5	2	10	9	1	21	38	53	4	15	227
	10min	20	30	12	12	11	6	3	10	9	1	27	40	55	4	21	261
	Total	20	31	12	13	11	6	3	11	10	1	29	41	56	4	21	269
(no combinations)	2min	18	23	15	14	11	9	3	10	10	3	31	43	61	5	16	272
	10min	23	29	15	17	11	10	3	10	11	3	33	45	59	5	18	292
	Total	23	31	16	17	11	10	3	10	12	3	37	46	62	5	18	304
(w. combinations)	2min	18	24	15	15	11	10	3	10	10	3	33	44	62	5	16	279
	10min	23	37	15	17	11	10	3	11	12	3	35	46	64	5	21	313
	Total	23	37	16	17	11	10	3	11	13	3	39	47	65	5	21	321

TABLE 3 Number of faults detected by each fitness function. Totals are out of 26 faults (Chart), 133 (Closure), 24 (CommonsCLI), 22 (CommonsCodec), 12 (CommonsCSV), 14 (CommonsJXPath), 9 (Guava), 13 (JacksonCore), 39 (JacksonDatabind), 5 (JacksonXML), 64 (Jsoup), 65 (Lang), 102 (Math), 38 (Mockito), 27 (Time), and 593 (Overall).

explore combinations of criteria. In Section 4.4, we explore the generation factors that indicate efficacy or lack of efficacy. Finally, in Section 4.5, we explore the source code metrics that indicate efficacy or lack of efficacy.

4.2 | Comparing Fitness Functions

From Table 3, we can see that suites differ in effectiveness between criteria. Overall, branch coverage outperforms the other criteria, detecting 257 faults. Branch is closely followed by direct branch (249 faults), line coverage (236), and weak mutation coverage (234). These four fitness functions are trailed by the other four, with exception coverage showing the weakest results (92 faults). These rankings do not differ much on a per-system basis. At times, ranks may shift—for example, direct branch coverage occasionally outperforms branch coverage—but we can see two clusters form among the fitness functions. The first cluster contains branch, direct branch, line, and weak mutation coverage—with branch and direct branch leading the other two. The second cluster contains exception, method, method (no-exception), and output coverage—with output coverage producing the best results and exception coverage producing the worst.

Due to the stochastic nature of the search, one suite generated by EvoSuite may not always detect a fault detected by another suite—even if the same criterion is used. To more clearly understand the effectiveness of each fitness function, we must not track only whether a fault was detected, but how *reliably* it is detected. We are interested in the likelihood of detection—if a fresh suite is generated, how likely is it to detect a particular fault. To measure the likelihood, we record the proportion of detecting suites to the total number of suites generated for that fault. The average likelihood of fault detection is listed for each criterion, by system and budget, in Table 4. Figure 6 shows boxplots of the likelihood of detection for each fitness function and combination of functions.

We largely observe the same trends as above. Branch coverage has the highest overall likelihood of fault detection, with 22.60% of suites detecting faults given a two-minute search budget and 25.24% of suites detecting faults given a ten-minute budget. Direct branch coverage and line coverage follow with a 20.62-23.88% and 19.87-22.24% success rate, respectively. While the effectiveness of each criterion varies between system—direct branch outperforms all other criteria for Closure, for example—the two clusters noted above remain intact. Branch, line, direct branch, and weak mutation coverage all perform well, with the edge generally going to branch coverage. On the lower side of the scale, output, method, method (no exception), and exception coverage perform similarly, with a slight edge to output coverage.

	Budget	Chart	Closure	CommonsCLI	CommonsCodec	CommonsCSV	CommonsXPath	Guava	JacksonCore	JacksonDatabind	JacksonXML	Jsoup	Lang	Math	Mockito	Time	Overall
BC	2min	45.00 %	4.66 %	26.67 %	33.18 %	57.50 %	25.00 %	17.78 %	50.77 %	14.62 %	60.00 %	18.13 %	34.00 %	27.94 %	9.21 %	34.81 %	22.60 %
	10min	48.46 %	5.79 %	28.33 %	31.36 %	60.83 %	25.00 %	20.00 %	60.00 %	13.08 %	60.00 %	21.72 %	40.15 %	32.75 %	8.42 %	39.26 %	25.24 %
	% Change	7.69 %	24.19 %	6.25 %	-5.48 %	5.80 %	0.00 %	12.50 %	18.18 %	-10.53 %	0.00 %	19.83 %	18.10 %	17.19 %	-8.57 %	12.77 %	11.72 %
DBC	2min	34.23 %	5.11 %	27.50 %	35.91 %	60.00 %	18.57 %	20.00 %	55.38 %	14.62 %	20.00 %	16.56 %	30.00 %	24.51 %	8.16 %	31.11 %	20.62 %
	10min	40.77 %	6.09 %	26.67 %	36.82 %	65.83 %	25.71 %	17.78 %	54.62 %	14.36 %	22.00 %	20.31 %	38.77 %	28.63 %	8.42 %	40.37 %	23.88 %
	% Change	19.10 %	19.12 %	-3.03 %	2.53 %	9.72 %	38.46 %	-11.11 %	-1.39 %	-1.75 %	10.00 %	22.64 %	29.23 %	16.80 %	3.23 %	29.76 %	15.78 %
EC	2min	22.31 %	1.35 %	5.83 %	12.27 %	16.67 %	0.00 %	0.00 %	20.00 %	3.33 %	20.00 %	6.41 %	7.54 %	6.37 %	6.05 %	9.26 %	6.56 %
	10min	21.54 %	0.98 %	5.83 %	12.27 %	16.67 %	1.43 %	0.00 %	22.31 %	1.79 %	20.00 %	5.94 %	9.23 %	7.06 %	5.26 %	9.63 %	6.64 %
	% Change	-3.45 %	-27.78 %	0.00 %	0.00 %	0.00 %	-%	-%	11.54 %	-46.15 %	0.00 %	-7.32 %	22.45 %	10.77 %	-13.04 %	4.00 %	1.29 %
LC	2min	38.85 %	4.14 %	21.25 %	22.73 %	59.17 %	20.71 %	21.11 %	51.54 %	15.13 %	60.00 %	12.03 %	31.23 %	25.78 %	5.79 %	30.00 %	19.87 %
	10min	46.15 %	4.81 %	22.08 %	21.36 %	50.00 %	25.71 %	20.00 %	53.08 %	17.95 %	56.00 %	17.50 %	34.31 %	29.22 %	5.79 %	36.67 %	22.24 %
	% Change	18.81 %	16.36 %	3.92 %	-6.00 %	-15.49 %	24.14 %	-5.26 %	2.99 %	18.64 %	-6.67 %	45.45 %	9.85 %	13.31 %	0.00 %	22.22 %	11.97 %
MC	2min	30.77 %	1.58 %	9.17 %	16.36 %	18.33 %	0.00 %	11.11 %	25.38 %	0.77 %	4.00 %	7.34 %	7.54 %	10.98 %	0.53 %	8.15 %	7.77 %
	10min	30.77 %	2.26 %	7.92 %	12.27 %	16.67 %	0.00 %	12.22 %	23.85 %	1.03 %	8.00 %	6.56 %	7.69 %	10.88 %	1.58 %	8.15 %	7.71 %
	% Change	0.00 %	42.86 %	-13.64 %	-25.00 %	-9.09 %	-%	10.00 %	-6.06 %	33.33 %	100.00 %	-10.64 %	20.04 %	-0.89 %	200.00 %	0.00 %	-0.87 %
MNEC	2min	23.46 %	2.18 %	9.17 %	14.55 %	12.50 %	0.71 %	11.11 %	20.00 %	4.10 %	2.00 %	7.50 %	6.62 %	12.16 %	1.05 %	6.67 %	7.59 %
	10min	30.77 %	1.88 %	7.50 %	15.00 %	12.50 %	4.29 %	12.22 %	24.62 %	5.90 %	0.00 %	7.50 %	7.54 %	12.06 %	0.79 %	5.19 %	8.09 %
	% Change	31.15 %	-13.79 %	-18.18 %	3.12 %	0.00 %	500.00 %	10.00 %	23.08 %	43.75 %	-100.00 %	0.00 %	13.95 %	-0.81 %	-25.00 %	-22.22 %	6.67 %
OC	2min	1.15 %	2.03 %	14.17 %	24.55 %	10.00 %	0.71 %	14.44 %	8.46 %	7.95 %	40.00 %	5.31 %	7.85 %	16.57 %	3.68 %	9.63 %	9.31 %
	10min	23.85 %	2.56 %	16.25 %	25.00 %	10.83 %	2.14 %	18.89 %	13.08 %	7.69 %	40.00 %	5.00 %	10.92 %	16.76 %	2.89 %	12.22 %	10.25 %
	% Change	12.73 %	25.93 %	14.71 %	1.85 %	8.33 %	200.00 %	30.77 %	54.55 %	-3.23 %	0.00 %	-5.88 %	39.22 %	1.18 %	-21.43 %	26.92 %	10.14 %
WMC	2min	38.08 %	4.44 %	19.17 %	31.36 %	41.67 %	15.00 %	16.67 %	44.62 %	8.97 %	26.00 %	15.00 %	24.15 %	23.04 %	5.79 %	25.19 %	17.59 %
	10min	46.15 %	5.56 %	20.00 %	33.64 %	45.00 %	17.86 %	18.89 %	42.31 %	7.69 %	18.00 %	18.28 %	32.15 %	27.45 %	5.53 %	27.04 %	20.34 %
	% Change	21.21 %	25.42 %	4.35 %	7.25 %	8.00 %	19.05 %	13.33 %	-5.17 %	-14.29 %	-30.77 %	21.88 %	33.12 %	19.15 %	-4.55 %	7.35 %	15.63 %
Default	2min	47.31 %	4.51 %	32.08 %	44.09 %	52.50 %	16.43 %	17.78 %	47.69 %	14.10 %	20.00 %	15.63 %	23.85 %	25.78 %	11.84 %	25.93 %	20.56 %
	10min	48.08 %	7.07 %	34.58 %	45.91 %	50.00 %	21.43 %	20.00 %	52.31 %	16.15 %	20.00 %	20.94 %	32.62 %	32.84 %	10.79 %	33.33 %	24.69 %
	% Change	1.63 %	56.67 %	7.79 %	4.12 %	-4.76 %	30.43 %	12.50 %	9.68 %	14.55 %	0.00 %	34.00 %	36.77 %	27.38 %	-8.89 %	28.57 %	20.10 %
BC-EC-MC	2min	43.08 %	5.64 %	30.42 %	37.27 %	60.00 %	20.71 %	17.78 %	50.00 %	14.87 %	20.00 %	19.38 %	40.46 %	30.39 %	10.26 %	35.93 %	24.03 %
	10min	53.85 %	8.05 %	30.83 %	38.18 %	50.00 %	26.43 %	21.11 %	56.15 %	14.62 %	20.00 %	25.00 %	48.15 %	34.31 %	10.53 %	47.04 %	27.84 %
	% Change	25.00 %	42.67 %	1.37 %	2.44 %	-16.67 %	27.59 %	18.75 %	12.31 %	-1.72 %	0.00 %	29.03 %	19.01 %	12.90 %	2.56 %	30.93 %	15.86 %

TABLE 4 Average likelihood of fault detection, broken down by fitness function, budget, and system. % Change indicates the average gain or loss in efficacy when moving from a two-minute to a ten-minute budget.

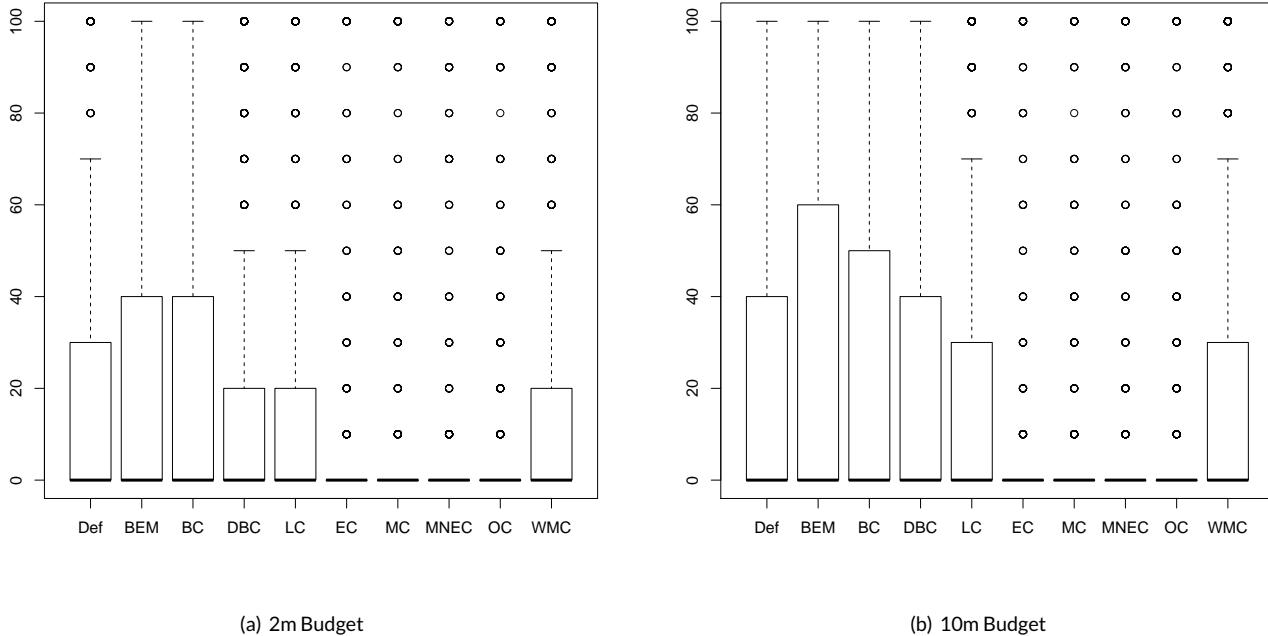


FIGURE 6 Boxplots of the likelihood of detection for each fitness function and combination. “Def” = Default Combination, “BEM” = BC-EC-MC Combination.

The boxplots in Figure 6 echo these results. All methods have medians near 0%, reflecting that many faults are not detected. However, for both budgets, branch coverage has a larger upper quartile than other fitness functions—indicating a large variance in results, but also that branch coverage yields more suites with a higher likelihood of detection than other methods. At the two-minute mark, it has a higher upper whisker as well. Direct branch coverage, line coverage, and weak mutation coverage follow in terms of third quartile size and upper whisker.

	Default Combination	BC	DBC	EC	LC	MC	MNEC	OC	WMC
BC	0.87	-	-	-	-	-	-	-	-
DBC	1.00	0.97	-	-	-	-	-	-	-
EC	< 0.01	< 0.01	< 0.01	-	-	-	-	-	-
LC	1.00	0.59	1.00	< 0.01	-	-	-	-	-
MC	< 0.01	< 0.01	< 0.01	1.00	< 0.01	-	-	-	-
MNEC	< 0.01	< 0.01	< 0.01	1.00	< 0.01	1.00	-	-	-
OC	< 0.01	< 0.01	< 0.01	0.77	< 0.01	0.96	0.94	-	-
WMC	0.91	0.07	0.75	< 0.01	0.99	< 0.001	< 0.01	< 0.01	-
BC-EC-MC Combination	0.56	0.99	0.79	< 0.01	0.27	< 0.01	< 0.01	< 0.01	0.02

TABLE 5 P-values for Nemenyi comparisons of fitness functions (two-minute search budget). Cases where we can reject the null hypothesis are bolded.

Branch coverage is the most effective criterion, detecting 257 faults. Branch coverage suites have, on average, a 22.60-25.24% likelihood of fault detection (2min/10min budget).

From Table 4, we can see that almost all criteria benefit from an increased search budget. Direct branch coverage and weak mutation benefit the most, with average improvements of 15.78% and 15.63% in effectiveness. In particular, it is reasonable that direct branch coverage benefits more than traditional branch coverage. In traditional branch coverage, branches executed through indirect chains of calls to program methods contribute to the total coverage. In direct branch coverage, only calls made directly by the test cases count towards coverage. Therefore, the test generator requires more time, and more method calls, to attain the same level of branch coverage. As a result, direct branch coverage attains slightly worse results than traditional branch coverage given the same time budget.

In general, all distance-driven criteria—branch, direct branch, line, weak mutation, and, partially, output coverage—improve given more time. These criteria all have complex, informative fitness functions that are able to guide the search process. Discrete fitness functions, such as those used by method coverage or exception coverage, benefit less from the budget increase. In such cases, the fitness function is unable to guide the search towards better solutions. More time is of benefit—as the generator can make more guesses. However, such time is not guaranteed to be beneficial, and does not necessarily result in improved test suites.

Distance-based functions benefit from an increased search budget, particularly direct branch and weak mutation.

Further increases in generation time beyond ten minutes may yield further improvements in the likelihood of fault detection. However, there is likely to be a plateau due to test obligations that the generator cannot satisfy, due to limitations in coverage of private code or manipulation of complex objects. Further, if test generation requires more time than it takes for a human to write tests, then the benefits of automation are more limited. Therefore, there is likely to be a limit to the gain from increasing the search budget.

We can perform statistical analysis to assess our observations. For each pair of criteria, we formulate hypothesis H and its null hypothesis, H0:

- H: Given a fixed search budget, test suites generated using criterion A will have a different distribution of likelihood of fault detection results than suites generated using criterion B.
- H0: Observations of fault detection likelihood for both criteria are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H0 without any assumptions on distribution, we use the Friedman non-parametric alternative to the parametric repeated measures ANOVA [?]. Due to the limited number of faults for several systems, we have analyzed results across the combination of all systems. We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

At both budgets, the Friedman test confirms with p-value < 0.001 that the results for all fitness functions are not drawn from the same distribution. To differentiate and rank methods, we apply the post-hoc Nemenyi test in order to assess all pairs of fitness functions. The resulting p-values are listed in Tables 5-6.

	Default Combination	BC	DBC	EC	LC	MC	MNEC	OC	WMC
BC	1.00	-	-	-	-	-	-	-	-
DBC	1.00	1.00	-	-	-	-	-	-	-
EC	< 0.01	< 0.01	< 0.01	-	-	-	-	-	-
LC	0.95	0.71	1.00	< 0.01	-	-	-	-	-
MC	< 0.01	< 0.01	< 0.01	1.00	< 0.01	-	-	-	-
MNEC	< 0.01	< 0.01	< 0.01	1.00	< 0.01	1.00	-	-	-
OC	< 0.01	< 0.01	< 0.01	0.50	< 0.01	0.90	0.95	-	-
WMC	0.73	0.38	0.92	< 0.01	1.00	< 0.01	< 0.01	< 0.01	-
BC-EC-MC Combination	0.47	0.81	0.23	< 0.01	0.02	< 0.01	< 0.01	< 0.01	< 0.01

TABLE 6 P-values for Nemenyi comparisons of fitness functions (ten-minute search budget). Cases where we can reject the null hypothesis are **bolded**.

Function	Two-Minute Budget		Ten-Minute Budget		All Budgets	
	Number of Faults (W. Combinations)	Number of Faults (No Combinations)	Number of Faults (W. Combinations)	Number of Faults (No Combinations)	Number of Faults (W. Combinations)	Number of Faults (No Combinations)
Branch Coverage	5	13	2	7	0	1
Direct Branch Coverage	4	6	3	6	0	1
Exception Coverage	3	4	3	5	1	2
Line Coverage	2	5	3	4	0	0
Method Coverage	0	0	0	1	0	0
Method, No Exception	0	1	1	1	0	0
Output Coverage	1	3	2	7	1	2
Weak Mutation Coverage	3	6	4	7	0	0
Default Combination	2	-	5	-	0	-
BC-EC-MC Combination	4	-	13	-	1	-

TABLE 7 Number of faults uniquely detected by each suites generated using each fitness function (with and without considering combinations) for each budget, and for the combination of budgets.

The results of these tests further validate the “two clusters” observation. For the four criteria in the top cluster—branch, direct branch, line, and weak mutation coverage—we can always reject the null hypothesis with regard to the remaining four criteria in the bottom cluster. This is also true in the opposite direction. The performance of the four criteria in the bottom cluster—exception, method, MNEC, and output coverage—is drawn from a different distribution to the criteria in the other cluster. Within each cluster, we usually fail to reject the null hypothesis.

Branch, direct branch, line, and weak mutation coverage outperform, with statistical significance, method, MNE, output, and exception coverage (both budgets).

Another way to consider performance is—regardless of overall performance—to look at whether a criterion leads to suites that detect faults that other criteria would not detect. Table 7 depicts the number of faults uniquely detected by each fitness function for each search budget, then the number of faults uniquely detected regardless of budget. Results are listed when combinations are considered, which we will discuss in Section 4.3, and when only considering the individual criteria.

From these results, we can see that almost all criteria clearly have *situational applicability*—that is, there are situations where their use leads to the detection of faults missed by other criteria. At both search budgets, a total of 38 faults are detected by a single criterion. Most interestingly, there are six faults that are—regardless of search budget—only detected by a single criterion.

The general efficacy of branch and direct branch coverage clearly can still be seen here, where each detects one fault that nothing else can detect, regardless of budget. However, criteria like exception coverage or output coverage—which have low average performance—can also detect faults that no other criterion can expose. Both criteria detect two faults, regardless of budget, that nothing else can catch. At each independent budget

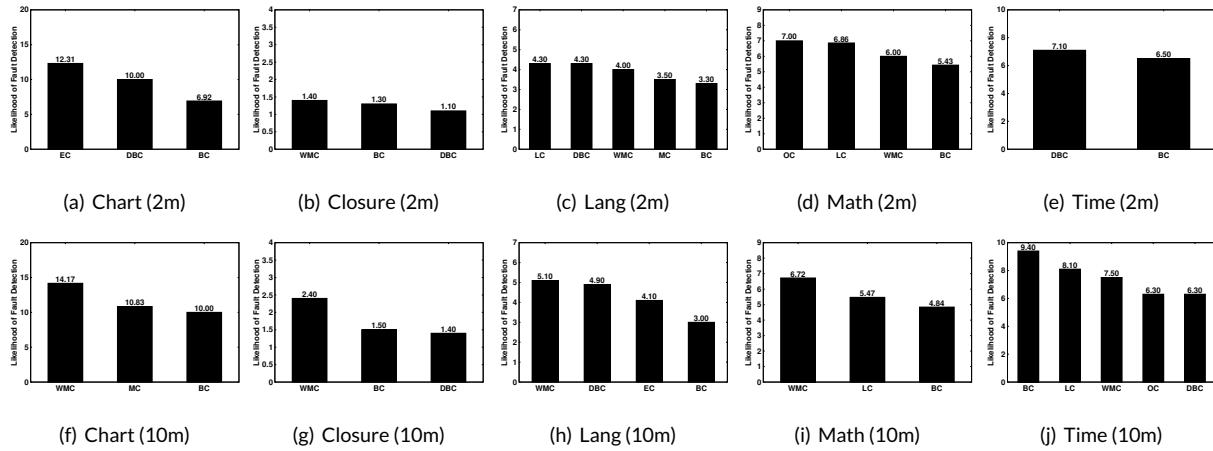


FIGURE 7 Average % likelihood of fault detection for fitness functions once data is filtered for faults where the most effective function for that system has < 30% chance of detection.

level, exception, output, and weak mutation coverage each detect a number of faults that other criteria miss. This is especially worth noting at the ten-minute budget level, where suites generated to satisfy output and weak mutation coverage detect as many unique faults as suites generated to satisfy branch coverage.

These results suggest that—regardless of absolute efficacy—each criterion results in different test suites, each of which exercise the code under test in a distinct manner. Even if a criterion is not universally effective, it offers some form of situational applicability where it could be considered for use, and where it may have some value as part of a portfolio of testing tools.

For example, consider fault 100 for the Math project⁶. To address this fault, an estimation method switches from getting all parameters to only getting unbound parameters. Tests generated for exception coverage cause an exception by passing in a parameter with no measurements—i.e., an unbound parameter. This exception, even if triggered, is not retained by any other fitness function. Exception coverage, by prioritizing exceptions, ensures that the observed failure is retained and passed along to testers.

Output coverage is focused on coverage of abstract value classes for particular types of function output. It is particularly well-defined for numeric types. This makes it well-suited to discovering faults related to such numeric data types. Consider Mockito fault 26⁷. This fault lies in the Mockito framework’s code for replicating Java’s primitive datatypes. Illegal casts can be made from `integer` to other primitive types. Output coverage is able to trigger this fault by illegally casting `integer` variables to `double` variables. This contextual use of the class is not suggested by code coverage, and is not attempted by any of the other criteria.

To further understand the situational applicability of criteria, we filter the set of faults for those that the top-scoring criterion is ineffective at detecting. In Figure 7, we have taken the faults for five of the systems (Chart, Closure, Lang, Math, and Time), isolated any where the “best” criterion for that system (generally branch coverage, see Table 4) has < 30% likelihood of detection, and calculated the likelihood of fault detection for each criterion for that subset of the faults. In each subplot, we display the average likelihood of fault detection over the subset for any criterion that outperforms the best from the full set.

From these plots, we can see that there are always two-to-four criteria that are more effective in these situations. The exact criteria depend strongly on the system, and likely, on the types of faults examined. However, we frequently see the criteria mentioned above—including exception, weak mutation, and output coverage. Interestingly, despite the similarity in distance functions and testing intent, direct branch and line coverage are often more effective than branch coverage in situations where it has a low chance of detection. In these cases, the criteria drive tests to interact in such a way with the CUT that they are better able to detect the fault. The efficacy of alternative criteria in situations where the overall top performer offers poor results further emphasizes that:

Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults.

⁶<https://github.com/Greg4cr/defects4j/blob/master/framework/projects/Math/patches/100.src.patch>

⁷<https://github.com/Greg4cr/defects4j/blob/master/framework/projects/Mockito/patches/26.src.patch>

Given its strong overall performance, we would recommend that practitioners prioritize branch coverage—of the studied options—when generating test suites. However, we also stress that other criteria should not be ignored. Several—including exception and output coverage—can be quite effective at times, even if they are not effective on average. More research is clearly needed to codify the situations where such criteria can be effective and should be employed. Alternatively, such criteria could be used in *combination* with generally effective criteria such as branch coverage.

4.3 | Combinations of Fitness Functions

The analysis above presupposes that only one fitness function can be used to generate test suites. However, many search-based generation algorithms can simultaneously target multiple fitness functions. EvoSuite’s default configuration, in fact, attempts to satisfy all eight of the fitness functions examined in this study [58]. In theory, suites generated through a combination of fitness functions could be more effective than suites generated through any one objective because—rather than focusing exclusively on one goal—they can simultaneously target multiple facets of the class under test.

For example, combining exception and branch coverage may result in a suite that both thoroughly explores the structure of the system (due to the branch obligations) and rewards tests that throw a larger number of exceptions. Such a suite may be more effective than a suite generated using branch or exception coverage alone. In fact, rather than generating tests for multiple independent criteria—when one or more of those criteria may only be effective situationally—a tester could, in theory, simultaneously generate for a combination of criteria in an attempt to produce suites effective in all situations.

To better understand the potential of combined criteria, we have generated tests for two combinations. The first is EvoSuite’s default combination of the eight criteria that were the focus of this study. The second is a more lightweight combination of branch, exception, and method coverage—a combination of the most effective criterion overall with two that were selectively effective. In a recent study on combination of criteria on a subset of the Defects4J database, the BC-EC-MC combination was suggested as an effective baseline for new, untested systems [25].

Overall, EvoSuite’s default configuration performs well, but fails to outperform all individual criteria in most situations. Table 3 shows that the default configuration detects 245 faults—fewer than branch and direct branch coverage, but more than the other individual criteria. It also uniquely detects two faults at the two-minute budget level and five at the ten-minute level (see Table 7). At the two minute level, this is worse than several individual criteria, but at the ten-minute level, it finds more faults than any individual criterion. According to Table 4, the default configuration’s average overall likelihood of fault detection is 20.56% (2m budget)-24.69% (10m budget). At the two-minute level, this places it below branch, and direct branch coverage. At the ten-minute level, it falls below branch coverage, but above all other criteria. This places the default configuration in the top cluster—an observation confirmed by statistical tests (Tables 5 and 6).

However, it fails to outperform branch coverage in almost all situations. In theory, a combination of criteria should be able to detect more faults than any single criterion. In practice, combining *all* criteria results in suites that are fairly effective, but fail to reliably outperform individual criterion. The major reason for the less reliable performance of this configuration is the difficulty in attempting to satisfy so many obligations at once. As noted in Table 2, the default configuration must attempt to satisfy, on average, 1681 obligations. The individual criteria only need to satisfy a fraction of that total. As a result, the default configuration also benefits more than any individual criterion from an increased search budget—a 20.10% improvement in efficacy.

EvoSuite’s default configuration has an average 20.56-24.69% likelihood of fault detection—in the top cluster, but failing to outperform all individual criteria.

Our observations imply that combinations of criteria could be more effective than individual criteria. However, a combination of all eight criteria results in unstable performance—especially if search budget is limited. Instead, testers may wish to identify a smaller, more targeted subset of criteria to combine during test generation. In fact, we can see the wisdom in such an approach by examining the results for the focused BC-EC-MC combination.

From Table 3, we can see that the BC-EC-MC combination finds a total of 269 faults—more than any individual criterion. From Table 4, this combination has a 24.03% (two-minute) to 27.84% (ten-minute) likelihood of detection. Again, this is better than any of the individual criteria. This combination also detects four faults uniquely at the two-minute budget, 13 at the ten-minute budget, and one fault regardless of the budget. The boxplots in Figure 6 show that the BC-EC-MC combination has a higher third-quartile box than any other method, indicating that it returned more results in that range than other methods. Statistical tests place this configuration in the top cluster, where it also outperforms weak mutation coverage with significance (Tables 5 and 6).

There are still situations where this combination can be outperformed by an individual criterion, particularly at the two-minute budget level. This combination requires fewer obligations than the eight-way default combination—around 354, on average—but still more than any individual criterion. This means that the combination benefits from a larger search budget (15.86% average improvement), and offers more stable performance at higher budgets. Still, this combination is clearly quite effective.

Of the studied criteria, exception coverage is unique in that it does not prescribe static test obligations. Rather, it simply rewards suites that cause more exceptions to be thrown. This means that it can be added to a combination with little increase in search complexity. The simplicity of exception coverage explains its poor performance as the *primary* criterion. It lacks a feedback mechanism to drive generation towards exceptions. However, exception coverage appears to be very effective *when paired with criteria that effectively explore the structure of the CUT*. Branch coverage gives exception coverage the feedback mechanism it needs to explore the code. Adding exception coverage to branch coverage adds little cost in terms of generation difficulty, and generally outperforms the use of branch coverage alone.

An example of effective combination can be seen in fault 60 for Lang⁸—a case where two methods can look beyond the end of a string. No single criterion is effective, with a maximum of 10% chance of detection given a two-minute budget and 20% with a ten-minute budget. However, combining branch and exception coverage boosts the likelihood of detection to 40% and 90% for the two budgets. In this case, if the fault is triggered, the incorrect string access will cause an exception to be thrown. However, this only occurs under particular circumstances. Therefore, exception coverage alone never detects the fault. Branch coverage provides the necessary means to drive program execution to the correct location. However, two suites with an equal coverage score are considered equal. Branch coverage alone may prioritize suites with slightly higher (or different) coverage, missing the fault. By combining the two, exception-throwing tests are prioritized and retained, succeeding where either criterion would fail alone.

Method coverage adds another “low-cost” boost. In general, a class will not have a large number of methods, and methods are either covered or not covered. Thus, even if method coverage is not a particularly helpful addition to a combination, its inclusion does not substantially increase the number of obligations that the test generator is tasked with fulfilling. An example where the addition of method coverage boosts efficacy can be seen in Lang fault 34⁹. This fault resides in two small (1-2 line) methods. Calling either method will reveal the fault, but branch coverage alone can easily overlook them because their invocation does not substantially improve branch coverage of the class as a whole. The addition of method coverage adds a useful “reminder” for the generator to invoke these simple methods.

A combination of branch, exception, and method coverage has an average 24.03-27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.

It is unlikely that the BC-EC-MC combination is the strongest possible combination, and we can see some situations where a single criterion is still the most effective. In fact, it is unlikely that any one criterion or combination of criteria will ever universally be the “best”. The most effective criteria depend on the type of system under test, and the types of faults that the developers have introduced into the code. Still, there is a powerful idea at the heart of this combination. When generating tests, a strong coverage-focused criterion should be selected as the primary criterion. Then, a small number of targeted, orthogonal criteria can be added to that primary criterion. More investigation is needed into the situational applicability of criteria in order to better understand when any one criterion or a combination of criteria will be effective.

4.4 | Understanding the Generation Factors Impacting Fault Detection

Using the TAR3 treatment learner [27], we have generated five treatments from each of the two generation factor datasets for each of the three classifications (“Not Detected”, “Low Performance”, and “High Performance”). The treatments are scored according to their impact on class distribution, and top-scoring treatments are presented first.

First, the following treatments indicate the factors pointing most strongly to a “high” likelihood of fault detection:

Two-Minute Dataset:

1. BC (fixed) > 79.71%, LC (fixed) > 87.89%, % of obligations satisfied > 89.59%, BC (faulty) > 79.85%
2. BC (fixed) > 79.71%, LC (fixed) > 87.89%, % of obligations satisfied > 89.59%, LC (faulty) > 88.09%
3. LC (fixed) > 87.89%, % of obligations satisfied > 89.59%, BC (faulty) > 79.85%
4. BC (fixed) > 79.71%, % of obligations satisfied > 89.59%, LC (faulty) > 88.09%

⁸<https://github.com/apache/commons-lang/commit/a8203b65261110c4a30ff69fe0da7a2390d82757>.

⁹<https://github.com/apache/commons-lang/commit/496525b0d626dd5049528cdef61d71681154b660>

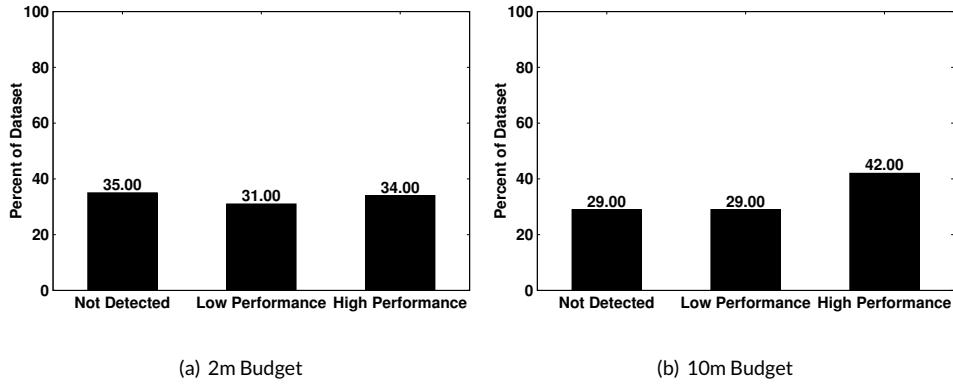


FIGURE 8 Class distributions of the data subsets fitting the top treatments learned from each dataset for the “High Performance” class.

5. BC (fixed) > 79.71%, LC (fixed) > 87.89%, % of obligations satisfied > 89.59%, BC (faulty) > 79.85%, LC (faulty) > 88.09%

Ten-Minute Dataset:

1. BC (faulty) > 86.25%, % of obligations satisfied > 94.34%, LC (faulty) > 92.08%
2. BC (faulty) > 86.25%, % of obligations satisfied > 94.34%, LC (fixed) > 92.23%
3. BC (faulty) > 86.25%, % of obligations satisfied > 94.34%, LC (faulty) > 92.08%, LC (fixed) > 92.23%
4. BC (faulty) > 86.25%, % of obligations satisfied > 94.34%, BC (fixed) > 85.97%, LC (fixed) > 92.23%
5. BC (faulty) > 86.25%, % of obligations satisfied > 94.34%, BC (fixed) > 85.97%, LC (faulty) > 92.08%

In Figure 8, we plot the class distribution of the subset fitting the highest-ranked treatment learned from both datasets. Comparing the plots in Figure 4 to the subsets in Figures 8, we can see that the treatments do impose a large change in the class distribution—a lower percentage of cases have the “Not Detected” class, and more have the other classifications. This shows that the treatments do reasonably well in predicting for success. Test suites fitting these treatments are not guaranteed to be successful, but are likely to be.

Note that some treatments are subsets of other treatments. For example, the third treatment for the two-minute dataset above is a subset of the top treatment. Each treatment indicates a set of attributes and value ranges for those attributes that, when applied *together*, tend to lead to particular outcomes. A smaller treatment that is a subset of a larger treatment, when applied, will lead to a different subset of the overall data set with a different class distribution to the larger treatment. In general, smaller treatments are easier for humans to understand—this is why we have limited treatment size to five attributes. However, within that limit, a larger treatment may have more explanatory power than a smaller treatment. In this case, as treatments are ranked by score, the larger treatment is more indicative of the target class and the additional factors offer additional explanatory power.

We can make several observations. First, the most common factors selected as indicative of efficacy are all coverage-related factors. Even if their goal is not to attain coverage, successful suites thoroughly explore the structure of the CUT. The fact that coverage is important is not, in itself, entirely surprising—if patched code is not well covered, the fault is unlikely to be discovered.

More surprising is how much weight is given to coverage. Suite size has been a focus in recent work, with Inozemtseva et al. (and others) finding that the size has a stronger correlation to efficacy than coverage level [36]. However, size attributes—number of tests and test length—do not appear in any of the generated treatments. Kendall correlation tests further reinforce this point. For both budgets and for both suite size and length, correlation strengths were all approximately 0.25—“weak to low” correlations.

However, rather than indicating that larger test suites are not necessarily more effective at detecting real faults, it is important to look at the suites themselves. As can be seen in Section 3.4, test suite sizes do not range dramatically between each of the fitness configurations. Within the size ranges of suites in this study, larger suites also do not necessarily outperform smaller suites. Suites for Output Coverage, which performs poorly on average, are often similar in size to those yielded for the higher-scoring fitness functions. The suites for the combinations are, naturally, the largest. However, the largest suites belong to the “default” combination—which is often outperformed by branch coverage and the BC-EC-MC combination. Exception Coverage, the poorest performing fitness function on average, does have the smallest test suites. However, other factors, such as its low coverage of source code, seem to play a larger role in determining suite efficacy than size alone.

The other factor noted as indicative of efficacy is the percent of obligations satisfied. This too seems reasonable. If a suite covers more of its test obligations, it will be better at detecting faults. For coverage-based fitness functions like branch and line coverage, a high level of satisfied obligations

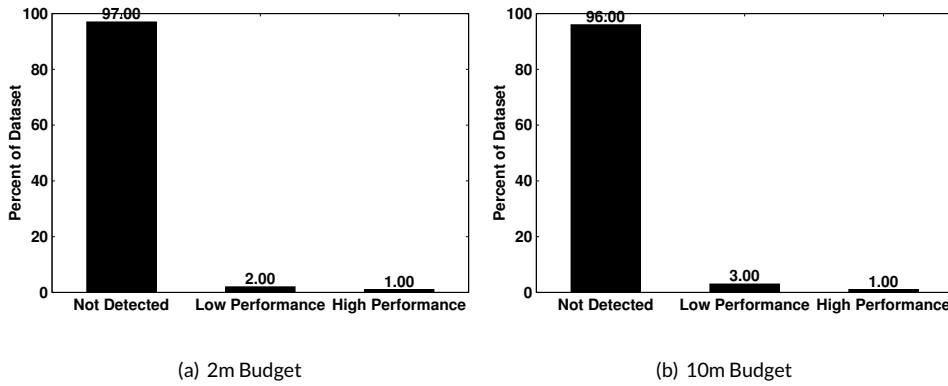


FIGURE 9 Class distributions of the data subsets fitting the top treatments learned from each dataset for the “Not Detected” class.

naturally correlates with a high level of branch or line coverage. For other fitness functions, the correlation may not be as strong, but it is also likely that suites satisfying more of their obligations also tend to explore more of the structure of the CUT.

Factors that strongly indicate a high level of efficacy include line or branch coverage over either version of the code and high coverage of their own test obligations. Coverage and obligation satisfaction are favored over factors related to suite size or test obligations.

Note, however, that we still do not entirely understand the factors that indicate a high probability of fault detection. From Figure 8, we can see that the treatments radically alter the class distribution from the baseline in Figure 4. Still, we can also see that suites from the “Not Detected” class still form a significant portion of that class distribution. From this, we can conclude that factors predicted by treatments are a necessary precondition for a high likelihood of fault detection, but are not sufficient to ensure that faults are detected. Unless code is executed, faults are unlikely to be found. Thus, coverage is impotent. However, how code is executed matters, and execution alone does not guarantee that faults are triggered and observed as failures. The fitness function determines how code is executed. It may be that fitness functions based on stronger adequacy criteria (such as complex condition-based criteria [67]) or combinations of fitness functions will better guide such a search. While coverage increases the likelihood of fault detection, it does not ensure that suites are effective.

To better understand factors indicating success, we can also perform treatment learning for the opposite scenario—what indicates that we will not detect a fault? Factors that indicate a lack of success include:

Two-Minute Dataset:

1. LC (faulty) \leq 11.77%, LC (fixed) \leq 12.41%, BC (faulty) \leq 8.96%
2. LC (faulty) \leq 11.77%, LC (fixed) \leq 12.41%, BC (faulty) \leq 8.96%, BC (fixed) \leq 9.33%
3. LC (faulty) \leq 11.77%, LC (fixed) \leq 12.41%
4. LC (faulty) \leq 11.77%
5. LC (faulty) \leq 11.77%, BC (faulty) \leq 8.96%

Ten-Minute Dataset:

1. LC (fixed) \leq 15.02%, BC (fixed) \leq 12.12%
2. LC (fixed) \leq 15.02%, BC (faulty) \leq 11.91%
3. LC (fixed) \leq 15.02%, BC (faulty) \leq 11.91%, BC (fixed) \leq 12.12%
4. BC (fixed) \leq 12.12%
5. LC (fixed) \leq 15.02%, LC (faulty) \leq 14.54%

In Figure 9, we plot the class distribution of the subset fitting the highest-ranked treatment learned from both datasets for the “Not Detected” outcome. Comparing the plots in Figure 4 to the subsets in Figures 8, we can see a dramatic change in the class distribution. These treatments predict quite clearly a lack of success, with almost no data records from the other classes still matching the treatment.

The factors indicating a lack of success are entirely coverage-based. If coverage—line or branch—is less than approximately 15%, then the odds of effective fault detection are extremely low. This further reinforces the discussion above:

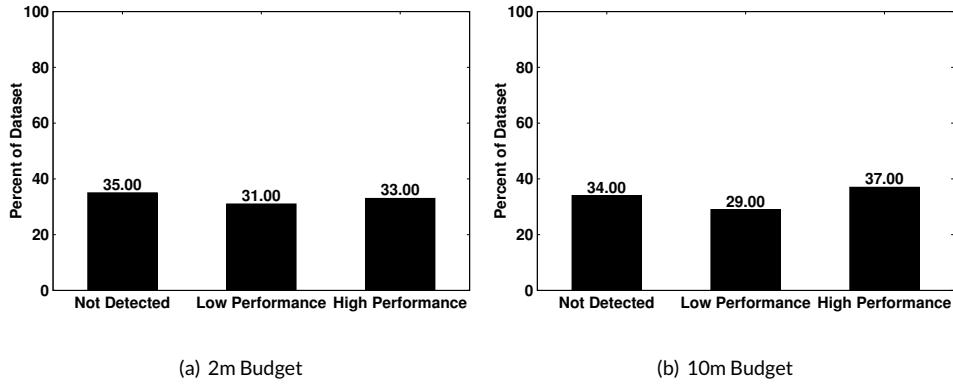


FIGURE 10 Class distributions of the data subsets fitting the top treatments learned from each dataset for the “Low Performance” class.

While coverage may not ensure success, it is a prerequisite. If the code is not exercised, then the fault will not be found.

Finally, we can examine one additional classification—“low” efficacy. What factors differentiate situations where a fault is highly likely to be found from situations where it is still generally found, but with a low likelihood of detection? The factors that suggest this situation include:

Two-Minute Dataset:

1. LC (faulty) > 88.09%, % of obligations satisfied > 89.59%, LC (fixed) > 87.89%
2. LC (faulty) > 88.09%, % of obligations satisfied > 89.59%
3. BC (faulty) > 79.85%, BC (fixed) > 79.71%, % of obligations satisfied > 89.59%, LC (fixed) > 87.89%
4. BC (fixed) > 79.71%, LC (fixed) > 87.89%, % of obligations satisfied > 89.59%, BC (faulty) > 79.85%, LC (faulty) > 88.09%
5. BC (fixed) > 79.71%, LC (faulty) > 88.09%, % of obligations satisfied > 89.59%, LC (fixed) > 87.89%

Ten-Minute Dataset:

1. LC (faulty) > 92.08%, BC (fixed) > 85.97%
2. LC (faulty) > 92.08%, BC (fixed) > 85.97%, LC (fixed) > 92.23%
3. BC (faulty) > 86.25%, BC (fixed) > 85.97%, LC (fixed) > 92.23%
4. BC (faulty) > 86.25%, BC (fixed) > 85.97%, LC (faulty) > 92.08%
5. BC (faulty) > 86.25%, LC (fixed) > 92.23%

These treatments, and their resulting class distributions—illustrated in Figure 10—are very similar to the factors predicting “high” performance. This is particularly true for the two-minute dataset, where we simply see a small downgrade in the number of “high” cases. From this, we can again see that coverage is needed to detect faults, but more data is needed to help ensure reliable detection.

However, we can make one interesting observation from the treatments learned from the ten-minute dataset. The ten-minute dataset includes more effective test suites from the start, allowing us to better differentiate high efficacy from low—but extant—efficacy. The treatments learned from the ten-minute dataset for “low efficacy” are lacking any reference to satisfaction of their obligations—a factor that is always present in the treatments learned for the “high efficacy” classification. We can also see a shift in the resulting class distribution in Figure 10 from that in Figure 8. The percent of “low” efficacy examples remains the same, but there are fewer “high” cases and more “not detected” cases.

The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion’s test obligations.

Thus, we can observe that the most effective test suites are those that both cover a large percentage of their own obligations and thoroughly exercise the targeted code. In the case of criteria like branch coverage, these two go hand-in-hand. However, this also illustrates why criteria based on orthogonal factors to code coverage—like exceptions—tend to be best used in combination with coverage-based criteria. In future work, we will further explore such factors and others, and investigate how to best ensure effective test suite generation.

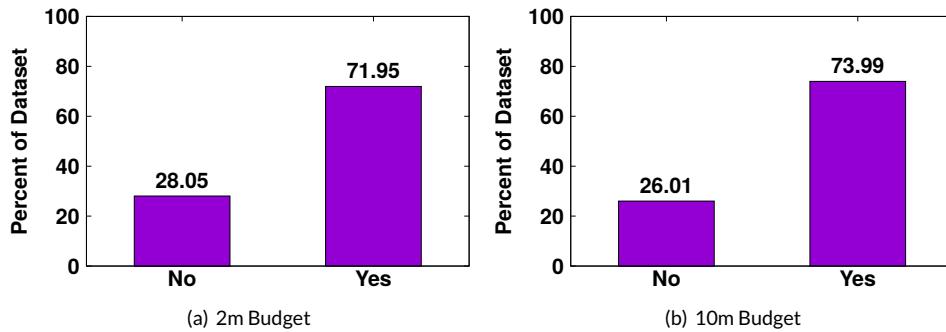


FIGURE 11 Class distributions for the subsets of the two overall datasets fulfilling the top-ranked “Yes” treatment for each.

4.5 | Understanding the Code Metrics Impacting Fault Detection

The following treatments were reported by TAR3, from the “overall” (all fitness functions) code metric datasets, as indicative of situations where generated test suites—regardless of the targeted criterion—were able to detect a fault. The treatments are scored according to their impact on class distribution and the number of cases in the treatment-fulfilling subset of the data, and the top-scoring treatments are presented first. Definitions of metrics are listed in Table 1.

Two-Minute Overall Dataset:

1. TCD = [0.52..0.93], NOD = [0.00..1.00]
2. TCD = [0.52..0.93], NS = [0.00..1.00]
3. TCD = [0.52..0.93], TNLS = [0.00..1.00]
4. TCD = [0.52..0.93], TNS = [0.00..1.00]
5. TCD = [0.52..0.93], TNLS = [0.00..1.00], NS = [0.00..1.00]

Ten-Minute Overall Dataset:

1. TCD = [0.52..0.93], CD = [0.53..0.93]
2. TCD = [0.52..0.93]
3. PDA = [30.00..208.00], DLOC = [348.00..4017.00]
4. TNLPM = [38.00..287.00], DLOC = [348.00..4017.00]
5. CD = [0.53..0.93]

Figure 11 illustrates the shift in the class distribution for the subset of each overall dataset fitting the top-ranked treatment targeting the “Yes” classification for each respective budget. Comparing to the baseline distribution in Figure 5, we can see that the class distribution has shifted heavily in favor of the “Yes” class—from 47.44-71.95% and 49.80-73.90% respectively. As approximately 25% of the examples still have a “No” verdict, these treatments are not perfectly explanatory. Some classes may match the treatment and still evade fault detection. However, the shift in distribution still suggests that the metrics and value ranges named in the treatments have explanatory power. The treatments indicate that:

Generated test suites are effective at detecting faults in well-documented classes.

The most consistently-identified metric from the two-minute dataset—and one that appears in treatments for the ten-minute dataset as well—is that a high Total Comment Density (52-93%) tends to indicate that the fault is more likely to be found. This is above the 75th percentile of the results depicted in Figure 1. From the ten-minute dataset, we can also see that suites tend to detect faults in classes with a Comment Density of 52-93%, 348-4,017 Documented Lines of Code, and with 30-208 documented public methods (PDA).

Test generation will yield better results if more of the class is publicly accessible.

A Total Number of Local Public Methods from 38-287 indicates that generated suites are more likely to detect a fault. From Table 1, we can see that this is well above the median TNLPM (12.00), indicating that allowing direct access to more of your methods will yield better test generation

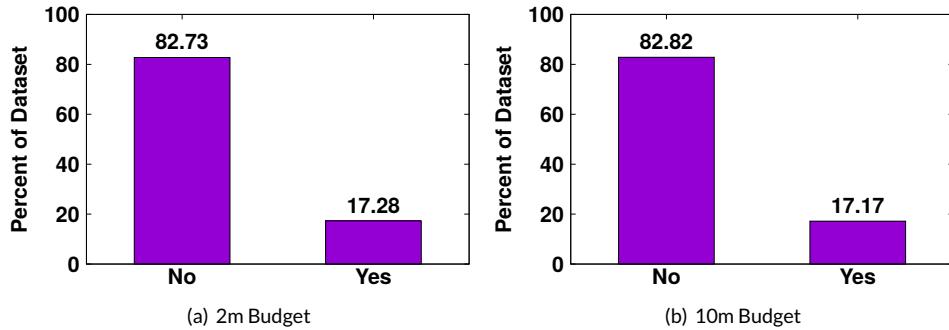


FIGURE 12 Class distributions for the subsets of the two “overall” datasets fulfilling the top-ranked “No” treatment for each.

results. A number of treatments also suggest a (Total) Number of (Local) Setters (TNS, NS, TNLS) of 0. Having no setters implies that all attributes are publicly-accessible. In addition, while Public Documented API is intended to capture how well-documented the class is, it also illustrates this point—a high PDA indicates not just that there are a large number of documented methods, but that a large number of methods are public as well.

The top treatment for the two-minute dataset also suggests a Number of Descendants (NOD) of 0. This value reflects the vast majority of classes, and also appears in the treatments for the “No” classification. Therefore, we consider it to be a coincidental factor.

The test suites for Exception, Method, Method (Top Level, No Exception), and Output Coverage did not detect enough faults to yield useful treatments for the “Yes” classification. Test suites for the remaining four criteria—Branch, Direct Branch, Line, and Weak Mutation Coverage—yielded treatments largely echoing the “overall” dataset. However, multiple treatments for those criteria-specific datasets included the metric-value pairings CLLC = [0.24..0.86] and CC = [0.27..0.91]. These metrics—Clone Logical Line Coverage (CLLC) and Clone Coverage (CC)—tell us that:

Faults are easier to detect if a large proportion of the class contains duplicate code.

These value ranges are the high end of the scale, and do not include the majority of classes, indicating that the importance of these metrics is not coincidental. Intuitively, if there is a lot of duplicate code, the overall class structure will be easier to cover. Test generation methods driven by code coverage will be able to quickly achieve high levels of coverage, making it easier to reach and execute the code containing the fault.

The following treatments were reported by TAR3, based on the all-fitness-function datasets, as indicative of situations where generated test suites—regardless of the targeted criterion—were **not able** to detect a fault. Figure 12 illustrates the shift in the class distribution for the subset of each overall dataset fitting the top-ranked treatment targeting the “No” classification for each.

Two-Minute Overall Dataset:

1. DIT = [1.00..2.00], PDA = [1.00..4.00]
2. NPM = [0.00..3.00], NOC = [0.00..1.00]
3. NPM = [0.00..3.00], NOD = [0.00..1.00]
4. NPM = [0.00..3.00], TNLPA = [0.00..1.00]
5. CBO = [17.00..98.00], NOC = [0.00..1.00]

Ten-Minute Overall Dataset:

1. NPM = [0.00..3.00], LCOM5 = [1.00..2.00]
2. NS = [0.00..1.00], CBO = [17.00..98.00]
3. NPM = [0.00..3.00], NOC = [0.00..1.00]
4. NPM = [0.00..3.00], NOD = [0.00..1.00]
5. NPM = [0.00..3.00], NOD = [0.00..1.00], NLPA = [0.00..1.00]

Comparing to the baseline distribution in Figure 5, we can see that the class distribution has shifted heavily in favor of the “No” class—from 52.56–82.73% and 50.20–82.82% respectively. Once again, not all examples in the subset have a “No” classification. However, the sharp shift in distribution suggests that the treatments have explanatory power. From the produced treatments, we can see that:

Test generation methods struggle with classes that have a large number of private methods or attributes.

The majority of the treatments include a Number of Public Methods (NPM) between 0-3 methods. This is an extremely low number of public methods—far below the median—indicating that much of the class is private. In these cases, the generated suites are likely to have low overall code coverage, and are more likely to miss a fault. One treatment also includes a PDA of 1-3 methods. In this case, this metric’s importance is likely not to be as an indicator of the level of documentation, but an indicator that there are a low number of public methods.

Treatments also include a (Total) Number of Local Public Attributes (TNLPA/NLPA) of 0—indicating that any extant attributes are private. Like with the “Yes” treatments, the attributes serve a different purpose than methods in test generation—serving as a way to configure class state and drive coverage of the code contained in the methods. A lack of public attributes limits the ability of the generation framework to control class state.

It can be more difficult to generate tests for classes with dependencies or inherited state.

A Coupling Between Objects of 17-98 is well above the mean of 8, indicating that generated suites are likely to miss a fault in a class coupled to a large number of other classes. In such cases, the test generation framework would need to set up dependencies and put them in the state needed to expose the fault in the targeted class—a non-trivial task.

A related metric is the Depth of the Inheritance Tree (DIT). A DIT of 1 indicates that the target class has a parent class. The existence of a parent class indicates that some methods and attributes are inherited from a parent. This, in itself, is not a problem as the test generator does not need to properly set up a parent like in the last case (and many classes in the dataset have a parent). While we have discussed the metric-value pairs in each treatment as largely being independent, the pairs in a treatment can be related. In this case, the treatment pairs the DIT of 1 with a low PDA. This implies situations where part of the class is inherited from a parent, and where the class has a low number of public methods of its own. If much of the complexity of the class is inherited and new functionality is largely private, the test generation framework may have difficulty in driving execution to the fault location.

Once again, the fact that the vast majority of classes have a NOC/NOD of 0 and that this metric-value pair appears for both targeted classes suggests that these two metrics are coincidental, and we have chosen to ignore them in our discussion. LCOM5 = 1 matches the vast majority of classes in the dataset, and is not particularly informative. Therefore, we also suspect that it is coincidental. All three of these metrics—NOC, NOD, and LCOM5—are paired with a low NPM. We suspect that the low NPM is the true indicator of test efficacy.

One treatment includes the metric-value pair NS = 0. This pairing also appeared for the “Yes” treatments, and captures a majority of case examples. We believe it is coincidental for this classification, but not the “Yes” classification, as the treatments for the “Yes” classification included a number of related metrics and often included this pairing. In this case, it was paired with a high CBO—a more informative metric.

The datasets for all eight coverage criteria offered very similar results to the overall datasets for the “No” classification. No additional metric-value pairs were observed.

We will summarize the trends observed among the identified metrics, and discuss their implications. First, **test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when the class structure is accessible**. The clearest indication of this trend is in the treatments produced for the “No” classification, where a low number of public methods appeared in almost every treatment. In the “Yes” treatments, a large number of public methods is prescribed. This finding is further backed by the prevalence of the Public Documented API metric for both classifications, where the “Yes” treatments prescribe for a PDA of 30-208 methods, and in the “No” treatments, where the treatments prescribe 1-3 methods. PDA is a documentation metric, intended to highlight classes in need of more documentation. However, as it measures the proportion of public methods that are documented, it has a strong correlation to the Number of Public Methods (0.63, measured using the Kendall correlation test). In this experiment, PDA seems to further indicate the effect of private methods on test generation effectiveness.

This finding makes intuitive sense. The test generation technique explored in this experiment is driven by various coverage criteria—largely based on different ways of executing structural elements of the code. The obligations of such criteria will inevitably require that code structures within private methods be executed in the mandated manner. In practice, coverage can be measured over private code, and the feedback mechanisms that power search-based test generation will still reward greater coverage of that code. However, without the ability to directly call such methods, the generation technique will struggle to actually obtain coverage. Private methods must be covered *indirectly*, through calls to public methods. While the generation technique will attempt to adjust the input provided to the public methods to obtain higher indirect coverage of private methods, this indirect manipulation may prove unsuccessful due to constraints placed on how the public method can invoke the secondary private method or discontinuity in the scoring method introduced by this indirect manipulation.

Both the “Yes” and “No” treatments touch on the use of private attributes as well, through the (Total) Number of (Local) Setters and Total Number of Public Attributes. Private attributes do not directly prevent code from being covered in the same manner as private methods do. However, they may still result in lower coverage and missed faults by limiting the ability of the test generation technique to manipulate the state of the class.

Certain methods may only be coverable by setting the class attributes to particular values, and some failures may only be triggered under particular class states. If the class attributes are private, the test generation technique will need to find indirect means of manipulating those attributes. Again, this is a non-trivial task.

Authors have previously hypothesized that private methods are a reason for poor test generation results [60, 8, 24]. Our findings offer evidence supporting this hypothesis. In practice, we would not advocate that developers reduce the use of private methods or attributes—the protection offered through this feature is crucial. Instead, test generation techniques need to be augmented with better means of increasing coverage of private methods. For example, machine learning techniques may be able to form a behavioral model of the indirectly-called method that could offer better feedback than the existing scoring function used to guide search-based generation.

Second, **generated suites seem to be more effective at detecting faults in classes with more documentation**. The metrics that were the most common indicators of success are largely documentation-related metrics such as the (Total) Comment Density, the Documented Lines of Code, and—to a lesser extent—the Public Documented API. As discussed previously, Public Documented API is strongly correlated to the Number of Public Methods, and the documentation connection is likely to be a coincidence. TCD and DLOC both have a moderate correlation to the PDA (strengths of 0.48 and 0.47, respectively). However, TCD is only weakly correlated to NPM (0.24) and DLOC is only moderately correlated (0.42). Therefore, TCD, CD, and DLOC are important indicators of efficacy in their own right, and are not merely indicative of a higher number of public methods.

There is no reason to expect the presence of documentation to assist automated test case generation techniques, as such techniques do not make use of documentation in any way. Instead, *it is important to consider what the presence of documentation implies*. One theory is that there is an unintended selection bias in how the case examples were gathered for Defects4J. The studied faults were identified by searching for project commits that referenced bug reports. Bug reports are more likely to be filed for classes that are frequently invoked by the users and developers of a project. In turn, classes that are more heavily used tend to be ones that developers spend more time refining and polishing. Classes that are expected to be used more heavily will, naturally, be better documented. As a result, it could be that well-documented classes are more likely to be identified as subjects for Defects4J than classes with low amounts of documentation.

However, this theory does not completely explain these results. The majority of classes in Defects4J do not have a high TCD or DLOC. The median TCD—36%—is likely to be higher than the median for all classes in the wild, but is still well below the TCD prescribed by the treatments—52–93%. The classes with a high TCD are also not necessarily smaller than other examples. The median LLOC—non-comment lines of code—for classes with a TCD greater than 52% is 288.50, compared to an overall median of 208.50. The median TNM is 37.50, slightly above the median of 37. This implies that the well-documented classes are actually larger than the average class in Defects4J. These are not simpler examples. Further, TCD and DLOC do not have a strong correlation with any non-documentation metric, meaning that there is not a simple explanation within the collected data.

Therefore, there must be some additional factor implied by the presence of high levels of documentation. More research is needed to understand the impact of documentation in these case examples. While the presence of documentation should not directly assist automated test case generation techniques, its presence may hint at the maturity, testability, and understandability of the class.

Third, **classes with a large number of dependencies are more difficult to test than more self-contained classes**. This is indicated in the “No” treatments by a Coupling Between Objects (CBO) of 17–98 classes, and—to a lesser extent—an inheritance tree depth (DIT) of 1, meaning that the target class inherits functionality from a parent. For a class to be included in Defects4J, it must be directly changed by the patch applied to fix the fault. This means that the fault lies in the class that depends on other classes, rather than being a fault in one of the dependencies.

If a class depends on other classes, then the test generation technique may also need to initialize and manipulate the dependencies properly as part of the test generation process. By not doing so properly, we may not be able to achieve high coverage of the target class. Further, we may fail to expose the fault even if the code is covered, as we are trying to make use of the target class outside of the “normal” use of the rest of its dependencies. Because of that, we may see the same incorrect behavior from both the working and faulty versions of the class, missing the fault. Researchers have discussed the problem of configuring complex dependencies as part of the broader challenges of controlling the execution environment when generating test cases [8, 7, 23]. Again, our observations provide clear motivation for further research on this topic.

Fourth, when structure-based criteria like Branch Coverage are targeted, **test generation techniques are more effective when a large proportion of the class is duplicated code**. This is supported by the prevalence of Clone Logical Line Coverage (CLLC) and Clone Coverage (CC) in the criteria-specific “Yes” treatments. This observation makes intuitive sense. If a large proportion of the code is identical, then that code will be easier to cover through automated generation. Even if the fault does not lie in the duplicated code, it will be easier to guide execution to the faulty code.

Code duplication is discouraged during development, as any changes will need to be made in multiple locations. Further, duplicating code rather than encapsulating it in one location and invoking it throughout the class will not have any benefit, as a high CLLC simply implies that there is not a significant quantity of non-duplicated code. Rather than offering actionable information, this observation simply indicates that classes with a lot of duplicate code and very little other functionality are easier to test than complex classes.

5 | RELATED WORK

Those who advocate the use of adequacy criteria hypothesize that criteria fulfillment will result in test suites more likely to detect faults—at least with regard to the structures targeted by that criterion. If this is the case, we should see a correlation between higher attainment of a criterion and the chance of fault detection for a test suite [32]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed [52, 48, 11, 18, 19, 17, 30, 36, 29]. Inozemtseva et al. provide a good overview of work in this area [36]. Our focus differs—our goal is to examine the relationship between fitness function and fault detection efficacy for search-based test generation. However, fitness functions are largely based on, and intended to fulfill, adequacy criteria. Therefore, there is a close relationship between the fitness functions that guide test generation and adequacy criteria intended to judge the resulting test suites. In recent work, McMinn et al. have even proposed using search techniques to evolve new coverage criteria that combine features of existing criteria [46].

EvoSuite has previously been used to generate test suites for the systems in the Defects4J database. Shamshiri et al. applied EvoSuite, Randoop, and Agitar to each fault in the Defects4J database to assess the general fault-detection capabilities of automated test generation [60]. They found that the combination of all three tools could identify 55.7% of the studied faults. Their work identifies several reasons why faults were not detected, including low levels of coverage, heavy use of private methods and variables, and issues with simulation of the execution environment. In their work, they only used the branch coverage fitness function when using EvoSuite. Yu et al. used EvoSuite to generate tests for 224 of the faults in Defects4J, examining whether such tests could be used for program repair [69]. In our initial study, we expanded the number of EvoSuite configurations to better understand the role of the fitness function in determining suite efficacy [24]. We have also compared and contrasted test suites generated to achieve traditional branch coverage and direct branch coverage, noting that each fitness function detects different faults [26].

We used the Defects4J faults to understand the effect of combining fitness functions, identifying lightweight combinations of fitness functions that could effectively detect faults [25]. Rojas et al. also examined combining fitness functions, finding that, given a fixed generation budget, multiple fitness functions could be combined with minimal loss in coverage of any single criterion and with a reasonable increase in test suite size [58]. Others have explored combinations of coverage criteria with non-functional criteria, such as memory consumption [41] or execution time [68]. Few have studied the effect of such combinations on fault detection. Jeffrey et al. found that combinations are effective following suite reduction [37]. Recent efforts have been made to introduce many-objective search algorithms that can better balance and cover multiple coverage criteria simultaneously [12, 54].

Object-oriented source code metrics have been used for a variety of purposes. For example, Chowdhury et al. used complexity, coupling, and cohesion metrics as early indicators of vulnerabilities [14]. Singh et al. tried to find the relationship between object-oriented metrics and fault-proneness at different fault severity levels [61]. Mansoor et al. used metrics to detect code smells [44]. Cinneide et al. used cohesion and cloning metrics to guide automated refactoring [53]. Tripathi et al. [66] developed models to predict the change-proneness of the classes using code metrics [66]. Relevant to this study, Toth et al. used SourceMeter to gather the same metrics that we used on classes from Java projects on GitHub [65]. They gathered metrics for multiple revisions, focusing on pairs of revisions related to faulty and working versions of the system. They used this dataset for defect prediction. While our purposes differ and there is no overlap in the studied systems, our dataset could potentially be used to augment their study. Recently, Sobreira et al. also assembled a dataset characterizing the faults in Defects4J [62]. Rather than focusing on class characteristics, they focus on the patches used to fix each fault, characterizing them in terms of size, spread, and the repair actions needed to perform automated program repair.

6 | THREATS TO VALIDITY

External Validity: Our study has focused on fifteen systems—a relatively small number. Nevertheless, we believe that such systems are representative of, at minimum, other small to medium-sized open-source Java systems. We believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar projects.

We have used a single test generation framework. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same number and variety of fitness functions. Therefore, a more thorough comparison of tool performance cannot be made at this time. Still, our goal is to examine the coverage criteria, not the generation framework. By using the same framework to generate all test suites, we can compare criteria on an equivalent basis.

To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, this process still yielded 118,600 test suites to use in analysis. We believe that this is a sufficient number to draw stable conclusions.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known *a priori*, and normality cannot be assumed.

Our learning results are based on a single learning technique. Treatment learning was used to analyze the gathered data, as it is designed to offer succinct, explanatory theories based on classified data [47]—fitting the goal of our work. TAR3 was thought to be appropriate, as it is the most common treatment learning approach and is competitive with other approaches [27].

7 | CONCLUSIONS

We have examined the role of the fitness function in determining the ability of search-based test generators to produce suites that detect complex, real faults. From the eight fitness functions and 593 faults studied, we can conclude:

- Collectively, 51.26% of the examined faults were detected by generated test suites.
- Branch coverage is the most effective criterion—detecting more faults than any other single criterion and demonstrating a higher likelihood of detection for each fault than other criteria (on average, a 22.60–25.24% likelihood of detection, depending on the search budget).
- Regardless of overall performance, most criteria have situational applicability, where their suites detect faults no other criteria can detect. Exception, output, and weak mutation coverage—in particular—seem to be effective for particular types of faults, even if their average efficacy is low.
- While EvoSuite’s default combination performs well, the difficulty of simultaneously balancing eight functions prevents it from outperforming all individual criteria.
- However, a combination of branch, exception, and method coverage has an average 24.03–27.84% likelihood of fault detection—outperforming each of the individual criteria. It is more effective than the default eight-way combination because it adds lightweight situationally-applicable criteria to a strong, coverage-focused criterion.
- Factors that strongly indicate a high level of efficacy include high line or branch coverage over either version of the code and high coverage of their own test obligations.
- Coverage does not ensure success, but it is a prerequisite. In situations where achieved coverage is low, the fault does not tend to be found.
- The most important factor differentiating cases where a fault is occasionally detected and cases where a fault is consistently detected is satisfaction of the chosen criterion’s test obligations. Therefore, the best suites are ones that both explore the code and fulfill their own goals, which may be—in cases such as exception coverage—orthogonal to code coverage.
- Test generation methods struggle with classes that have a large number of private methods or attributes, and thrive when a large portion of the class structure is accessible.
- Generated suites are more effective at detecting faults in well-documented classes. While the presence of documentation should not directly assist automated test generation, its presence may hint at the maturity, testability, and understandability of the class.
- Faults in classes with a large number of dependencies are more difficult to detect than those in self-contained classes, as the generation technique must initialize and manipulate multiple complex objects during generation.

Theories learned from the collected metrics suggest that successful criteria thoroughly explore and exploit the code being tested. The strongest fitness functions—branch, direct branch, and line coverage—all do so. We suggest the use of such criteria as *primary* fitness functions. However, our findings also indicate that coverage does not guarantee success. The fitness function must still execute the code in a manner that triggers the fault, and ensures that it manifests in a failure. Criteria such as exception, output, and weak mutation coverage are situationally useful, and should be applied as *secondary* testing goals to boost the fault-detection capabilities of the primary criterion—either as part of a multi-objective approach or through the generation of a separate test suite.

Our findings represent a step towards understanding the use, applicability, and combination of common fitness functions. Our observations provide evidence for the anecdotal findings of other researchers [8, 24, 23, 60, 7] and motivate improvements in how test generation techniques understand the behavior of private methods or manipulate environmental dependencies. More research is needed to better understand the factors that contribute to fault detection, and the joint relationship between the fitness function, generation algorithm, and CUT in determining the efficacy of test suites. In future work, we plan to further explore these topics.

8 | ACKNOWLEDGEMENTS

This work is supported by National Science Foundation grant CCF-1657299.

References

- [1] Albrecht, A. J. and J. E. Gaffney, 1983: Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, **SE-9**, no. 6, 639–648, doi:10.1109/TSE.1983.235271.
- [2] Ali, S., L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, 2010: A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, **36**, no. 6, 742–762.
- [3] Almulla, H., A. Salahirad, and G. Gay, 2017: Using search-based test generation to discover real faults in Guava. *Proceedings of the Symposium on Search-Based Software Engineering*, Springer Verlag, SSBSE 2017.
- [4] Alshahwan, N. and M. Harman, 2014: Coverage and fault detection of the output-uniqueness test selection criteria. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2014, 181–192.
URL <http://doi.acm.org/10.1145/2610384.2610413>
- [5] Anand, S., E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al., 2013: An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, **86**, no. 8, 1978–2001.
- [6] Arcuri, A., 2013: It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, **23**, no. 2, 119–147.
- [7] Arcuri, A., G. Fraser, and J. P. Galeotti, 2014: Automated unit test generation for classes with environment dependencies. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ACM, New York, NY, USA, ASE ’14, 79–90.
URL <http://doi.acm.org/10.1145/2642937.2642986>
- [8] Arcuri, A., G. Fraser, and R. Just, 2017: Private api access and functional mocking in automated unit test generation. *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 126–137.
- [9] Basili, V. R., L. C. Briand, and W. L. Melo, 1996: A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, **22**, no. 10, 751–761, doi:10.1109/32.544352.
- [10] Bianchi, L., M. Dorigo, L. Gambardella, and W. Gutjahr, 2009: A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, **8**, no. 2, 239–287, doi:10.1007/s11047-008-9098-4.
URL <http://dx.doi.org/10.1007/s11047-008-9098-4>
- [11] Cai, X. and M. R. Lyu, 2005: The effect of code coverage on fault detection under different testing profiles. *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, ACM, New York, NY, USA, A-MOST ’05, 1–7.
URL <http://doi.acm.org/10.1145/1082983.1083288>
- [12] Campos, J., Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, 2018: An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, doi:<https://doi.org/10.1016/j.infsof.2018.08.010>.
URL <http://www.sciencedirect.com/science/article/pii/S0950584917304858>
- [13] Chidamber, S. R. and C. F. Kemerer, 1991: Towards a metrics suite for object oriented design. *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, ACM, New York, NY, USA, OOPSLA ’91, 197–211.
URL <http://doi.acm.org/10.1145/117954.117970>
- [14] Chowdhury, I. and M. Zulkernine, 2011: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, **57**, no. 3, 294–313.
- [15] Dorigo, M. and L. M. Gambardella, 1997: Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, **1**, no. 1, 53–66.
- [16] Feldt, R. and S. Pouling, 2015: Broadening the search in search-based software testing: It need not be evolutionary. *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, 1–7.
- [17] Frankl, P. G. and O. Iakounenko, 1998: Further empirical studies of test effectiveness. *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, SIGSOFT ’98/FSE-6, 153–162.
URL <http://doi.acm.org/10.1145/288195.288298>

-
- [18] Frankl, P. G. and S. N. Weiss, 1991: An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. *Proceedings of the Symposium on Testing, Analysis, and Verification*, ACM, New York, NY, USA, TAV4, 154–164.
URL <http://doi.acm.org/10.1145/120807.120821>
- [19] — 1993: An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, **19**, no. 8, 774–787, doi:10.1109/32.238581.
- [20] Fraser, G. and A. Arcuri, 2013: Whole test suite generation. *Software Engineering, IEEE Transactions on*, **39**, no. 2, 276–291, doi:10.1109/TSE.2012.14.
- [21] — 2014: Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, **20**, no. 3, 783–812.
- [22] Fraser, G., M. Staats, P. McMinn, A. Arcuri, and F. Padberg, 2013: Does automated white-box test generation really help software testers? *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA, 291–301.
URL <http://doi.acm.org/10.1145/2483760.2483774>
- [23] Gay, G., 2016: Challenges in using search-based test generation to identify real faults in mockito. *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8–10, 2016, Proceedings*, Springer International Publishing, Cham, 231–237.
URL http://dx.doi.org/10.1007/978-3-319-47106-8_17
- [24] — 2017: The fitness function for the job: Search-based generation of test suites that detect real faults. *Proceedings of the International Conference on Software Testing*, IEEE, ICST 2017.
- [25] — 2017: Generating effective test suites by combining coverage criteria. *Proceedings of the Symposium on Search-Based Software Engineering*, Springer Verlag, SSBSE 2017.
- [26] — 2018: To call, or not to call: Contrasting direct and indirect branch coverage in test generation. *Under submission, International Conference on Software Testing*, IEEE, ICST 2018, draft available from <http://greggay.com/pdf/18dbc.pdf>.
- [27] Gay, G., T. Menzies, M. Davies, and K. Gundy-Burlet, 2010: Automatically finding the control variables for complex system behavior. *Automated Software Engineering*, **17**, no. 4, 439–468, doi:10.1007/s10515-010-0072-x.
URL <http://dx.doi.org/10.1007/s10515-010-0072-x>
- [28] Gay, G., A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl, 2016: The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Trans. Softw. Eng. Methodol.*, **25**, no. 3, 25:1–25:34, doi:10.1145/2934672.
URL <http://doi.acm.org/10.1145/2934672>
- [29] Gay, G., M. Staats, M. Whalen, and M. Heimdahl, 2015: The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, **PP**, no. 99, doi:10.1109/TSE.2015.2421011.
- [30] Gligoric, M., A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, 2013: Comparing non-adequate test suites using coverage criteria. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2013, 302–313.
URL <http://doi.acm.org/10.1145/2483760.2483769>
- [31] Gopinath, R., C. Jensen, and A. Groce, 2014: Mutations: How close are they to real faults? *25th International Symposium on Software Reliability Engineering*, 189–200.
- [32] Groce, A., M. A. Alipour, and R. Gopinath, 2014: Coverage and its discontents. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ACM, New York, NY, USA, Onward!'14, 255–268.
URL <http://doi.acm.org/10.1145/2661136.2661157>
- [33] Gui, G. and P. D. Scott, 2006: Coupling and cohesion measures for evaluation of component reusability. *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ACM, New York, NY, USA, MSR '06, 18–21.
URL <http://doi.acm.org/10.1145/1137983.1137989>
- [34] Harman, M. and B. Jones, 2001: Search-based software engineering. *Journal of Information and Software Technology*, **43**, 833–839.
- [35] Holland, J. H., 1992: *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

- [36] Inozemtseva, L. and R. Holmes, 2014: Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the 36th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE 2014, 435–445.
URL <http://doi.acm.org/10.1145/2568225.2568271>
- [37] Jeffrey, D. and N. Gupta, 2007: Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, **33**, no. 2, 108–123, doi:10.1109/TSE.2007.18.
- [38] Just, R., 2014: The major mutation framework: Efficient and scalable mutation analysis for java. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2014, 433–436.
URL <http://doi.acm.org/10.1145/2610384.2628053>
- [39] Just, R., D. Jalali, and M. D. Ernst, 2014: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, ISSTA 2014, 437–440.
URL <http://doi.acm.org/10.1145/2610384.2628055>
- [40] Kit, E. and S. Finzi, 1995: *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- [41] Lakhotia, K., M. Harman, and P. McMinn, 2007: A multi-objective approach to search-based test data generation. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ACM, New York, NY, USA, GECCO '07, 1098–1105.
URL <http://doi.acm.org/10.1145/1276958.1277175>
- [42] Linda, P. E., V. M. Bashini, and S. Gomathi, 2011: Metrics for component based measurement tools. *International Journal of Scientific & Engineering Research Volume 2, Issue 5*.
- [43] Malburg, J. and G. Fraser, 2011: Combining search-based and constraint-based testing. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, Washington, DC, USA, ASE '11, 436–439.
URL <http://dx.doi.org/10.1109/ASE.2011.6100092>
- [44] Mansoor, U., M. Kessentini, B. R. Maxim, and K. Deb, 2017: Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, **25**, no. 2, 529–552.
- [45] McMinn, P., 2004: Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, **14**, 105–156.
- [46] McMinn, P., M. Harman, G. Fraser, and G. M. Kapfhammer, 2016: Automated search for good coverage criteria: Moving from code coverage to fault coverage through search-based software engineering. *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ACM, New York, NY, USA, SBST '16, 43–44.
URL <http://doi.acm.org/10.1145/2897010.2897013>
- [47] Menzies, T. and Y. Hu, 2003: Data mining for very busy people. *Computer*, **36**, no. 11, 22–29, doi:10.1109/MC.2003.1244531.
URL <http://dx.doi.org/10.1109/MC.2003.1244531>
- [48] Mockus, A., N. Nagappan, and T. Dinh-Trong, 2009: Test coverage and post-verification defects: A multiple case study. *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, 291–301.
- [49] Mohri, M., A. Rostamizadeh, and A. Talwalkar, 2012: *Foundations of Machine Learning*. MIT Press.
- [50] Molokken, K. and M. Jorgensen, 2003: A review of software surveys on software effort estimation. *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, 223–230.
- [51] Myers, G. J. and C. Sandler, 2004: *The Art of Software Testing*. John Wiley & Sons.
- [52] Namin, A. and J. Andrews, 2009: *The influence of size and coverage on test suite effectiveness*.
- [53] Ó Cinnéide, M., L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, 2012: Experimental assessment of software metrics using automated refactoring. *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ACM, 49–58.
- [54] Panichella, A., F. M. Kifetew, and P. Tonella, 2018: Incremental control dependency frontier exploration for many-criteria test case generation. *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds., Springer International Publishing, Cham, 309–324.

-
- [55] Perry, W., 2006: *Effective Methods for Software Testing, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA.
- [56] Pezze, M. and M. Young, 2006: *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons.
- [57] Rayadurgam, S. and M. Heimdahl, 2001: Coverage based test-case generation using model checkers. *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, IEEE Computer Society, 83–91.
- [58] Rojas, J. M., J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, 2015: Combining multiple coverage criteria in search-based unit test generation. *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds., Springer International Publishing, volume 9275 of *Lecture Notes in Computer Science*, 93–108.
URL http://dx.doi.org/10.1007/978-3-319-22183-0_7
- [59] Roy, C. K. and J. R. Cordy, 2007: A survey on software clone detection research.
- [60] Shamshiri, S., R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, 2015: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, New York, NY, USA, ASE 2015.
- [61] Singh, Y., A. Kaur, and R. Malhotra, 2010: Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, **18**, no. 1, 3.
- [62] Sobreira, V., T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, 2018: Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *arXiv preprint arXiv:1801.06393*.
- [63] SourceMeter, 2014: Sourcemerter java documentation. <https://www.sourcemerter.com/resources/java/>.
- [64] Tóth, Z., P. Gyimesi, and R. Ferenc, 2016: A public bug database of github projects and its application in bug prediction. *Computational Science and Its Applications – ICCSA 2016*, O. Gervasi, B. Murgante, S. Misra, A. M. A. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds., Springer International Publishing, Cham, 625–638.
- [65] — 2016: A public bug database of github projects and its application in bug prediction. *Computational Science and Its Applications – ICCSA 2016*, O. Gervasi, B. Murgante, S. Misra, A. M. A. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds., Springer International Publishing, Cham, 625–638.
- [66] Tripathi, A. and K. Sharma, 2015: Improving software quality based on relationship among the change proneness and object oriented metrics. *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, IEEE, 1633–1636.
- [67] Whalen, M., G. Gay, D. You, M. Heimdahl, and M. Staats, 2013: Observable modified condition/decision coverage. *Proceedings of the 2013 Int'l Conf. on Software Engineering*, ACM.
- [68] Yoo, S. and M. Harman, 2010: Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, **83**, no. 4, 689 – 701, doi:<http://dx.doi.org/10.1016/j.jss.2009.11.706>
URL <http://www.sciencedirect.com/science/article/pii/S0164121209003069>
- [69] Yu, Z., M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, 2017: Test case generation for program repair: A study of feasibility and effectiveness. *CoRR*, **abs/1703.00198**.
URL <http://arxiv.org/abs/1703.00198>

How to cite this article: Alireza Salahirad, Hussein Almulla, and Gregory Gay (2017), Choosing The Fitness Function for the Job: Automated Generation of Test Suites that Detect Real Faults, *Software Testing, Verification and Reliability*, 2018;00:1–XX.

Detecting Real Faults in the Gson Library Through Search-Based Unit Test Generation

Gregory Gay

University of South Carolina, Columbia, SC, USA,^{**}
greg@greggay.com

Abstract. An important benchmark for test generation tools is their ability to detect *real faults*. We have identified 16 real faults in Gson—a Java library for manipulating JSON data—and added them to the Defects4J fault database. Tests generated using the EvoSuite framework are able to detect seven faults. Analysis of the remaining faults offers lessons in how to improve generation. We offer these faults to the community to assist future research.

Keywords: Search-based test generation, automated test generation, software faults

1 Introduction

Automation of unit test creation can assist in controlling the cost of testing. One promising form of automated generation is *search-based* generation. Given a measurable testing goal, powerful optimization algorithms can select test inputs meeting that goal [6].

To impact practice, automated generation techniques must be effective at detecting the complex faults that manifest in real-world software projects [2]. “Detecting faults” is not a goal that can be measured. Instead, search-based generation relies on *fitness functions*—based on coverage of code structures, synthetic faults, and other targeted aspects—that are believed to increase the probability of fault detection. It is important to identify which functions produce tests that detect real faults.

By offering case examples, fault databases—such as Defects4J [5]—allow us to explore questions like those above. The Google Gson library [1] offers an excellent opportunity for expanding Defects4J. Gson is an open-source library for serializing and deserializing JSON input that is an essential tool of Java and Android development and is one of the most popular Java libraries [4].

Gson serves as an interesting benchmark because much of its functionality is related to the parsing of JSON input and creation and manipulation of complex objects. Manipulation of complex input and non-primitive objects is challenging for automated generation. Gson is also a mature project. Its faults will generally be more complex than the simple syntactic mistakes modeled by mutation testing [2]. Rather, detecting faults will require specific, contextual, combinations of input and method calls. By studying these faults, we may be able to learn lessons that will improve test generation tools.

^{**} This work is supported by National Science Foundation grant CCF-1657299.

¹ <https://github.com/google/gson>

We have identified 16 real faults in the Gson project, and added them to Defects4J. We generated test suites using the EvoSuite framework [6]—focusing on eight fitness functions and three combinations of functions—and assessed the ability of these suites to detect the faults. Ultimately, EvoSuite is able to detect seven faults. Some of the issues preventing detection include a need for stronger coverage criteria, the need for specific data types or values as input, and faults that only emerge through class interactions—requiring system testing to detect. We offer these faults and this analysis to the community to assist future research and improve test generation efforts.

2 Study

In this study, we have extracted faults from the Gson project, gathering faulty and fixed versions of the code and developer-written test cases that expose each fault. For each fault, we have generated tests for each affected class-under-test (CUT) with the EvoSuite framework [6]—using eight fitness functions and three combinations of functions—and assessed the efficacy of generated suites. We wish to answer the following research questions: (1) *can suites optimizing any function detect the extracted faults?*, (2) *which fitness function or combination of functions generates suites with the highest overall likelihood of fault detection?* and (3), *what factors prevented fault detection?*

In order to answer these questions, we have performed the following experiment:

1. **Extracted Faults:** We have identified 16 real faults in the Gson project, and added them to the Defects4J fault database (See Section 2.1).
2. **Generated Test Cases:** For each fault, we generated 10 suites per fitness function and combination of functions, using the fixed version of each CUT. We repeat this step with a two-minute and a ten-minute search budget per CUT (See Section 2.2).
3. **Removed Non-Compiling Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (See Section 2.2).
4. **Assessed Fault-finding Efficacy:** For each budget, function, and fault, we measure the likelihood of fault detection. For each undetected fault, we examined gathered data and the source code to identify possible detection-preventing factors.

2.1 Fault Extraction

Defects4J is an extensible database of real faults extracted from Java projects [5]. Currently, the core dataset consists of 395 faults from six projects, with an experimental release containing 597 faults from fifteen projects.² For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose each fault, and a list of classes and lines of code modified to fix the fault.

We have added Gson to Defects4J. This process consisted of developing build scripts that would compile and execute all tested project versions, extracting candidate faults using Gson’s version control and issue tracking systems, ensuring that each candidate could be reliably reproduced, and minimizing the “patch” used to distinguish fixed and faulty classes until it only contains fault-related code. Following this process, we extracted 16 faults from a pool of 132 candidate faults that met all requirements.

² Core: <http://defects4j.org>, Experimental: <http://github.com/Greg4cr/defects4j>

Each fault is required to meet three properties. First, the fault must be related to the source code. The “fixed” version must be explicitly labeled as a fix to an issue³, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix to the fault must be isolated from unrelated code changes such as refactoring.

The faults used in this study can be accessed through the experimental version of Defects4J⁴. Additional data about each fault can be found at <http://greqgay.com/data/gson/GsonFaults.csv>, including commit IDs, fault descriptions, and a list of triggering tests. We plan to add additional faults and improvements in the future.

2.2 Test Generation and Removal

EvoSuite applies a genetic algorithm in order to evolve test suites over several generations, forming a new population by retaining, mutating, and combining the strongest solutions [6]. In this study, we used EvoSuite version 1.0.5 with eight fitness functions: Branch Coverage, Direct Branch Coverage, Line Coverage, Exception Coverage, Method Coverage, Method (Top-Level, No Exception) Coverage, Output Coverage, and Weak Mutation Coverage. Rojas et al. provide a primer on each [6]. We have also used three combinations of fitness functions: all eight of the above, Branch/Exception Coverage, and Branch/Exception/Method Coverage. The first is EvoSuite’s default configuration, and the other two were found to be generally effective at detecting faults [3]. When a combination is used to generate tests, the individual fitness functions are calculated and added to obtain a single fitness score.

Tests are generated from the fixed version of the system and applied to the faulty version in order to eliminate the oracle problem. Given the potential difficulty in achieving coverage over Gson classes, two search budgets were used—two and ten minutes, a typical and an extended budget [2]. As results may vary, we performed 10 trials for each fault, fitness function, and budget. Generation tools may generate flaky (unstable) tests [2]. We automatically remove non-compiling test cases. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, less than 1% of tests are removed from each suite.

3 Results and Discussion

In Table I, we list—for each search budget and fitness function—the likelihood of fault detection (the proportion of suites that detected the fault). Seven of the sixteen faults were detected. EvoSuite failed to generate test suites for Fault 12. At the two minute budget, the most effective fitness function is a combination of Branch/Exception/Method Coverage, with an average likelihood of fault detection of 40.67%—closely followed by the Branch/Exception combination and Branch Coverage alone. At the ten minute

³ The commit message for the “fixed” version must reference either a reported issue or a pull request that describes and fixes a fault (that is, it must not add new functionality).

⁴ These faults will be migrated into the core dataset following additional testing and study.

Fault	Budget	BC	DBC	EC	LC	MC	M(TLNE)	OC	WMC	C-All	C-BE	C-BEM
2	2m	100.00%	100.00%	70.00%	70.00%	-	-	-	100.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	40.00%	90.00%	-	-	-	100.00%	100.00%	100.00%	100.00%
3	2m	70.00%	60.00%	-	80.00%	-	-	-	60.00%	30.00%	90.00%	70.00%
	10m	100.00%	80.00%	-	100.00%	-	-	-	100.00%	70.00%	90.00%	100.00%
6	2m	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
8	2m	20.00	30.00%	-	50.00%	-	-	-	10.00%	10.00%	10.00%	40.00%
	10m	90.00%	60.00%	-	100.00%	-	-	-	80.00%	80.00%	100.00%	90.00%
10	2m	100.00%	100.00%	30.00%	100.00%	20.00%	10.00%	40.00%	50.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	10.00%	100.00%	30.00%	10.00%	40.00%	70.00%	100.00%	100.00%	100.00%
13	2m	100.00%	100.00%	20.00%	30.00%	100.00%	100.00%	-	90.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	10.00%	-	100.00%	100.00%	-	100.00%	100.00%	100.00%	100.00%
16	2m	100.00%	100.00%	60.00%	100.00%	40.00%	10.00%	-	40.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	30.00%	100.00%	-	30.00%	30.00%	10.00%	100.00%	100.00%	100.00%
Average	2m	39.33%	39.33%	18.67%	35.33%	17.33%	14.67%	9.33%	30.00%	36.00%	40.00%	40.67%
	10m	46.00%	42.67%	12.67%	39.33%	15.33%	16.00%	11.33%	37.33%	43.33%	46.00%	46.00%

Table 1. Likelihood of fault detection for each fitness function (two-minute/ten-minute budget). (D)BC = (Direct) Branch Coverage, EC = Exception Coverage, LC = Line Coverage, M(TLNE)C = Method (Top-Level, No Exception) Coverage, OC = Output Coverage, WMC = Weak Mutation Coverage, C-All = combination of all criteria, C-BE = combination of BC/EC, C-BEM = combination of BC/EC/MC. Undetected faults (1, 4, 5, 7, 9, 11, 14, and 15) are omitted.

budget, these three configurations perform equally, with an average detection likelihood of 46.00%. Unlike in other Defects4J systems [3], Exception Coverage does not add significant value. Specialized metrics, like Output Coverage, also do not seem to have much situational applicability.

Fault 6 was detected the most reliably, regardless of search budget or fitness function. This fault updates Gson to be compliant with the 2014 JSON RFC 7159 standard, and adds a leniency check to enable backwards compatibility[5]. Compliance checks are spread throughout the code, resulting in fault detection if even a small amount of coverage is attained. Fault 13[6], dealing with an index out of bounds error, is a classic example of what automated generation excels at. The fix adds boundary checks, which are efficiently covered by Branch Coverage—ensuring differing output between versions.

Fault 8[7] was detected the least reliably. This fault causes issues with deserializing map structures when a key is an unquoted long or integer. The generated tests arguably expose the fault—they produce differing behavior and result in the same exception as the human-written test cases. Yet, these failing tests also point out an issue with test suite generation. The test cases fail in the same manner as the human-written cases, but not for the same reason. The failing tests pass strings to methods meant to handle long or integer values and expect a `NumberFormatException`—which is not thrown by the faulty version. The exception thrown instead—a complaint about a string—makes sense, given the input used. Rather than helping a human tester identify actual issues, these test cases only show that the two versions of the code behave differently.

EvoSuite failed to detect the other eight faults. Therefore, our next step was to examine these faults to identify factors preventing detection. These factors include:

Stronger Adequacy Criteria are Required: Fault 14[8] causes all instances of `-0` (“negative 0”) to be converted to 0. Catching this fault would require the generation

⁵ <https://github.com/google/gson/commit/af68d70cd55826fa7149effd7397d64667ca264c>

⁶ <https://github.com/google/gson/commit/9e6f2bab20257b6823a5b753739f047d79e9dcdb>

⁷ <https://github.com/google/gson/commit/2b08c88c09d14e0b1a68a982bab0bb18206df76b>

⁸ <https://github.com/google/gson/commit/9a2421997e83ec803c88ea370a2d102052699d3b>

framework to produce `-0` as input—an unlikely choice. However, the test generator could be guided towards this input. The fixed version of the class has a complex `if`-condition that includes this corner case. Branch Coverage simply requires the full predicate to evaluate to `true` and `false`, so coverage can be achieved without `-0` input. However, a stronger criterion such as Modified Condition/Decision Coverage [11] would require `-0` to attain full coverage.

Specific Data Types are Required as Input: Fault 9⁹ causes an error when Gson attempts to initialize an interface or abstract class. This fault can only be detected if a test case attempts to instantiate either type of object. Most generation frameworks will not attempt this, and the feedback provided by criteria like Branch Coverage is not sufficient to suggest such an action.

Fault Emerges Through Class Interactions and System Testing: Fault 1¹⁰ causes Gson to fail to serialize or deserialize a class when its super class has a type parameter. Like Fault 9, this is a case where tests would need to attempt to generate a highly specific object. In addition, the developer-written test exposing this fault is a *system-level* test, not a unit test—working through Gson’s top-level serialization and deserialization functions. It is possible that unit testing could expose the fault, but this is code that—like Fault 9 above—that would be hard to cover. System testing is more likely to expose the fault, but external context would still be needed to guide data type selection.

By default, Gson converts application classes to JSON using its built-in type adapters. If Gson’s default JSON conversion isn’t appropriate for a type, users can specify their own adapter using an annotation. Fault 5¹¹ deals with ensuring that custom type adapters safely handle null objects. However, performing unit testing of the modified class will not expose the fault. Rather, one needs to define a type adapter for a null class, then use Gson’s top-level API. Fault 7¹² modifies the same class, fixing a null pointer exception when a null object is returned instead of a proper `TypeAdapter`. A similar scenario exists for Fault 11¹³, where custom adapters are ignored for primitive fields. In all three cases—as long as the right input is chosen—system testing will expose this fault while unit testing may not be able to replicate the same example. However, system testing alone will still not be sufficient. Each of these scenarios requires external context to create the specific conditions called for to detect the fault.

Gson is a complex system designed to be accessed through a simple API. Human-written tests tend to use that API, even when testing specific classes. Unit test generation may not be suited to detecting some of the faults that emerge from this type of system, and even if it can, the generated test suites may not be easily understood by human developers. Many of the most mature test generation approaches are based on unit testing, and more work clearly needs to be conducted in the system testing realm.

Regardless of the form of testing, better means are needed of extracting *context* from the system and its associated artifacts. Automation requires information to guide test creation. Often, this is some form of code coverage. However, code coverage doesn’t

⁹ <https://github.com/google/gson/commit/0f66f4fac441f7d7d7bc4afc907454f3fe4c0faa>

¹⁰ <https://github.com/google/gson/commit/c6a4f55d1a9b191dbbd958c366091e567191ccab>

¹¹ <https://github.com/google/gson/commit/57b08bbc31421653481762507cc88ee3eb373563>

¹² <https://github.com/google/gson/commit/dea305503ad8827121e8212248c271f1f2f90048>

¹³ <https://github.com/google/gson/commit/bb451eac43313ae08b30ac0916718ca00c39656d>

provide the same type of information developers use during test creation, and many of the studied faults were detected by *almost any* coverage criterion or *no* criterion. Rather, information from the project is needed to guide input generation. Methods of gleaning that information, either through seeding from existing test cases or data mining of project elements, may assist in improving the efficacy of test generation. Approaches to mining of requirements information or bug reports, for instance, might suggest using particular data types or values as input.

4 Conclusion

Testing costs can be reduced through automated unit test generation. An important benchmark for such tools is their ability to detect *real faults*. We have identified 16 real faults in Gson, and added them to Defects4J. We generated test suites and found that EvoSuite is able to detect seven faults. Some of the issues preventing fault detection include a lack of fitness functions for stronger coverage criteria, the need for specific data types or values as input, and faults that only emerge through class interactions—requiring system testing rather than unit testing to detect. We offer these faults to the community to assist future research.

References

1. Chilenski, J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C. (April 2001)
2. Gay, G.: The fitness function for the job: Search-based generation of test suites that detect real faults. In: Proceedings of the International Conference on Software Testing. ICST 2017, IEEE (2017)
3. Gay, G.: Generating effective test suites by combining coverage criteria. In: Proceedings of the Symposium on Search-Based Software Engineering. SSBSE 2017, Springer Verlag (2017)
4. Idan, H.: The top 100 java libraries in 2017 - based on 259,885 source files (2017), <https://blog.takipi.com/the-top-100-java-libraries-in-2017-based-on-259885-source-files/>
5. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), <http://doi.acm.org/10.1145/2610384.2628055>
6. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 9275, pp. 93–108. Springer International Publishing (2015). http://dx.doi.org/10.1007/978-3-319-22183-0_7

Investigating Faults Missed by Test Suites Achieving High Code Coverage

Amanda Schwartz, Daniel Puckett

University of South Carolina Upstate

aschwarz2@uscupstate.edu, dpuckett@email.uscupstate.edu

Ying Meng, Gregory Gay

University of South Carolina

ymeng@email.sc.edu, greg@greggay.com

Abstract

Code coverage criteria are commonly used to determine the adequacy of a test suite. However, studies investigating code coverage and fault-finding capabilities have mixed results. Some studies have shown that creating test suites to satisfy coverage criteria has a positive effect on finding faults, while other studies do not. In order to improve the fault-finding capabilities of test suites, it is essential to understand what is causing these mixed results. In this study, we investigated one possible source of variation in the results observed: fault type. Specifically, we studied 45 different types of faults and evaluated how effectively human-created test suites with high coverage percentages were able to detect each type of fault. Our results showed, with statistical significance, there were specific types of faults found less frequently than others. However, improvements in the formulation and selection of test oracles could overcome these weaknesses. Based on our results and the types of faults that were missed, we suggest focusing on the strength of test oracles along with code coverage to improve the effectiveness of test suites.

Keywords: Code Coverage, Automated Testing, Software Testing, Test Suite Effectiveness

1. Introduction

In order to ensure software quality, it is essential that the software is tested thoroughly. However, what exactly constitutes *thorough* testing can be subjective. As developers lack knowledge of the faults that may reside in their systems, guidance and a means of judging test suite *adequacy* is required. Currently, one of the most popular ways to evaluate the adequacy of a test suite is through the use of code coverage criteria.

Code coverage criteria evaluate test suites by examining how well they cover structural elements such as functions, statements, branches, and/or conditions of a software system [1]. Each criterion establishes a set of test obligations over the class-under-test (CUT) that must be fulfilled in order to satisfy the criterion. Coverage criteria are commonly used in both academic research and industry, as they are easy to understand, establish clear guidelines and stopping conditions for testing, and are well-supported across a variety of programming languages [2]. Consequently, the confidence in code coverage being a proper method of evaluating test suites has become high in the software testing community. In fact, the confidence is so high that in some domains, such as avionics, evaluating test suites through the use of code coverage is legally required [3, 4]. Many research studies also validate proposed techniques by their ability to achieve some level of code coverage (e.g. [5, 6]).

Contrary to the widespread use and acceptance of code coverage being an adequate measure of test suite effectiveness, studies investigating the relationship between code coverage and fault-finding capabilities do not consistently support this. Some studies have shown that generating test suites to satisfy code coverage criteria has a positive effect on finding faults (e.g. [7, 8, 9]), while other studies do not (e.g. [10, 11]). To better evaluate whether code coverage is a proper method of evaluating test suites, it is important to understand why there are such differences in these findings.

In order to accept high code coverage as an indicator for a test suite's ability to find faults, there should be consistent evidence that test suites with high

code coverage are capable of finding more faults than test suites with lower code coverage. However, since the research does not always support this, it is important to investigate whether there are particular factors that affect the ability of a test suite achieving high code coverage to find faults. Unfortunately, very little work has been done in this area. Few studies were found [12, 4, 13] that identify factors that influence the relationship between code coverage and fault detection. These studies provide some insight—particularly around the influence of program structure—but cannot completely explain the different findings in the studies investigating code coverage and fault detection. More research needs to be conducted to investigate this important issue.

In our previous work [14], we began to investigate the impact particular *fault types* had on the relationship between code coverage and fault finding effectiveness, as modeled through the use of mutation testing—the seeding of synthetic faults into the CUT. Specifically, we were interested in whether there were particular fault types that went undetected more frequently than other fault types when programs are evaluated by test suites that achieve high code coverage. Our research showed that the rate of fault detection varied significantly according to fault type. We also noticed that there were certain types of faults consistently found less frequently than others. These were interesting findings that could inspire future research on why these particular fault types were found less frequently. However, this study was limited in two ways. First, only class-level mutation operators were considered. Second, there were many mutation operators that did not produce enough mutants to have enough data to perform a statistical analysis or form any solid conclusions. We address these limitations in this paper. Specifically, this paper makes the following contributions:

- The paper is extended to consider 19 Traditional Mutant Operators in addition to the original 26 Class Level Mutation Operators considered in our previous work [14].
- An additional 25,100 Class-Level mutants were created to supplement the 15,834 Class-Level mutants created in our previous work, for a total of

40,934 Class-Level mutants.

- A total of 122,985 Traditional mutants were created and analyzed.
- Statistical tests were performed and presented to determine whether there is a statistical significance to the faults that go undetected more frequently than other faults.
- A discussion of the fault types identified as outliers by the statistical tests is included.
- A suggestion, based on the nature of the faults identified as outliers, on how to improve test suites and test suite evaluation techniques is provided.

Our results identified that two types of mutants were found disproportionately often—AORB (Arithmetic Operator Replacement) and ROR (Relational Operator Replacement). However, four types of mutants were found less often than expected: AOIS (Arithmetic Operator Insertion), PCI (Type Case Operator Insertion), EAM (Accessor Method Change), and AODU (Arithmetic Operator Deletion). Test oracles that more thoroughly inspect internal state would aid in revealing such faults. Ultimately, code coverage alone does not ensure that faults are triggered and detected, and the selection of input and oracle have a dual influence on the effectiveness of a test suite. More attention should be given to the thoroughness of the selected oracle, and to the variables that are monitored and checked by the oracle.

The rest of the paper is organized as follows. Section 2 presents background information and related work. Section 3 explains our experimental procedures. Section 4 presents the results of our experiment. A discussion of our results is presented in Section 5. And finally, conclusions are discussed in Section 6.

2. Background and Related Work

Software testing is extremely important to the development process as the means of ensuring that software has the correct functionality and is not defec-

tive. However, software testing can be very time consuming and costly. Therefore, a significant amount of time and effort has been spent to identify ways to reduce the cost of software testing. Much of this attention has been spent researching automated software testing procedures. As a result, many different testing platforms and methods have been proposed, developed, and evaluated (e.g.[15, 16, 17]) and a great deal of progress has been made on reducing the time necessary for software testing. By reducing time, the hope is it will also reduce cost. However, the cost will only be reduced if testing continues to be effective at finding faults. Missed faults are extremely costly, and any reduction in cost resulting from the decreased time would be lost with the increase of costs associated with missed faults. Therefore, automated testing methods need to be evaluated to be sure they are effective at finding faults.

The most commonly used metric to evaluate automated test suites is to use some form of code coverage criteria. Coverage criteria reports a percentage according to how much source code is executed by the test suites. The exact calculation depends on the coverage method used. For example, line coverage is a very simple coverage metric that simply reports the percentage of lines of code that are covered by the test suite. Another popular coverage metric, branch coverage, adds an additional requirement that at each conditional both the true path and the false path will be executed.

Coverage criteria has become widely accepted in the software testing community as an adequate measure of test suite effectiveness [2]. It is used to validate new automated testing methods (e.g. [18, 19, 20]), used to compare testing methods based on level of coverage (e.g. [21, 22, 23]), and is used in many domains to determine whether a test suite is adequate. In fact, in some safety critical domains, such as avionics, code coverage is legally required to determine the adequacy of a test suite [3].

Since coverage criteria is frequently the standard for many in terms of evaluating test suites, it is important to make sure code coverage *actually* is a good indicator of test suite effectiveness. Some recent research has been conducted to evaluate whether increasing code coverage also increases a test suite's ability

to find faults. The results of this research has been mixed. Some studies show achieving high code coverage is a good indicator for fault-finding capabilities, while other studies do not.

Studies which provide support for coverage criteria being an adequate measure of test suite effectiveness show a correlation between code coverage and fault-finding capabilities. For example, early work by Frankl and Weiss [24] shows a correlation between test suite effectiveness and all-use and decision coverage criteria. Cai and Lyu [25] found a moderate correlation between fault-finding capabilities and four different code coverage criteria. Frate et. al [26] report a study which finds a higher correlation between test suite effectiveness and block coverage than there is between test suite effectiveness and test suite size. Gligoric et. al [27] studied a total of 26 programs and found that test suite effectiveness was correlated with coverage, and reported branch coverage as performing the best. A recent study by Kochhar et. al [28] evaluated two industrial programs and found a correlation between code coverage and faults detected. In our past work examining the factors that indicated a high likelihood of fault detection, we found that high levels of code coverage had a stronger correlation to the likelihood of fault detection than the majority of the other measured factors [9]. However, we also found that coverage alone was not enough to ensure fault detection.

Even though there are a number of studies that show a correlation between code coverage and fault-finding effectiveness, there are also many studies which do not. In previous work, we reported that satisfying code coverage alone was a poor indication of test suite effectiveness when suites are generated specifically to achieve coverage [10, 29]. We studied the fault-finding effectiveness of automatically generated test suites that satisfied five code coverage criteria (branch, decision, condition, MC/DC, and Observable MC/DC) and compared them to randomly generated test suites of the same size for five different production avionics systems. We found that, for most criteria, test suites automatically generated to achieve coverage performed *significantly worse* than random test suites of equal size which did not work to achieve high coverage. We did find

that coverage had some utility as a *stopping criterion*, however. Randomly-generated test suites that used coverage as a stopping criterion outperformed equally-sized suites generated purely randomly. The results of this study confirm that satisfying code coverage criteria has the potential to find faults, but there are factors (i.e. program structure, types of faults, and choice of variables monitored by the test oracle) that affect a test suite’s ability to find them even when high code coverage is achieved.

Another group of studies have investigated an additional factor: the size of the test suite, and how it interacts with both code coverage and suite efficacy. Inozemtseva and Holmes [11] performed an empirical study investigating the relationship between coverage and fault-finding capabilities and observed the difference between the results when size was controlled and when it was not controlled. They noticed that when size was not controlled for a test suite there was a moderate to high correlation between coverage and fault-finding effectiveness. However, when size was controlled they saw the correlation drop significantly. They suggest that the effectiveness is not correlated with coverage, but instead with the size of the test suite. A similar study by Namin and Andres [8] studied the relationship between size, code coverage, and fault-finding effectiveness. They found that coverage is only *sometimes* correlated with effectiveness when the size of the test suite is controlled. They also noted that no linear relationship existed between the three variables. This directly contradicts the findings of Frate et. al [26] mentioned previously that found a higher correlation between test suite effectiveness and block coverage than between test suite effectiveness and test suite size.

These opposing results make this an interesting problem to solve. They indicate there are factors that are not yet discovered which prevent test suites from finding faults even when high code coverage is achieved. Unfortunately, very little work has been done to understand why these studies report such different findings. We [12, 4] previously investigated the effect program structure had on the ability of test suites satisfying the MC/DC coverage criterion to find faults. Specifically, we compared the difference in test obligations and

the resulting test suites when implementation is varied between a small number of complex, inlined expressions and a larger number of simple expressions. We found that the MC/DC criterion is highly sensitive to the different implementations, and reported that MC/DC satisfaction requires significantly more test cases—and generally, more complex test cases—on inlined implementations than it did on the non-inlined version of the same implementation. The use of an inlined implementation produces robust test suites that are significantly more adept at detecting faults. More recently, Zhang and Mesbah investigated the influence test assertions have on code coverage and test suite effectiveness [13]. They suggested *assertions* are the underlying reason behind the strong correlation between test suite size, code coverage, and test suite effectiveness. The results of their study provide empirical evidence that assertion quantity and assertion coverage are strongly correlated with a test suite’s effectiveness and the correlation between statement coverage and test suite effectiveness decreases dramatically when assertion coverage is controlled.

These studies identify two specific factors apart from code coverage that influence test suite effectiveness and provide empirical evidence that code coverage alone is not always a good indicator of test suite effectiveness. Additional factors which affect the effectiveness and/or additional test suite evaluation metrics should be proposed and studied. Perez et. al [30] proposed a test evaluation metric which incorporates not only *if* a component is covered by a test suite, but also *how* it is covered.

Our paper advances this area of research by investigating the impact fault type has on the relationship between code coverage and test suite effectiveness. Our study reveals specific fault types that frequently go undetected even though a test suite achieves high code coverage. Based on the nature of these faults we suggest an area of future research that could improve a test suite’s ability to find these types of faults and increase the overall effectiveness of a test suite.

3. Study

In the preceding sections, the variability in studies investigating the fault-finding effectiveness of test suites meeting high levels of code coverage was discussed. In this work we conduct an empirical study to investigate whether fault type could be a contributing factor to the variability shown in the studies. Specifically, we answer the following research questions:

1. Are there particular fault types that go undetected more frequently by developer-written test suites achieving high code coverage?
2. Are there particular fault types that are detected with disproportionate frequency by developer-written test suites achieving high code coverage?

To answer these questions, we have performed the following steps:

1. Selected programs with large code bases and developer-written test suites achieving at least 80% code coverage (Section 3.1).
2. Generated mutants—synthetic faults—for all classes within each program (Section 3.2).
3. Executed the developer-written test suite against each mutant (Section 3.3).
4. Collected data on fault detection (Section 3.4).

In the following sections, we explain the procedures we followed for our experiment.

3.1. Selection of Object Programs

To begin the experiment, we needed to select our object programs. We selected these programs based on the following specific criteria:

1. A developer-written test suite must exist for the program.
2. This test suite must achieve at least 80% statement coverage over the program code.
3. The program must have at least 10,000 lines of code.

Table 1: Experiment Objects and Associated Data

	Commons Compress	Joda Time	Commons Lang	Commons Math	JSQL Parser
Thousand Lines of Code (KLOC)	11	14	13	49	10
Number of Test Cases	556	2157	3526	6248	315
Statement Coverage (%)	83	89	93	89	82
Branch Coverage (%)	81	81	90	85	76
Number of Mutation Faults	24,194	43,660	14,917	42,936	3,510
Commons CLI					
Thousand Lines of Code (KLOC)	7	19	6	20	356
Number of Test Cases	408	648	303	887	13,341
Statement Coverage (%)	96	81	94	93	87
Branch Coverage (%)	94	77	89	90	80
Number of Mutation Faults	1,462	3,738	1,083	4,354	12,202

Table 1 lists the objects of analysis along with the following information for each program: the number of thousands of lines of code (KLOC), the number of test cases, line coverage percentage, branch coverage percentage, and number of mutation faults. Number of lines of code are counted without including commented and whitespace lines. We ran both Cobertura ¹ and ECLEmma ² coverage tools to obtain coverage information. For each of the programs, these tools reported within less than a percent of each other. The numbers reported in Table 1 are therefore an average of those numbers rounded to the nearest percent.

The programs we chose all have existing JUnit test suites. The JUnit test suites are developer-written (as opposed to test suits that are generated by automated test generation tools), which reflects common practice in industry. This also means the tests are most closely tailored for each specific program. We are focused on developer-written suites, as automatically-generated suites typically are very different than developer-written suites [31, 32]. Automatically-generated suites often use shorter sequences of test steps [10], choose inputs that do not resemble those chosen by humans [31], follow sequences of events that differ from those chosen by humans [33], and may be verified with gen-

¹cobertura.github.io/cobertura

²www.eclemma.org

erated test oracles that differ from those created by humans [33]. Therefore, focusing on automatically-generated test suites or mixing test creation methods would introduce a risk of conflated results. Instead, we focus in this study on developer-written tests and the typical strengths and weaknesses of such suites.

We wanted programs with existing test suites that satisfy high code coverage percentages. We required programs with high code coverage because many studies that found correlation between code coverage and fault-finding capabilities, only found that correlation when code coverage was high (e.g. [34, 24, 35, 36]). By choosing programs with high code coverage there is a better chance of finding the faults (as indicated by studies revealing there to be a correlation between high code coverage and fault detection). We wanted to have the highest chance of finding the fault so we could identify faults that were still going undetected. Faults going undetected in this situation could explain some variance in the findings of studies investigating this relationship. Based on the results of studies which found a correlation between coverage and fault detection at high coverage rates (e.g. [34, 35]), we chose 80% as the minimum target ³.

In order to provide an ample opportunity for a large number of faults to be inserted, all of our programs have over 6,000 non-commented, non-whitespace lines of code. This was necessary to collect enough fault data for statistical analysis. We chose ten medium-sized Java programs that met the criteria we were searching for. The programs range from 6,000 to 356,000 lines of code. Joda Time ⁴ is an opensource replacement for the date and time classes in Java versions previous to Java 8. The Apache Commons Math library ⁵ is an opensource Mathematics library containing Math and Statistics components.

³JSQL Parser and JSoup had statement coverage above 80%, but the branch coverage for these programs fell just below. Because the statement coverage met the threshhold and the branch coverage was very close, we decided to include these two programs as well. All other programs met or exceeded 80% for both statement and branch coverage.

⁴<http://joda-time.sourceforge.net/>

⁵<http://commons.apache.org/proper/commons-math/>

The Apache Commons Compress package⁶ is an opensource API that provides Java with the ability to work with file compression. The Apache Commons Lang API⁷ is an opensource library that provides helper utilities for the java.lang API. JSQLParser⁸ is an opensource API that parses SQL statements into a hierarchy of Java classes. The Apache Commons CLI API⁹ provides support for parsing command-line options passed to programs. The Jsoup HTML Parser allows Java programs to work with real-world HTML¹⁰. The Apache Commons CSV API reads and writes files in variations of the comma-separated value (CSV) format¹¹. The Apache Commons Codec API provides implementations of common encoders and decoders¹². Finally, the Closure Compiler is a JavaScript checker and optimizer, written in Java¹³.

3.2. Mutant Creation

To investigate whether certain types of faults consistently go undetected when tested with test suites satisfying high code coverage criteria, we needed to have programs with faults. More specifically, we needed to know the location and nature of each fault in order to determine whether the test suite was able to catch it, and which types of faults were getting caught less frequently than others.

Mutation testing is used in software testing to take the original program and modify it in small ways. This can be helpful for many different reasons. One main use of mutation testing is to evaluate the quality of software tests. First, small variations of the original program are created by inserting a known fault into the program. Then, the test suite will be run on the variation to see if it is able to catch that fault. This process provides an indicator of how effective the

⁶<http://commons.apache.org/proper/commons-compress/>

⁷<http://commons.apache.org/proper/commons-lang/>

⁸<http://jsqlparser.sourceforge.net/>

⁹<https://commons.apache.org/proper/commons-cli/>

¹⁰<https://jsoup.org/>

¹¹<https://commons.apache.org/proper/commons-csv/>

¹²<https://commons.apache.org/proper/commons-codec/>

¹³<https://github.com/google/closure-compiler>

test suite is at finding faults.

Mutation testing works by using *mutation operators* to create the small variations of the programs. Each mutation operator provides the mechanism to insert one specific type of fault. For example, the JSD (Static Modifier Deletion) operator changes class variables to instance variables by removing the static modifier. An example mutant is shown below.

Original:

```
public static int x = 100;
```

Mutated:

```
public int x = 100;
```

The previous example was just one example of a simple mutation operator. Many mutation operators have been proposed and studied in the literature (e.g. [37, 38]). Mutation operators are built with the goal of emulating common programming mistakes. Each small variation of the program created through the use of mutation operators are called *mutants*. Mutants can be created by making a single change to the program, or by making more than one change to the program. When mutants are created with multiple changes, the faults may interfere with each other and cause different results than if each fault was inserted individually in a mutant. For example, a test case which would fail with the insertion of one fault, may no longer fail when another fault is added. Similarly, a test case may not fail with a single fault, but fail with additional faults. This problem of fault interference and interaction has been studied and discussed in recent studies [39, 40]. To eliminate the threat of fault interference and to get accurate results according to each individual fault type, we created each mutant with a single fault.

In our experiment, we used mutation testing because of its ability to provide programs with a large number of known, categorized faults. Our experiment is concerned with the frequency at which faults are found, as well as the nature of these faults. Therefore, we used mutation testing software to create mutants with known fault location and type. Then, we could run the test suites over the

mutated programs and determine which ones were caught by the test suite and which ones were not. In recent research, there has been some concern whether conclusions drawn from studies using faults created through mutation operators can be generalized for real faults. In our study the concern would be whether the ability to find the seeded faults would be representative of a test suite's ability to find real faults. There is research to support that mutant detection is strongly correlated with real fault detection [41, 42]. Further, our study isn't focused solely on the exact mutant operator being missed, but instead on the nature of that type of fault. We are using the results of the study to be able to expand the knowledge base and answer questions such as: What problems do faults being missed cause? How can test suites be improved to find faults which cause that type of problem?

Our study uses Java programs, so we needed a mutation tool to be able to create mutants for Java programs. There are a few tools available to do this. The most popular mutation tools for Java include Pit¹⁴, Major [43], and MuJava [44]. Of the three, MuJava provided the most complete set of mutation operators, providing both traditional and class-level mutation operators. Therefore, we used MuJava to generate the mutants in this study. MuJava provides 19 traditional mutation operators and 28 class-level mutation operators [45]. There were two class-level mutation operators (EOA and IHD) that MuJava was unable to generate in any of our programs selected for this study, therefore we only used the remaining 26 class-level mutation operators in this study. The number of mutants created for each program is included in the last row in Table 1. A description of the traditional operators is provided in Table 2 and a description of the class-level mutation operators is provided in Table 3.

There are some mutant operators that would appear to be very specific to Java programs, or at least to object-oriented languages. For example, the IOD (Overriding Method Deletion) operator deletes an entire declaration of an overriding method in a subclass, so that it will use the parent's version of

¹⁴<http://pitest.org/>

Table 2: Traditional Mutation Operators and Descriptions

Mutation Operators	Description
AORB (Arithmetic Operator Replacement)	Replace basic binary arithmetic operators with other binary arithmetic operators.
AORS (Arithmetic Operator Replacement)	Replace short-cut arithmetic operators with other unary arithmetic operators.
AOIU (Arithmetic Operator Insertion)	Insert basic unary arithmetic operators.
AOIS (Arithmetic Operator Insertion)	Insert short-cut arithmetic operators.
AODU (Arithmetic Operator Deletion)	Delete basic unary arithmetic operators.
AODS (Arithmetic Operator Deletion)	Delete short-cut arithmetic operators.
ROR (Relational Operator Replacement)	Replace relational operators with other relational operators, and replace the entire predicate with true and false.
COR (Conditional Operator Replacement)	Replace binary conditional operators with other binary conditional operators.
COD (Conditional Operator Deletion)	Delete unary conditional operators.
COI (Conditional Operator Insertion)	Insert unary conditional operators.
SOR (Shift Operator Replacement)	Replace shift operators with other shift operators.
LOR (Logical Operator Replacement)	Replace binary logical operators with other binary logical operators.
LOI (Logical Operator Insertion)	Insert unary logical operator.
LOD (Logical Operator Delete)	Delete unary logical operator.
ASRS (Short-Cut Assignment Operator Replacement)	Replace short-cut assignment operators with other short-cut operators of the same kind.
SDL (Statement Deletion)	SDL deletes each executable statement by commenting them out. It does not delete declarations.
VDL (Variable Deletion)	All occurrences of variable references are deleted from every expression. When needed to preserve compilation, operators are also deleted.
CDL (Constant Deletion)	All occurrences of constant references are deleted from every expression. When needed to preserve compilation, operators are also deleted.
ODL (Operator Deletion)	Each arithmetic, relational, logical, bitwise, and shift operator is deleted from expressions and assignment operators. When removed from assignment operators, a plain assignment is left.

the method. However, there is still useful information that can be gained by understanding a test suite’s ability to catch this type of error. This type of error would be similar to just calling the wrong method. It would be considered a control flow error, as it is executing the wrong area of code. This type of problem can be caused in languages that are not object-oriented as well. For example, in functional languages it is quite common to put a function call as a parameter in another function call (and possibly nested a few times). The function calls could be accidentally transposed, calling the wrong method. This problem would be similar to the type of problem caused by the IOD operator. This example illustrates how—even though our experiment uses Java programs and mutant operators built for Java—the knowledge gained from our study could be useful in other circumstances as well.

3.3. Execute Test Suites Over Mutants

After the programs were chosen and mutants were created for them, we executed the entire test suite for each program against every mutant. Although

Table 3: Class Level Mutation Operators and Descriptions

Mutation Operators	Description
IHD (Hiding Variable Deletion)	The IHD operator will delete a variable in a subclass that has the same name and type as a variable in the parent class.
IHI (Hiding Variable Insertion)	The IHG operator inserts a variable of the same name as a variable in the parent scope, so as to "hide" the variable in the parent scope.
IOD (Overriding Method Deletion)	The IOD operator deletes an entire declaration of an overriding method in a subclass so it will use the parent's version.
IOP (Overriding method calling position change)	When a child class makes a call to a method it overrides in the parent class, the IOP operator will move that call to a different location in the method.
IOR (Overridden method rename)	The IOR operator renames the parent's version of an overridden method.
ISI (super Keyword Insertion)	The ISI operator inserts the super keyword so that a reference to a variable or method in a child class uses the parent variable or method.
ISD (super Keyword Deletion)	The ISD operator deletes occurrences of the super keyword so that a reference to a variable or method is no longer to the parent class' variable or method.
IPC (Explicit Call of a Parent's Constructor Deletion)	The IPC operator deletes super constructor calls, causing the possibility for a child object to not be initialized correctly.
PNC (New Method Call with Child Class Type)	The PNC operator causes an object reference to refer to an object of a different compatible type.
PMD (Member Variable Declaration with Parent Class Type)	The PMD operator changes the declared type of an object reference to the parent of the original declared type.
PPD (Parameter Variable Declaration with Child Class Type)	The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables.
PCI (Type Cast Operator Insertion)	The PCI operator changes the actual type of an object reference to the parent or child of the original declared type.
PCC (Cast Type Change)	The PCC operator changes the type that a variable is cast into.
PCD (Type Cast Operator Deletion)	The PCD operator deletes a type casting operator.
PRV (Reference Assignment with Other Compatible Types)	The PRV operator changes operands of a reference assignment to be assigned to objects of subclasses.
OMR (Overloading Method Contents Change)	The OMR operator replaces the body of a method with the body of another method that has the same name.
OMD (Overloading Method Deletion)	The OMD operator deletes overloading method declarations.
JTI (this Keyword Insertion)	The JTI operator inserts the keyword this.
JTD (this Keyword Deletion)	The JTD operator deletes uses of the keyword this.
JSI (static Modifier Insertion)	The JSI operator adds the static modifier to change instance variables to class variables.
JSD (static Modifier Deletion)	The JSD operator removes the static modifier to change class variables to instance variables.
JID (Member Variable Initialization Deletion)	The JID operator removes the initialization of member variables in the variable declaration.
JDC (Java-supported Default Constructor Deletion)	The JDC operator forces Java to create a default constructor by deleting the implemented default constructor.
EOA (Reference Assignment and Content Assignment Replacement)	The EOA operator replaces an assignment of an object reference with a copy of the object, using the Java clone() method.
EOC (Reference Comparison and Content Comparison Replacement)	The EOC operator changes an object reference check to object content comparison check through the use of Java's equal() method.
EAM (Accessor Method Change)	The EAM operator changes an accessor method name for other compatible accessor method names (where compatible means they have the same signature).
EMM (Modifier Method Change)	The EMM does the same as EAM except it works with modifier methods instead of accessor methods.
OAN (Argument Number Change)	The OAN operator changes the number of arguments in the method invocations, but only if there is an overloading method that can accept the new argument list.

recent studies [11, 8, 10] have reported size to be a factor in the effectiveness of test suites at finding faults, we did not try to limit the size of the test suite. We ran all tests across all mutants. If the test case found the seeded fault in the mutant, it is said to have *killed the mutant*. By running all tests (instead of a subset of tests that met the same coverage percentage), the chances of killing the mutant were higher. The goal of our study was to gain knowledge about particular fault types that go undetected most frequently. By revealing the faults that went undetected when all tests were executed, we successfully identify the most troublesome fault types.

3.4. Collect and Analyze Data

Finally, after running the test suites across all mutants, we completed the data collection. To be able to answer our research question, we needed to collect the following data: the number of mutants created *for each mutant operator* and the number of these mutants killed by the test suite. Because of the large number of mutants investigated in this study, in order to collect this data, we wrote some custom scripts to inspect the log files generated from running the JUnit test suites. Using these scripts we were able to determine, for every mutant, whether the bug was found (aka mutant killed) or not found. Then we could use this information and quickly tally totals for each mutation operator for each program.

3.5. Threats to Validity

This section discusses the construct, internal, and external threats to the validity of our study.

3.5.1. Construct Validity:

In our study, we measure fault-finding capabilities over seeded faults, rather than real faults. It is possible real faults could produce different results, even though recent studies [41, 42] show that the detection of mutation faults is strongly correlated with the detection of real faults. Motivationally, however,

we believe there is benefit in using mutations even if mutations do not perfectly map to real faults. Test suites that are effective at detecting a wide range of mutation types may be likely to be better at detecting real faults as well, simply because more effort went into creating a robust set of test cases. Guarding against mutations is guarding against additional potential shortcomings. If we can understand where developer-written suites tend to fall short, we can advise programmers on their practices and the creators of automated generation tools on where they can improve their work.

Further, the faults used in this study were limited to what MuJava was able to generate. Not all programs have mutants for every available mutant operator, and some mutant operators (EOA and IHD) were not generated for any of the studied programs. These operators were excluded from our results.

Mutation testing may produce equivalent mutants—mutants that always return the correct result, and are indistinguishable from normal programs. It is possible that some of the undetected faults are equivalent mutants, and detecting equivalency is often an undecidable problem. Therefore, we cannot state with certainty which of the undetected mutants were equivalent. However, in the majority of cases, the mutation types that were detected less often were not types that seem to be prone to producing equivalent mutants.

There is also the risk that the mutants generated could have been generated in the areas of code that are not covered by any tests in the test suite. These mutants would be recorded as not being killed, but potentially could have been killed if they had been in an area covered by the test suite. Although this has the potential to skew the results, we believe it is unlikely because we have chosen such a high coverage level and have excluded enough mutants from enough different classes to overcome any potential clustering of mutants in uncovered code.

3.5.2. Internal Validity:

The accuracy of the results of our study are dependent upon the accuracy of the tools used. We used tools to seed faults and measure coverage percentages.

We did use multiple tools to measure coverage and found the results to be very similar (varying less than one percent) so we are confident these results are accurate. Also, when tallying the total results, we wrote scripts to sift through the log files of each mutant in order to determine whether the fault was found or not. Although we did manually verify many of the logs for each program, there is still a possibility for a very small margin of error. Still, sifting manually through over 163,919 log files would have likely created a much larger margin of error as that would introduce more human error.

3.5.3. External Validity:

There are three main threats to external validity in our study. First, the programs we considered were all Java programs. Our results may not extend to programs written in other languages. Even though we explained in Section 3.2 that information *may* be gained from mutant operators built for Java in other programs as well, we do not have empirical evidence to quantify the extent to which this is true. Second, although the programs we chose were considerably larger than many of the previous studies mentioned in the Section 2, they are not large compared to industry systems. Our results may not generalize to these size systems. Last, the mutants created in our study were created by inserting a single fault into each mutant. We did this to eliminate the problem of fault interference that can occur with multiple faults [39, 40]. Because of fault interaction and interference, the results could be much different if multiple faults were inserted into each mutant.

4. Data and Analysis

In this section we present the results of our experiment. First, we will discuss the results for the traditional mutant operators, then we will discuss the results for the class-level mutant operators. Finally, we present the overall results and analyze our statistical findings.

Table 4: Traditional Mutant Results by Program

Mutant Operators	Commons Compress		Joda Time		Commons Lang		Commons Math		JSQl Parser	
	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
AORB	47	51.97%	1183	41.29%	789	33.94%	77	88.97%	25	68.00%
AORS	22	40.91%	134	8.21%	35	82.69%	347	16.43%	12	50.00%
AQIU	477	36.90%	1565	15.69%	619	71.54%	4094	69.32%	24	70.83%
AQIS	2370	34.51%	7852	22.82%	1865	66.49%	7626	72.21%	169	66.86%
AODU	8	37.50%	2241	2.05%	32	81.25%	382	64.14%	0	-
AODS	6	100.00%	1	100.00%	5	100.00%	68	41.18%	0	-
ROR	1830	43.50%	5333	27.68%	3479	73.93%	6269	71.93%	290	76.21%
COR	170	41.76%	231	60.17%	396	68.94%	382	85.34%	27	85.19%
COD	30	43.33%	18	77.78%	40	87.50%	163	66.87%	15	93.33%
COI	511	68.49%	1807	33.76%	942	93.31%	1492	70.84%	263	94.30%
SOR	26	100.00%	6	100.00%	0	-	64	18.75%	0	-
LOR	99	57.58%	24	25.00%	9	88.89%	267	31.84%	0	-
LOI	890	43.03%	2749	37.94%	1104	84.33%	3215	33.65%	48	75.00%
LOD	3	100.00%	4	0.00%	0	-	2	0.00%	0	-
ASRS	255	64.31%	144	18.75%	337	70.33%	784	84.44%	4	0.00%
SDL	1808	46.52%	6841	21.41%	2447	81.94%	4418	75.46%	1127	78.26%
VDL	262	45.04%	570	29.65%	256	74.22%	2880	82.92%	74	91.89%
CDL	219	31.96%	557	15.26%	161	62.11%	770	77.27%	104	87.50%
ODL	1036	35.91%	3993	13.00%	1161	68.99%	4803	88.24%	398	86.93%
Total	10509	43.20%	35253	24.44%	13677	76.41%	45647	74.01%	2570	80.82%
Mutant Operators	Commons CLI		JSoup		Commons CSV		Commons Codec		Closure	
	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
AORB	26	65.38%	109	99.08%	30	73.33%	198	87.88%	118	33.90%
AORS	1	100.00%	22	95.45%	6	82.33%	5	80.00%	13	46.15%
AQIU	38	73.68%	201	98.01%	62	88.71%	183	72.13%	139	43.88%
AQIS	92	68.48%	772	97.02%	154	91.56%	501	83.43%	499	26.85%
AODU	0	-	4	100.00%	0	-	0	-	7	0.00%
AODS	0	-	3	100.00%	0	-	5	100.00%	4	50%
ROR	278	71.58%	855	98.71%	142	80.99%	1071	79.55%	896	38.17%
COR	56	58.93%	134	94.78%	43	81.40%	135	82.22%	384	62.50%
COD	8	75.00%	22	100.00%	7	100%	24	87.5%	57	77.19%
COI	128	86.72%	199	100.00%	81	93.83%	299	92.64%	546	74.36%
SOR	0	-	0	-	0	-	4	100.00%	0	-
LOR	0	-	0	-	0	-	24	91.67%	14	64.29%
LOI	49	71.43%	295	98.31%	103	91.26%	311	78.78%	188	44.15%
LOD	0	-	0	-	0	-	0	-	0	-
ASRS	0	-	22	100.00%	4	0.00%	29	72.41%	9	11.11%
SDL	368	73.37%	554	95.49%	231	89.18%	898	76.39%	1645	58.78%
VDL	20	45.00%	47	100.00%	21	61.90%	82	71.95%	71	43.66%
CDL	31	19.35%	20	100.00%	18	38.89%	68	63.24%	74	35.14%
ODL	127	43.31%	268	97.76%	101	63.35%	375	70.67%	701	49.93%
Total	1222	68.17%	3527	97.65%	1003	83.95%	4212	79.27%	5365	51.11%

4.1. Traditional Mutants

MuJava provides 19 traditional mutant operators. A list of the 19 operators and a description of each is provided in Table 2. A total of 122,985 traditional mutants were created across the ten object programs. Table 4 provides a breakdown of the number of each individual traditional mutant that were created for each object program, as well as the percentage of those mutants that were killed by the test suite for that program.

The results show there is a wide variance in the kill rate between each mutant type for the traditional operators. Figure 1 provides a bar chart of the overall kill percentages (using the totals for all ten programs) for each traditional operator. There are two mutant operators with a kill rate of less than 25%: AODU (12%) and AORS (25%).

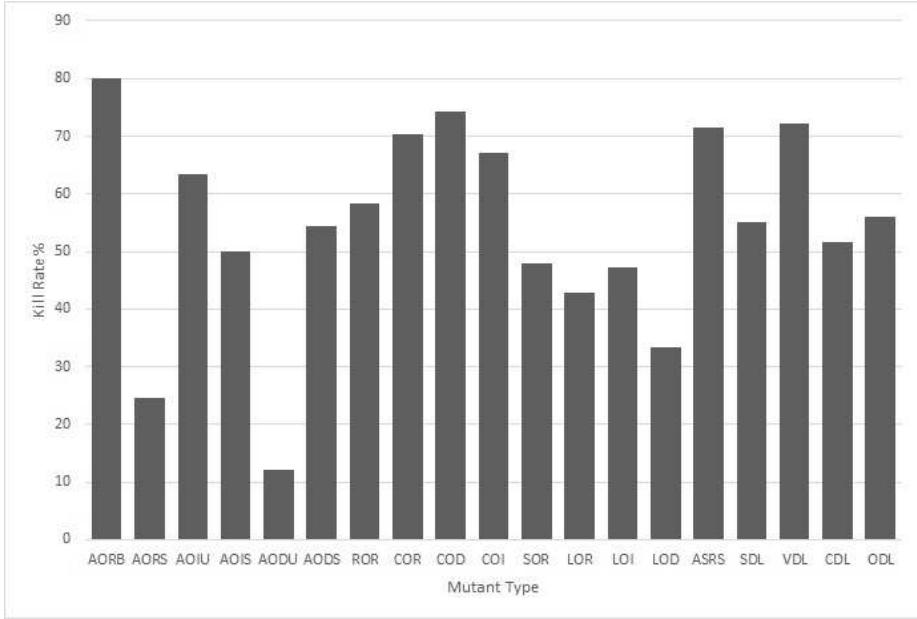


Figure 1: Traditional Mutants Kill Rates

4.2. Class-level Mutants

There are 26 class-level mutant operators that we considered in our experiment. A list of the operators and descriptions is provided in Table 3. A total of 40,934 class-level mutants were created across the ten object programs. Table 5 provides the number of mutants created for each operator for each program and the percentage of the mutants killed by the test suite for the program.

Like the traditional mutants, the kill rate for the class-level mutants vary greatly by mutant type. To quickly see which mutant types had the lowest kill rate overall, a bar graph is provided in Figure 2. The bar graph shows seven mutant operators with a kill rate of less than 20%: ISI (15%), JSD (10.09 %), PPD (0 %), IOR (6.63%), PMD (0 %), JID (17.41 %), and PNC (13.16 %).

4.3. Overall results

The bar graphs and tables provided in the previous sections show that there are some types of mutants found less frequently than others. The goal of this section is to see whether we can say *statistically* that these mutants are found

Table 5: Class Level Mutant Results by Program

Mutant Operators	Commons Compress		Joda Time		Commons Lang		Commons Math		JSQl Parser	
	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
EOC	3	0.00%	2	100%	6	33.33%	0	-	0	-
EMM	541	53.05%	28	85.71%	130	69.23%	65	27.69%	9	66.67%
OAN	5452	34.83%	3410	98.33%	146	89.73%	627	55.18%	2	0.00%
JDC	2	50.00%	1	100.00%	1	100.00%	5	80.00%	0	-
JSI	275	26.55%	17	29.41%	40	57.50%	217	19.35%	236	54.66%
ISI	20	0.00%	16	31.25%	0	-	24	16.67%	0	-
PCI	2635	0.00%	1363	47.25%	11	100.00%	317	64.35%	31	96.77%
PCC	11	0.00%	11	0.00%	0	-	13	92.31%	0	-
PCD	9	0.00%	8	37.50%	5	100.00%	9	88.89%	0	-
ISD	8	0.00%	8	12.50%	12	66.67%	11	63.64%	0	-
JSD	135	0.06%	43	20.93%	93	0.00%	259	9.65%	1	0.00%
EAM	702	25.07%	233	79.40%	260	84.62%	5524	33.76%	116	58.62%
PPD	0	-	0	-	0	-	4	0.00%	2	0.00%
IOR	84	0.00%	56	8.93%	0	-	24	25.00%	0	-
IOP	1	100.00%	0	-	0	-	8	50.00%	2	0.00%
PMD	19	0.00%	18	0.00%	2	0.00%	22	0.00%	0	-
PRV	409	14.18%	294	97.62%	80	97.50%	237	75.11%	55	85.45%
IOD	292	8.90%	196	39.80%	50	76.00%	225	17.78%	77	93.51%
JID	74	9.46%	8	87.50%	14	64.29%	23	8.7%	59	15.25%
JTI	104	48.08%	0	-	32	90.62%	128	26.56%	182	79.67%
JTD	53	67.92%	0	-	0	-	17	52.94%	161	83.83%
IPC	36	16.67%	10	30.00%	23	91.30%	78	62.82%	2	0.00%
OMD	28	3.57%	23	4.35%	13	84.62%	13	30.77%	1	0.00%
OMR	2775	3.24%	2662	98.08%	320	75.94%	1160	89.22%	4	0.00%
IHI	12	91.67%	0	-	0	-	80	82.50%	0	-
PNC	5	0.00%	0	-	2	100.00%	64	7.81%	0	-
Total	13685	19.89%	8407	85.93%	1240	74.35%	9154	43.34%	940	68.19%
Mutant Operators	Commons CLI		JSoup		Commons CSV		Commons Codec		Closure	
	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found	Total Mutants	Percent Found
EOC	0	-	1	0.00%	0	-	0	-	2	0.00%
EMM	8	75.00%	0	-	0	-	0	-	4929	34.79%
OAN	2	100.00%	11	63.64%	0	-	34	17.65%	10	60.00%
JDC	0	-	0	-	0	-	0	-	0	-
JSI	30	50.00%	20	40.00%	9	77.78%	15	40.00%	124	49.19%
ISI	5	-	14	0.00%	0	-	1	100.00%	1	100.00%
PCI	0	-	2	0.00%	0	-	0	-	307	68.40%
PCC	0	-	0	-	0	-	0	-	0	-
PCD	0	-	0	-	0	-	0	-	0	-
ISD	0	-	3	0.00%	1	100.00%	0	-	0	-
JSD	10	0.00%	3	0.00%	9	22.22%	35	0.00%	76	40.79%
EAM	96	42.71%	43	55.81%	38	65.79%	8	0.00%	997	48.04%
PPD	0	-	0	-	0	-	0	-	0	-
IOR	0	-	0	-	0	-	0	-	2	0.00%
IOP	0	-	3	0.00%	2	0.00%	0	-	3	0.00%
PMD	0	-	0	-	0	-	0	-	0	-
PRV	47	85.11%	7	57.14%	7	100.00%	0	-	135	82.96%
IOD	4	25.00%	30	26.67%	5	40.00%	12	33.33%	78	39.74%
JID	5	20.00%	2	0.00%	3	100.00%	4	25.00%	55	7.27%
JTI	21	47.62%	21	57.14%	5	0.00%	13	0.00%	58	46.55%
JTD	0	-	14	71.43%	0	-	1	0.00%	38	71.05%
IPC	1	100.00%	8	0.00%	0	-	6	0.00%	11	0.00%
OMD	0	-	0	-	0	-	0	-	0	-
OMR	16	31.25%	20	90.00%	1	0.00%	11	18.18%	10	30.00%
IHI	0	-	4	75.00%	0	-	0	-	1	100.00%
PNC	0	-	5	60.00%	0	-	0	-	0	-
Total	240	50.83%	211	45.97%	80	58.75%	140	14.29%	6837	39.61%

less often than other mutants. The percentages displayed in the bar graph can be deceiving because there are mutants on there with very high kill rates, but a low number of mutants created. We needed a way to determine if that high (or low) kill rate is really unusual, or just distorted because of the number of mutants created.

In statistics, “unusual” data points are called *outliers*—data points that are significantly different from other data points in the set. To determine the outliers in our data—the mutants with a significantly different kill rate than others—we

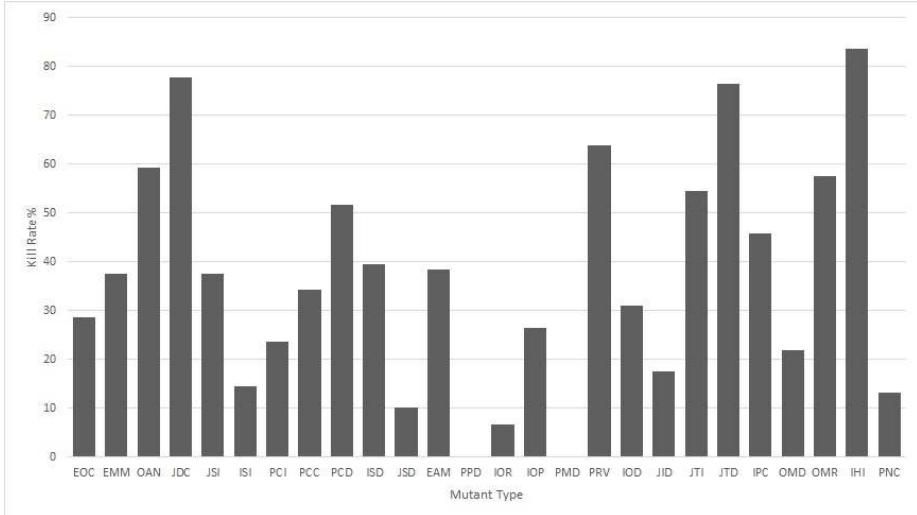


Figure 2: Class-Level Mutants Kill Rates

first completed a regression analysis, and then calculated studentized residuals.

Regression analysis can be used in statistics to estimate relationships among variables. In general, regression methods attempt to fit a model to observed data in order to quantify the relationship between two groups of variables. The fitted model can then be used to describe the relationship or predict new values. In our experiment, the data is the number of mutants created and the number of mutants killed. The regression analysis on our data provides, based on the data gathered, what the predicted—or *expected*—value would be for any number of mutants created for any mutant type. Figure 3 provides a scatterplot of our data. The line shows where the expected values would be. Specifically, it shows how many mutants would be expected to be killed (y-axis) according to the number of mutants that were created (x-axis).

The model created by regression analysis assumes that all mutant types are all killed at the same rate. However, we are trying to determine whether there are particular mutant types that are not killed at the same rate as others. To answer this question, we would need to determine which mutant types do not fit the regression model shown in Figure 3. The figure shows that many

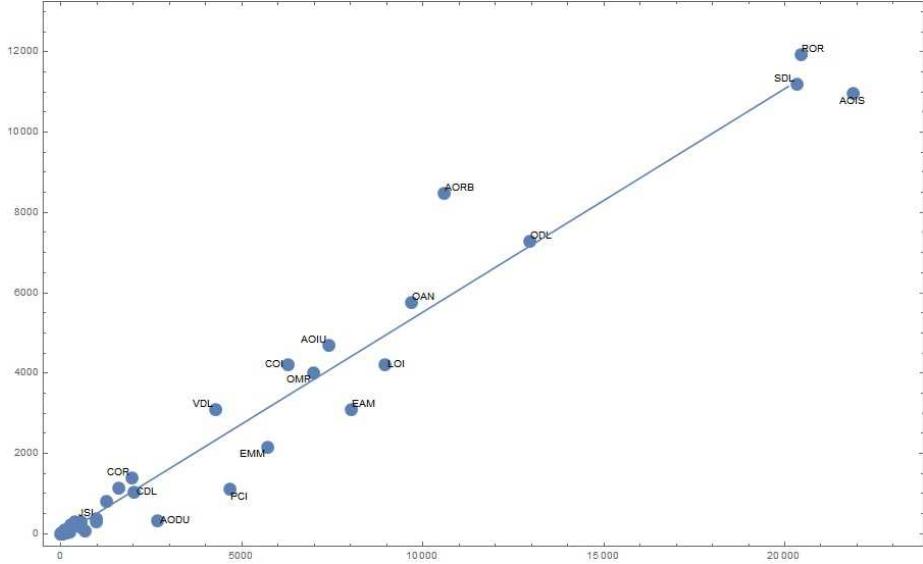


Figure 3: Scatter Plot of Mutant Types and Killed Mutants

of the mutant operators *do* follow that linear behavior. However, there are a few exceptions. AODU, PCI, EMM, EAM, LOI, and AOIS fall below the line, while AORB and ROR are above the line. The line shows how many mutants are expected to be killed according to the number of mutants created if the data set was linear (meaning all fault types are found at the same rate). When the results for a particular mutant type do not fall on the line (meaning it did not kill the number of mutants expected, according to the linear regression model), it indicates that particular mutant type *could* be an outlier. However, we cannot make any solid conclusions based on the visual appearance of the points plotted in Figure 3. Therefore, we calculate the *residuals* for each mutant type. Residuals are found by determining the difference between the observed value and the expected value (as shown in Equation 1).

$$\text{Residual} = \text{ObservedValue} - \text{PredictedValue} \quad (1)$$

One problem with residuals is the magnitude of the residual depends on the unit of measurement. This can make it difficult to determine unusual values. This problem can be eliminated by dividing the residual by an estimate of their

standard deviation, known as *studentized residuals*. Studentized residuals help identify outliers which are not consistent with the rest of the data. Data points that have a studentized residual with an absolute value of two or greater are considered statistically significant at the 95% α - level. A plot of the studentized residual values for each mutant operator is shown in Figure 4. Mutant operators with values of zero or near zero mean the results for this mutant operator are either the value that was expected according to the regression analysis (for values of zero) or close to expected (for values near zero). Results that are further from zero indicate that the rate of detection was different than what was expected.

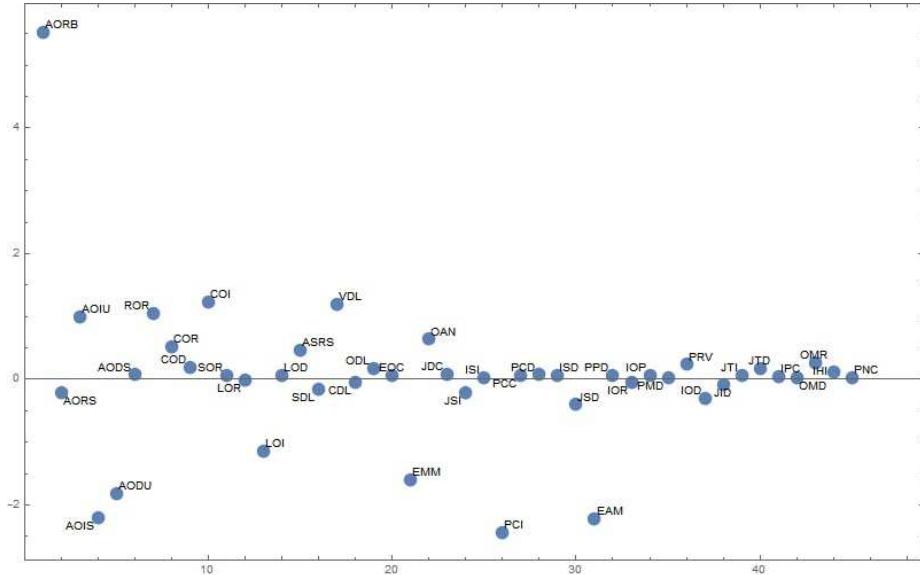


Figure 4: Studentized Residual with AORB

Table 6 provides the exact calculated studentized residual values for each mutant operator. The table includes five columns: the mutation operator, the total number of mutants created for that operator, the mutant's kill rate, the studentized residual using all mutants, and the studentized residual excluding AORB. The studentized residual for all mutants identified AORB as an *extreme* outlier, having a value of 5.53 (well above the absolute value of 2 which qualifies

Table 6: Mutant Operator Totals and Studentized Residuals

Mutant Operator	Total Created	Percent Killed	Studentized Residual (AORB Included)	Studentized Residual (AORB Excluded)
AORB	10586	80.14%	5.53	-
AORS	587	24.53%	-0.21	-0.23
AOIU	7402	63.48%	1	1.55
AOIS	21900	50.11%	-2.21	-2.14
AODU	2674	12.12%	-1.82	-2.31
AODS	92	54.35%	0.07	0.13
ROR	20443	58.34%	1.05	2.11
COR	1958	70.38%	0.53	0.77
COD	384	74.22%	0.19	0.28
COI	6268	67.21%	1.23	1.84
SOR	100	48.00%	0.06	0.11
LOR	437	42.79%	-0.01	0.02
LOI	8952	47.16%	-1.15	-1.22
LOD	9	33.33%	0.07	0.12
ASRS	1588	71.41%	0.47	0.68
SDL	20337	55.00%	-0.16	0.44
VDL	4283	72.19%	1.2	1.74
CDL	2022	51.58%	-0.06	0.01
ODL	12963	56.11%	0.17	0.61
EOC	14	28.57%	0.07	0.12
EMM	5710	37.58%	-1.59	-1.91
OAN	9694	59.32%	0.64	1.14
JDC	9	77.78%	0.08	0.13
JSI	983	37.54%	-0.21	-0.21
ISI	76	14.47%	0.02	0.06
PCI	4666	23.55%	-2.43	-3.12
PCC	35	34.29%	0.06	0.11
PCD	31	51.61%	0.07	0.12
ISD	43	39.53%	0.06	0.11
JSD	664	10.09%	-0.4	-0.48
EAM	8017	38.46%	-2.22	-2.7
PPD	6	00.00%	0.07	0.12
IOR	166	06.63%	-0.05	-0.04
IOP	19	26.32%	0.07	0.11
PMD	61	00.00%	0.02	0.06
PRV	1271	63.81%	0.24	0.37
IOD	969	30.96%	-0.3	-0.34
JID	247	17.41%	-0.07	-0.06
JTI	564	54.43%	0.06	0.13
JTD	284	76.41%	0.17	0.25
IPC	175	45.71%	0.05	0.09
OMD	78	21.79%	0.03	0.07
OMR	6979	57.42%	0.27	0.56
IHI	97	83.51%	0.12	0.18
PNC	76	13.16%	0.02	0.06

a value as an outlier with 95% confidence). A value with such an extreme residual value can pull the expected values line towards that observed value in such a way that other outliers could be missed. For this reason, we calculated the studentized residual values for the data excluding AORB as well. The studentized residual values excluding AORB are provided in the last column in Table 6 and a plot is shown in Figure 5.

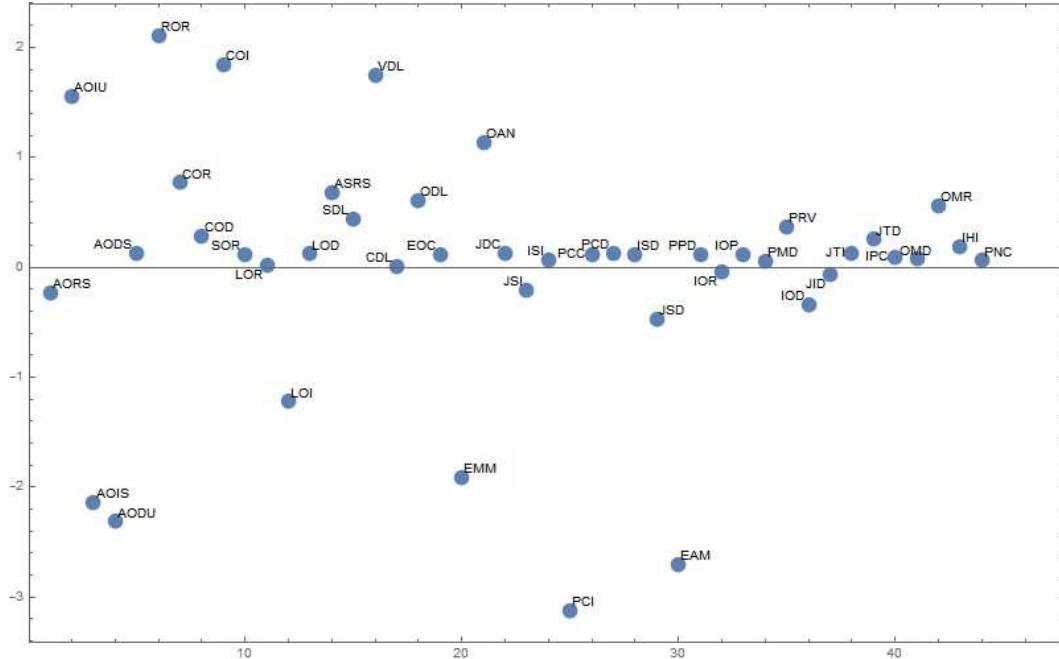


Figure 5: Studentized Residual without AORB

Examining the data with all mutant operators included reveals four outliers: AORB (with a value of 5.53), AOIS (with a value of -2.21), PCI (with a value of -2.43), and EAM (with a value of -2.22). A positive value indicates that the mutant operator was found *more often* than expected, while a negative value indicates the mutant type was killed *less often* than expected. Examining the data excluding the extreme outlier, AORB, reveals five outliers: AOIS (with a value of -2.14), AODU (with a value of -2.31), ROR (with a value of 2.11), PCI (with a value of -3.12), and EAM (with a value of -2.7).

5. Discussion

The results in the previous section identified six mutant types as outliers. Two of the mutant types are identified as outliers by having a kill rate *higher* than expected and four mutants types had a kill rate *lower* than expected. Since these are the unusual values that could be contributing the variance in the effectiveness of test suites, we direct our attention to these mutant types. Specifically, in this section, we provide a description of each mutant type of interest, a sample mutant that could be created using that mutant operator, and a discussion of the results for that mutant type.

5.1. Arithmetic Operator Replacement (Binary) (AORB)

The mutant with the most extreme results was AORB. It had a studentized residual value of 5.53. There were 10,586 mutants created for AORB with an overall kill rate of 80.14%. AORB is the *Arithmetic Operator Replacement (Binary)* mutant. It will replace basic binary arithmetic operators with other binary arithmetic operators. An example mutant, where the subtraction operator is replaced with the modulus operator, is provided below.

Original:	AORB Mutant:
<code>return i - 1;</code>	<code>return i % 1;</code>

This is a type of mutant operator where a high kill rate would be expected. Performing a completely different arithmetic operation will typically produce very different results, and should be caught by an effective test suite. Another factor that could have contributed to the results being an extreme outlier is the way MuJava seeds this particular type of fault. When MuJava sees the opportunity to insert this type of fault in the source code, it will create a mutation for every arithmetic operation. For instance, in the example above, it would create a mutant for `i % 1` as shown, but also for `i + 1`, `i * 1`, and `i / 1`. This leads to four mutants created for each opportunity found. Theoretically, if the test code is able to kill one of these mutants, it should kill the other three as well.

5.2. Relational Operator Replacement (ROR)

ROR is the *Relational Operator Replacement* mutant. It will replace a relational operator with another relational operator. An example is provided below.

Original:	OAN Mutant:
<code>return x > 0;</code>	<code>return x < 0;</code>

This mutation operator makes a drastic change to the program, often producing results exactly opposite of the intended action. It was not surprising that it was found at a higher rate than other mutation operators. In fact, it may be more surprising that it was *just* over the threshhold (at 2.11) to be considered an outlier.

5.3. Type Cast Operator Insertion (PCI)

PCI is the *type cast operator insertion* mutant. The PCI operator changes the actual type of an object reference to the parent or child of the original declared type. An example is provided below.

Original:	PCI Mutant:
<code>Child cRef = new Child();</code>	<code>Child cRef = new Child();</code>
<code>Parent pRef = cRef;</code>	<code>Parent pRef = cRef;</code>
<code>pRef.toString();</code>	<code>((Child)pRef).toString();</code>

There were 4,666 PCI mutants created with an overall kill rate of 23.6%. This particular mutant could be hard to find because a child and parent class have many similarities. Many of the methods behave exactly the same way, otherwise there would be no point in creating a child class. The test suite would have to determine which methods in the child class behave differently than the version from the parent class, and then focus on those methods. This mutation operator

likely results in a larger number of *equivalent mutants* than some of the other mutation operators, as it carries a high likelihood of producing a mutation with the same result, regardless of input used. Identifying equivalency is not generally a decidable problem, so we do not have the means currently of determining how often this particular mutation produces equivalent results. In practice, however, faults of this nature are possible and should be guarded against. Test oracles should be able to distinguish between versions of the system where the right method is called, and versions where the incorrect (parent) method is called.

5.4. Accessor Method Change (EAM)

This was one of the more surprising results. EAM is the *Accessor Method Change* mutant. The EAM mutant changes a call to an accessor method with the call to a different compatible accessor method. An example mutant is provided below.

Original:	EAM Mutant:
<code>point.getX();</code>	<code>point.getY();</code>

It is surprising this mutant operator was found less often than expected because retrieving the wrong value would be something that could certainly throw calculations and results off. This is an important finding because this could be an easy mistake for a programmer to make with most modern IDE's having some form of autocomplete or IntelliSense. It would be easy to accidentally choose the wrong method if all options are similarly-named. Test oracles could be made to find more of these mutants by validating the values of class attributes more frequently and more thoroughly.

5.5. Arithmetic Operator Insertion (Shortcut) (AOIS)

AOIS is another unexpected and interesting result. AOIS is the *Arithmetic Operator Insertion* mutant operator. It will increase or decrease a variable using the increment or decrement operator. An example is shown below.

Original:	AOIS Mutant:
return i;	return ++i;

Incrementing or decrementing a variable at places where it is not warranted could produce incorrect results or have other unpredictable outcomes. For example, if a mutant was created to increment a variable in the loop header, it could cause the loop to not execute the correct number of times. A mutant of this type going undetected could cause many unwanted behaviors. Better testing of boundary values could improve detection of these kind of “off-by-one” errors. This type of fault may also corrupt the internal state of the class, but may be hard to detect through inspection of method output alone. Oracles that more thoroughly inspect internal state may also help here.

5.6. Arithmetic Operator Deletion (Unary) (AODU)

Similar to the last two, AODU is also a surprising result. AODU is the *Arithmetic Operator Deletion*. It deletes basic unary arithmetic operators (+, -, ++, -, !). An example is shown below.

Original:	AODU Mutant:
return -1;	return 1;

Like AOIS, changing the value of a variable by changing the unary arithmetic operator should cause incorrect results and unpredictable behavior. The fact that test cases miss these type of errors suggests that the test oracles being used to check results are not specific enough—that they are allowing slight variations in the output, or ignoring whether a value is positive or negative.

5.7. General Trends

The last three mutants discussed presented three very surprising, and interesting, results. Each of these mutation types change the internal state of the

program by using the wrong value in some way—one by retrieving the wrong variable, one by incrementing or decrementing a value, and one by deleting a unary arithmetic operator. Each of these types of mutants could be caught more often by adding more validation to the values of variables in the program at different times throughout the test methods. For instance, better testing of boundary values could assist in situations where the result may be slightly off of the correct value. Regardless of the level of code coverage, the test must have an oracle sufficiently powerful to detect an exposed fault. Merely executing a line of code does not ensure that a fault is triggered. At the same time, we must design oracles that are thorough enough to detect a fault when it is triggered. The role of the test oracle is often downplayed in testing research [46]. More research is needed on this side of the equation—in deciding which variables to monitor and evaluate using the oracle, and in the types of assertions to use to maximize the likelihood of fault detection. The level of coverage and the thoroughness of the oracle have a dual influence on whether faults are detected, and we must make improvements in both regards.

5.8. Impact of the Study

The results of our study advance the state-of-the art by revealing the nature of faults that are getting missed most frequently (with statistical significance) by human-written test suites achieving high code coverage. Specifically, as described in the last section, three of the fault types missed at a statistically higher rate have a common problem: the internal state of the program is corrupted, and the results either do not change the output—or the oracle is not specific enough to detect the change. Developing test suites and test oracles more capable of finding these faults will improve the overall effectiveness of test suites.

Based on our results, we suggest that in order to strengthen test suites, test creation methods should consider whether test oracles are validating internal state along with meeting high code coverage. In order to find faults where the program has an incorrect internal state, tests need to not only execute the buggy code, but also have a test oracle sufficient to catch the buggy state.

Our recommendation is supported by the nature of the faults missed in our study as well as recent research investigating the relationship between assertions and test suite effectiveness [47, 13, 46]. Assertions are a type of a test oracle. Zhang and Mesbah conducted an empirical study and found that the number of assertions in a test suite strongly correlates with its effectiveness [13]. Chen et.al investigated how assertions impacted coverage-based test suite reduction techniques [47]. They found assertions are significantly correlated with the effectiveness of test suites used by coverage-based test suite reduction. In our past work, we found that the selection of program variables to monitor and check with the oracle has a major impact on fault detection [46]. We found that inspection of internal state has a major impact on the likelihood of fault detection, and that developers should monitor and inspect key bottleneck points in program execution.

These studies show that the choice and formulation of test oracles is important and makes a difference in test suite effectiveness. Our study advances this knowledge by providing empirical evidence showing *why* test oracles are important. We have identified specific fault types that create internal state problems, and provide evidence that test suites evaluated with coverage metrics alone did not detect them at the expected rate. Researchers and developers can improve their test suites by adding or improving test oracles in situations where bugs, similar to the ones shown in our study, are possible. Furthermore, researchers can use the evidence found in our study as motivation for developing test evaluation methods that consider code coverage and test oracles together, and their dual influence on test suite effectiveness.

6. Conclusions and Future Work

Code coverage is widely used as a method for evaluating the effectiveness of test suites. However, research has shown achieving high code coverage is not always a good indicator of fault-finding capability. Many past studies investigating the correlation between code coverage and fault detection have had

inconsistent findings, only sometimes showing a correlation between the two. In this work, we investigated one possible source of the inconsistencies observed: fault type. Specifically, we investigated how effective test suites achieving high code coverage were at detecting 45 different types of faults. Our results show that the effectiveness of finding the different faults varied greatly, and certain types of faults were missed at a much higher rate than others. Specifically, three of the four fault types identified as outliers were fault types that could corrupt internal state by using an incorrect value in some way. Based on this result, we suggest that test oracles should more thoroughly inspect internal state and boundary values, that developers carefully consider which variables are inspected using the test oracle, and that the dual influence of code coverage and test oracle be considered when evaluating test suites.

There is still much work to be done on this particular problem. Identifying the types of faults that are missed most frequently is only the first step. The information gained from this study can be used to propose improvements to test suites that would make them more capable of finding these faults. Also, although our study identifies one factor that is affecting the ability of test suites to find faults, there could be other factors as well—particularly centered around the role and formulation of test oracles. Additionally, other methods of evaluating test suites could be proposed and studied to identify whether they are a better indicator of a test suites’ ability to find faults.

7. Acknowledgements

This work has been partially supported by the Office of Sponsored Awards and Research Support (SARS) from the University of South Carolina Upstate.

References

- [1] M. Pezze, M. Young, Software Test and Analysis: Process, Principles, and Techniques, John Wiley and Sons, 2006.

- [2] A. Groce, M. A. Alipour, R. Gopinath, Coverage and its discontents, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward!'14, ACM, New York, NY, USA, 2014, pp. 255–268. doi:10.1145/2661136.2661157.
- URL <http://doi.acm.org/10.1145/2661136.2661157>
- [3] RTCA/DO-178C, Software considerations in airborne systems and equipment certification.
- [4] M. Heimdahl, M. Whalen, A. Rajan, M. Staats, On MC/DC and implementation structure: An empirical study, in: Digital Avionics Systems Conference (DASC), 2008, pp. 5–B.
- [5] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, F. Tip, A framework for automated testing of javascript web applications, in: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, 2011, pp. 571–580.
- [6] Q. Yang, J. J. Li, D. M. Weiss, A survey of coverage-based testing tools, *The Computer Journal* 52 (5) (2007) 589–597.
- [7] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 72–82.
- [8] A. S. Namin, J. H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the eighteenth International Symposium on Software Testing and Analysis, 2009, pp. 57–68.
- [9] G. Gay, The fitness function for the job: Search-based generation of test suites that detect real faults, in: Proceedings of the International Conference on Software Testing, ICST 2017, IEEE, 2017.
- [10] G. Gay, M. Staats, M. Whalen, M. Heimdahl, The risks of coverage-directed test case generation, *Software Engineering, IEEE Transactions on* PP (99). doi:10.1109/TSE.2015.2421011.

- [11] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 435–445.
- [12] G. Gay, A. Rajan, M. Staats, M. Whalen, M. P. E. Heimdahl, The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage, ACM Trans. Softw. Eng. Methodol. 25 (3) (2016) 25:1–25:34.
`doi:10.1145/2934672`
URL <http://doi.acm.org/10.1145/2934672>
- [13] Y. Zhang, A. Mesbah, Assertions are strongly correlated with test suite effectiveness, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 214–224.
- [14] A. Schwartz, M. Hetzel, The impact of fault type on the relationship between code coverage and fault detection, in: Proceedings of the 11th International Workshop on Automation of Software Test, ACM, 2016, pp. 29–35.
- [15] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al., An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.
- [16] C. S. Jensen, M. R. Prasad, A. Møller, Automated testing with targeted event sequence generation, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, 2013, pp. 67–77.
- [17] B. N. Nguyen, B. Robbins, I. Banerjee, A. Memon, Guitar: an innovative tool for automated testing of gui-driven software, Automated Software Engineering 21 (1) (2014) 65–105.
- [18] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT symposium

and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 416–419.

- [19] I. Ghosh, N. Shafiei, G. Li, W.-F. Chiang, JST: An automatic test generation tool for industrial java applications with strings, in: Proceedings of the International Conference on Software Engineering, 2013, pp. 992–1001.
- [20] P. D. Marinescu, C. Cadar, Katch: high-coverage testing of software patches, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 235–245.
- [21] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, A. M. Memon, Using gui ripping for automated testing of android applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 258–261.
- [22] F. Gross, G. Fraser, A. Zeller, Search-based system testing: high coverage, no false alarms, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM, 2012, pp. 67–77.
- [23] K. Inkumsah, T. Xie, Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution, in: 23rd International Conference on Automated Software Engineering, 2008, pp. 297–306.
- [24] P. G. Frankl, S. N. Weiss, An experimental comparison of the effectiveness of branch testing and data flow testing, Software Engineering, IEEE Transactions on 19 (8) (1993) 774–787.
- [25] X. Cai, M. R. Lyu, The effect of code coverage on fault detection under different testing profiles, ACM SIGSOFT Software Engineering Notes 30 (4) (2005) 1–7.
- [26] F. Del Frate, P. Garg, A. P. Mathur, A. Pasquini, On the correlation between code coverage and software reliability, in: Sixth International Symposium on Software Reliability Engineering, 1995, pp. 124–132.

- [27] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, D. Marinov, Comparing non-adequate test suites using coverage criteria, in: Proceedings of the International Symposium on Software Testing and Analysis, 2013, pp. 302–313.
- [28] P. S. Kochhar, F. Thung, D. Lo, Code coverage and test suite effectiveness: Empirical study with real bugs in large systems, in: 22nd International Conference on Software Analysis, Evolution, and Reengineering, 2015, pp. 560–564.
- [29] M. Staats, G. Gay, M. Whalen, M. Heimdahl, On the danger of coverage directed test case generation, in: Fundamental Approaches to Software Engineering, 2012, pp. 409–424.
- [30] A. Perez, R. Abreu, A. van Deursen, A test-suite diagnosability metric for spectrum-based fault localization approaches, in: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, 2017, pp. 654–664.
- [31] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? a controlled empirical study, *ACM Trans. Softw. Eng. Methodol.* 24 (4) (2015) 23:1–23:49. doi: 10.1145/2699688.
URL <http://doi.acm.org/10.1145/2699688>
- [32] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated white-box test generation really help software testers?, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA, ACM, New York, NY, USA, 2013, pp. 291–301. doi:10.1145/2483760.2483774.
URL <http://doi.acm.org/10.1145/2483760.2483774>
- [33] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, J. Benefelds, An industrial evaluation of unit test generation: Finding real faults in a financial

- application, in: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP), ICSE 2017, ACM, New York, NY, USA, 2017.
- [34] P. G. Frankl, O. Iakounenko, Further empirical studies of test effectiveness, ACM SIGSOFT Software Engineering Notes 23 (6) (1998) 153–162.
 - [35] P. G. Frankl, S. N. Weiss, C. Hu, All-uses vs mutation testing: an experimental comparison of effectiveness, Journal of Systems and Software 38 (3) (1997) 235–253.
 - [36] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria, in: Proceedings of the 16th International Conference on Software Engineering, 1994, pp. 191–200.
 - [37] Y.-S. Ma, Y.-R. Kwon, J. Offutt, Inter-class mutation operators for java, in: Proceedings of the 13th International Symposium on Software Reliability Engineering, 2002, pp. 352–363.
 - [38] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of mujava, in: Proceedings of the 2006 International Workshop on Automation of Software Test, 2006, pp. 78–84.
 - [39] V. Debroy, W. E. Wong, Insights on fault interference for programs with multiple bugs, in: Software Reliability Engineering, 2009. ISSRE’09. 20th International Symposium on, IEEE, 2009, pp. 165–174.
 - [40] N. DiGiuseppe, J. A. Jones, Fault interaction and its repercussions, in: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, IEEE, 2011, pp. 3–12.
 - [41] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings. 27th International Conference on Software Engineering, 2005, pp. 402–411.

- [42] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing, in: 22nd International Symposium on the Foundations of Software Engineering, 2014, pp. 654–665.
- [43] R. Just, The major mutation framework: Efficient and scalable mutation analysis for java, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 433–436.
- [44] Y.-S. Ma, J. Offutt, Y.-R. Kwon, MuJava: a mutation system for java, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 827–830.
- [45] J. Offutt, Y.-S. Ma, Description of mujava’s method-level mutation operators, in: Electronics and Telecommunications Research Institute, Korea, Tech. Rep, 2005.
- [46] G. Gay, M. Staats, M. Whalen, M. Heimdahl, Automated oracle data selection support, *Software Engineering, IEEE Transactions on* PP (99) (2015) 1–1. doi:10.1109/TSE.2015.2436920.
- [47] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, How do assertions impact coverage-based test-suite reduction?, in: Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on, IEEE, 2017, pp. 418–423.

Ensuring the Observability of Structural Test Obligations

Ying Meng, Gregory Gay, *Member, IEEE*, Michael Whalen, *Senior Member, IEEE*

Abstract—Test adequacy criteria are widely used to guide test creation. However, many of these criteria are sensitive to statement structure or the choice of test oracle. This is because such criteria ensure that execution *reaches* the element of interest, but impose no constraints on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered. To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that combines the obligations of a host criterion with an additional path condition that increases the likelihood that a fault encountered will propagate to a monitored variable.

Our study, conducted over five industrial systems and an additional forty open-source systems, has revealed that adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of 125.98% in mutation detection with the common output-only test oracle and per-model improvements of up to 1760.52%. Ultimately, there is merit to our hypothesis—observability reduces sensitivity to the choice of oracle and to the program structure.

Index Terms—Software Testing, Automated Test Generation, Test Adequacy Criteria, Model-Based Test Generation

1 INTRODUCTION

Test adequacy criteria defined over program structures—such as statement, branches, or atomic conditions—are widely used as measures to assess the efficacy of test suites. Such criteria are essential in offering guidance in the testing process, as they offer clear checklists of goals—called *test obligations*—to testers and the means to automate the creation of test suites. However, many of these criteria are highly sensitive to how statements are structured [1], [2] or the choice of test oracle—the artifact used to judge program correctness [3]–[5].

Consider the Modified Condition/Decision Coverage (MC/DC) coverage criterion [6]. MC/DC is used as an exit criterion when testing software in the avionics domain. For certification, a vendor must demonstrate that the test suite provides MC/DC coverage of the source code [7]. However, the efficacy of test suites created to satisfy MC/DC—particularly when test suite creation is automated—is highly dependent on the syntactic structure of the code under test. A complex Boolean expression, for example, could be written as a series of simple expressions, or as a single *inlined* expression. This simple transformation can dramatically improve the efficacy of MC/DC-satisfying test suites, increasing fault detection efficacy by orders of magnitude [1].

Such results are worrying, particularly given the importance of coverage criteria in safety certification, and

the improvements made in terms of automated generation. When examining the discrepancy in efficacy between test suites for non-inlined and inlined programs, we often found that the test case encountered a fault in the code—such as an erroneous Boolean operator—leading to a corrupted internal state. However, this corruption was *masked out* in a subsequent expression, and did not propagate to an output. This effect was prevalent in programs containing large numbers of simple Boolean expressions stored in local variables. Even in cases where a non-masking path could theoretically be found, it was common for a test case to not allow sufficient execution time for corrupted state to propagate.

This sensitivity to structure and choice of oracle is caused by the fact that the obligations of structural coverage criteria are only posed over specific syntactic elements—statements, branches, conditions. Such obligations ensure that execution *reaches* the element of interest, and exercises it in the prescribed manner. However, no constraints are imposed on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered.

To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that has the potential to eliminate masking. Observable coverage criteria combine the test obligations of their host criterion with an additional path condition that increases the likelihood that a fault encountered when executing the element of interest will propagate to a variable monitored and checked for correctness by the test oracle. Unlike many extensions to coverage criteria [8], this path condition does not increase the number of test obligations over its host criterion. Instead, it makes the existing obligations more stringent to satisfy, as the possibility of propagating a

Y. Meng and G. Gay are with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: ymeng@email.sc.edu, greg@greggay.com

M. Whalen is with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: mwwhalen@umn.edu

This work has been partially supported by NSF grant CCF-1657299. We also thank the Advanced Technology Center at Rockwell Collins Inc. for granting access to industrial case examples.

fault revealed by the original obligation must also be demonstrated. We hypothesize that observability will improve the effectiveness of the host criterion—no matter which criterion is chosen—particularly when used as a test generation target, paired with test oracles that only examine the values of output variables.

Focusing on safety-critical applications, we have implemented observable versions of Branch, Condition, Decision, and MC/DC Coverage as part of model-based test generation for the Lustre dataflow language [9]. While our implementation is for dataflow languages, the *concept* of observability is not restricted to any one language, generation paradigm, or product domain, and our semantic model should be valid for imperative languages such as C or Java.

This work is an extension of our prior work defining and exploring the concept of observability [10]–[12]. We first proposed the concept of observability as an extension of the MC/DC coverage criterion [10]. An extended study found that OMC/DC was more effective—and overcame many of the weaknesses of—traditional coverage criteria for a small set of studied systems [11]. We extend previous efforts by decoupling the notion of observability from MC/DC and exploring its application as a generic addition to any coverage criterion. This decoupling allows us to explore the impact of the choice of host criterion, and to explore the efficacy of observability as a general extension to adequacy criteria. Our new experimental studies also consider a wider range of programs than previously explored in order to better understand the efficacy of observability-based coverage criteria as the target of automated generation.

Our study, conducted over five industrial systems from Rockwell Collins and an additional forty open-source systems, has revealed the following insights:

- Test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (maximum oracle strategy)/87.03% (output-only oracle strategy) for the inlined Rockwell models, 98.85% (maximum)/85.88% (output-only) for the non-inlined Rockwell models, and 89.62% (maximum)/65.14% (output-only) for the Benchmarks models.
- Adding observability tends to improve efficacy over satisfaction of traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%.
- Factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output.
- The addition of observability requires a longer test generation process, with average increases ranging from 129.39%–2422.91%. However, this increase tends to be relatively small—seconds to minutes.

- The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the paths from each expression to the output.
- The addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to solve.
- The choice of host criterion influences efficacy, but the largest increase in complexity comes from the addition of observability. Varying both criterion and observability may allow testers to find an optimal level of efficacy and complexity.
- Observability reduces sensitivity to the choice of oracle, by ensuring a masking-free path from expression to the variables monitored by the test oracle.
- Observability reduces sensitivity to the program structure by capturing the complexity benefits of inlining in the path from expression to output.

Based on our results, *observability* is a valuable extension—regardless of the chosen host criterion—to coverage criteria for dataflow languages. Requiring observability increases test efficacy and produces suites that are robust to changes in the structure of program or the variables being monitored by the test oracle.

The remainder of this article is structured as follows. Section 2 introduces important background material. Section 3 presents the concept of observability and offers formal definitions and implementation details. Section 4 presents the details of our experiments, and Section 5 discusses our observations. Section 6 discusses threats to validity. Section 7 presents related work. Finally, Section 8 summarizes and concludes the manuscript.

2 BACKGROUND

In this research, we are interested in improvements to the criteria used to judge the adequacy of testing efforts—and to automatically generate test suites. In particular, we are focused on the safety-critical reactive systems that power our society. In this section, we will discuss background material on both topics.

2.1 Adequacy Criteria

The concept of adequacy is important in providing developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of our testing efforts. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*?

The most common methods of measuring adequacy involve coverage of structural elements of code, such as branches of control, and Boolean expressions [13]–[15]. Each adequacy criterion prescribes a series of *test*

obligations (or requirements) that tests must fulfill. For example, branch coverage requires that all outcomes of expressions that can result in different code segments being executed—such as if-then-else and loop conditions—be executed. The idea of measuring adequacy through coverage is simple, but compelling: unless code is executed, many faults are unlikely to be found. If tests execute elements as prescribed by the criterion, then testing is deemed “adequate” with respect to faults that manifest through such structures.

Adequacy criteria have seen widespread use, as they offer objective, measurable checklists [16] and—importantly—*stopping criteria* for the testing process. For that same reason, they are ideal as test generation targets [17]–[19], as coverage can be straightforwardly measured and optimized for [20].

2.2 Structural Coverage

Structural coverage criteria serve as a means to determine that the structure of system under test—the various elements making up the source code—have been thoroughly exercised by test cases. Many structural coverage criteria, defined with respect to specific syntactic elements of a program, have been proposed and studied over the past decades [11], [21]. These have been used to measure suite adequacy—as a means to assess the quality of existing test suites, and whether developers can stop adding tests. They are also commonly used as targets for automated test generation.

In this study, we are primarily concerned with reactive systems—safety-critical embedded systems that interact with the physical world. Such systems often have sophisticated logical structures in the code. Therefore, in this work, we are primarily concerned with structural coverage criteria defined over Boolean expressions. In particular, we are focused on Condition Coverage, Branch Coverage, Decision Coverage, and Modified Condition/Decision Coverage (MC/DC).

Decision Coverage: A decision is a Boolean expression. Decisions are composed of one or more conditions—atomic Boolean subexpressions—connected by operators (and, or, xor, not). Decision Coverage requires that all decisions in the system under test evaluate to both the true and false. Given the expression $((a \text{ and } b) \text{ and } (\text{not } c \text{ or } d))$, tests would need to be produced where the expression evaluates to true and the expression evaluated to false. In this case, the test input (TTTT), (TTTF) would satisfy Decision Coverage.

Branch Coverage: A branch is a particular type of decision that can cause program execution to diverge down a particular control flow path, such as that in if or case statements. Branch Coverage is defined in the same manner as Decision Coverage, but is restricted to branches, rather than all decision statements. Branch Coverage is arguably the most commonly used coverage

criterion, with ample tool support¹ and industrial adoption. Improving Branch Coverage is a common goal in automated test generation [18], [22].

Condition Coverage: A condition is an atomic Boolean subexpression within the broader decision. Condition Coverage requires that each condition evaluate to true and false. Given the expression $((a \text{ and } b) \text{ and } (\text{not } c \text{ or } d))$, achieving Condition Coverage requires tests where the individual atomic Boolean conditions a , b , c , and d evaluate to true and false. For this decision, test input (TTTF), (FFFT) would satisfy the obligations of Condition Coverage.

Note that satisfying the obligations of one form of coverage does not always imply that the obligations of others are fulfilled as well. The test input given above would satisfy Condition Coverage, but not Decision Coverage, as both test inputs result in the decision evaluating to false. Similarly, the input provided earlier for Decision Coverage—(TTTT), (TTTF)—would not satisfy Condition Coverage, as only d evaluates to both outcomes. Therefore, stronger criteria—such as Modified Condition/Decision Coverage—require that the obligations of both Decision and Condition Coverage be met.

Modified Condition/Decision Coverage (MC/DC): The MC/DC criterion is used as an exit criterion when testing software for critical software in the avionics domain, and is required for safety certification in that domain [23]. MC/DC further strengthens Condition and Decision Coverage by requiring that each decision evaluate to all possible outcomes, each condition take on all possible outcomes, and that each condition be shown to independently impact the outcome of the decision.

Independent effect is defined in terms of *masking*—a masked condition has no effect on the value of the decision; for example, given a decision of the form x and y , the truth value of x is irrelevant if y is false, so we state that x is masked out. A condition that is not masked out has *independent effect* for the decision.

Consider again the expression $((a \text{ and } b) \text{ and } (\text{not } c \text{ or } d))$. Suppose we examine the independent affect of d in the example; if $(a \text{ and } b)$ evaluates to false, than the entire decision will evaluate to false, masking the effect of d ; Similarly, if c evaluates to false, then $(\text{not } c \text{ or } d)$ evaluates to true regardless of the value of d . Only if we assign a , b , and c the value of true does the value of d affect the outcome of the decision. Showing independent impact requires a pair of test cases where all other conditions hold fixed values and our condition of interest flips values. If changing the value of the condition of interest changes the value of the decision as a whole, then the independent impact has been shown. In this example, the test inputs (TTTT), (TTTF), (FTTT), (TFTT), and (TTFF) satisfies MC/DC. Tests inputs 1 and 3 show the effect of a , 1 and 4 show b , 2 and 5 show c , and 1 and 2 show d .

1. Such as the Cobertura and EMMA IDE plug-ins—see <http://cobertura.github.io/cobertura/> and <http://www.eclemma.org/>

MC/DC can be satisfied in $(\text{number of conditions} + 1)$ test cases if care is taken in selecting test input.

Because both decisions and conditions are covered, we state that MC/DC *subsumes* the previously-defined forms of coverage. Satisfying the obligations of MC/DC also satisfies the obligations of Decision and Condition Coverage. This comes at a cost—satisfying MC/DC requires more test cases and more effort than satisfying any of the above criteria. Therefore, if no benefit is perceived from the additional requirements of MC/DC, testers often elect to satisfy a simpler criterion instead.

Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [24].

2.3 Mutations and Mutation Coverage

Mutation [25] is a technique in which a user generates many faulty implementations through small modifications of the original implementation, either through automated code transformation or by hand [26], [27]. Usually a single modification is made to each *mutant* implementation, such as changing a single expression (substituting addition for subtraction, e.g.), permuting the order of two statements, or many other possible changes. The mutations introduced generally match one or more models of the types of mistakes that real developers make when building code. Generally, mutants are introduced with the intent that they not be trivially detected—they are both syntactically and semantically valid [15]. That is, the mutants will compile, and no mutant will crash the system.

Mutations can be used to assess the effectiveness of a test suite by examining how many mutants are *killed* (that is, detected) by the tests within the test suite. Detection of mutants has also been the basis of multiple adequacy criteria [28], [29]. In theory, if a suite detects more mutants, it will also be more adequate at fault detection. To kill a mutant using *strong mutation*, the following conditions must be met [30]:

- (R) the test must *reach* the mutation.
- (I) the test must *infect* program state by causing it to differ between the original and mutated program.
- (P) incorrect state must *propagate* to program output.
- (R) the test oracle must *reveal* the difference.

In strong mutation coverage, the resulting corruption must influence an output variable. *Weak mutation* only requires the (R,I) steps. In weak mutation coverage, a mutant is considered detected if the mutated statement is reached, and the value of *that expression* is corrupted [31]. *Firm mutation* requires propagation to some point of observation in the system, but not necessarily an output [32]. For each of the metrics, a *mutation coverage* score can be determined by dividing the number of killed mutants by the number of all mutants.

2.4 Reactive Systems and Dataflow Languages

Increasingly, our society is powered by sophisticated software systems—such systems manage factories and

power plants, coordinate the many systems driving automobiles and airplanes, and even make life-saving decisions as part of medical devices implanted in human bodies. Many of these systems are what we refer to as *reactive systems*—embedded systems that interact with physical processes. Reactive systems operate in cycles—receiving new input from their environment, to which they react by issuing output.

Such systems are commonly designed using modeling languages, which are translated into C code that can be directly flashed to hardware. Models can be developed using visual notations, such as Simulink [33], Stateflow [34] and SCADE [35]. They can also be directly expressed using *dataflow languages*, such as Lustre.

Lustre is a synchronous dataflow language used in a number of domains to model or directly implement embedded systems [9]. It is a declarative programming language for manipulating *data flows*—infinite streams of variable values. These variables correspond to traditional data types, such as integers, booleans, and floating point numbers. Lustre offers an intermediate representation between behavioral model and traditional source code that is useful for specification, design, and analysis purposes. Because of the simplicity and declarative nature of Lustre, it is well-suited to model checking and verification, in particular with regards to its safety properties [36]. Lustre programs can be automatically generated from visual notations such as Simulink, and can be automatically compiled to target languages such as C/C++, VHDL, as well as to input models for verification tools such as model checkers.

A Lustre program is structured into a network of control modules (nodes) that specify relations between inputs and outputs of a system. A node specifies a stream transformer, mapping streams of input variables to streams of internal and output variables using a set of defined expressions. Lustre nodes have cyclic behavior—at execution cycle i , the node takes in the values of the input streams at instant i , manipulates those values, and issues new values for the internal and output variables. Nodes have a limited form of memory, and can access input, internal, and output values from previous instants (up to a statically-determined finite limit). To update program state within one computational step, combinatorial variables are used; to store current program state for the reference by later cycle or cycles, delay variables are used (i.e., $\frac{1}{z}$ blocks in Simulink). During a cycle, all variables are calculated according to their definitions: combinatorial variables are computed combinatorially using values at the current computational step, and delay variables are computed combinatorially using values from previous step or steps.

The body of a Lustre node consists of a set of equations of the form $x = \text{expr}$ where x is a variable identifier, and expr is the expression defining the value of x at instant i . Like in most programming languages, expression t can make use of any of the other input, internal, or output variables in defining x —as long as that variable

```
v1 = (0 -> (pre in1));
v2 = (v1 > 1);
v3 = (false -> (pre v2));
out = (if in2 then v2 else v3);
```

Fig. 1: Sample Lustre code fragment

has already been assigned a value during the current cycle of computation. The order of equations does not matter in Lustre, except for data dependencies. That is, within a computational step, as long as all the variables involved in an equation have already been computed, the equation can be evaluated.

Lustre supports many of the traditional numerical and boolean operators, including $+$, $-$, $*$, $/$, $<$, $>$, $\%$, etc. Lustre also supports two important *temporal* operators: $pre(x)$ and \rightarrow . The $pre(x)$ operator, or “previous”, evaluates to the value of x at instant $(i-1)$. The \rightarrow operator, or “followed by”, allows initialization of variables in the first instant of execution. For example, the expression $x = 0 \rightarrow pre(x) + 1$ defines the value of x to be 0 in instant 0, then defines it as 1 at instant 1—or, the value at instant 0 plus one—and so forth.

For example, consider the code fragment in Figure 1, in which $in1$ and $in2$ are input variables, $v1$, $v2$, and $v3$ are internal variables, and out is an output variable. Variables $in1$ and $v1$ are type of int and all the rests are type of boolean. Variables $in1$ and $v2$ are delay variables, values stored in them will be used by $v1$ and $v3$ in the next cycle, respectively. Variable $v1$ is initially assigned to 0 followed by (represented by operator *arrow*) $in1$'s value from the previous cycle, at each subsequent cycle. Similarly, values of variable $v3$ is a stream of boolean values, which starts with a *false* followed by $v2$'s value from the previous computational step.

2.4.1 Test Case Structure for Reactive Systems

There are two key artifacts necessary to construct a test case, the *test inputs*, or *test data*—inputs given to the system under test—and the *test oracle*—a judge on the resulting execution [37], [38]. A *test oracle* can be defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [39].

As reactive systems compute in cycles, multiple test inputs must generally be provided. Therefore, tests are divided into a series of *test steps*, where input and expected output is provided for *each step*. In each step, specific values are given for each input variable, then the internal and output variables are computed accordingly. The output at each step is compared to the expected output provided as part of the test oracle. Table 1 shows example test input that contains four steps together with corresponding evaluations of all internal and output variables. From this example, we can see how the values of delay variables impact other variables.

TABLE 1: Sample Lustre Program Evaluation

step	inputs (in1, in2)	internals (v1, v2, v3)	outputs (out)
1	(1, T)	(0, F, F)	(F)
2	(2, T)	(1, F, F)	(F)
3	(3, F)	(2, T, F)	(F)
4	(4, F)	(3, T, T)	(T)

3 OBSERVABILITY-BASED TEST CREATION

In this chapter, we will illustrate the common issue impacting the efficacy of test suites generated to satisfy structural coverage criteria—*masking*—and formally define our solution—*observability* [10]. We then will describe how extending common structural coverage criteria to require observability can overcome masking, and consequently, sensitivity to program structure and oracle. Finally, we will describe how we implemented our tool to generate test obligations for observability-based coverage criteria.

3.1 Masking

Previous research has shown that the efficacy of test suites satisfying structural coverage criteria—defined over specific program elements such as control-flow branches, conditions, or decisions—can be highly sensitive to how expressions are written [1], [2], [11] and the selection of variables monitored by the test oracle [3]–[5]. This is due to *masking*, when the value of a variable or subexpression is prevented from influencing the outcome of another expression. In other words, masking prevents the *propagation* of this effect, in the sense of mutation, to a program output.

In this work, we are primarily concerned with masking in terms of Boolean expressions. Masking occurs when the value of a condition (an atomic variable or subexpression) in a Boolean decision hides the effects of other conditions. We state a condition is masked if the outcome of a Boolean decision cannot be changed by varying the value of the condition while holding the rest of the conditions fixed (i.e., no matter what value the condition is, the final outcome of syntactic element of interest does not change). For example, input $a = \text{true}$ masks b in the decision $(a \text{ or } b)$. As the decision's outcome is always *true*, regardless of the value of b , a is masked. Similarly, $a = \text{false}$ masks b in decision $(a \text{ and } b)$, as the decision will always evaluate to *false*.

By requiring that each condition demonstrate an independent influence on its decision's outcome, MC/DC is designed to prevent masking *within* an expression. Test cases must exist where, if we flip the value of a single condition while the others are held constant, the outcome of that decision must be changed. Branch, Decision, and Condition Coverage lack any such guarantee. This is one reason MC/DC is often required for testing of avionics and other safety critical systems—its requirements are more strenuous, but the additional assurances of the independent impact requirement theoretically increase the probability that logic faults will be detected.

```
1. v1 = in1 or in2;
2. out = v1 and in3;
```

Fig. 2: Non-inlined sample code

```
1. out = ((in1 or in2) and in3);
```

Fig. 3: Inlined sample code

However, how the code is structured has a major impact on the formulation of the test obligations—the prescribed goals—for a criterion and the efficacy of the suites satisfying such obligations. Consider the code fragments in Figures 2 and 3. The two code fragments are *semantically* identical—they offer the same outcome—but are written in two different styles. The fragment in Figure 2 is split over two separate, simple equations (a *non-inlined* style). The fragment in Figure 3 is *inlined*—written as a single, complex expression.

As the obligations for criteria such as MC/DC are posed over individual program elements, the MC/DC obligations for the non-inlined version will be much simpler—and more trivially satisfied—than obligations for the inlined version. In the non-inlined version, for example, `in1` must be shown to overcome any masking from the value of `in2`. However, in the inlined version, `in1` must overcome masking from both `in2` and `in3`. As a result, MC/DC is much harder to satisfy over inlined implementations, and requires a larger number of test cases. The produced test suites tend to be far more effective [1], [2]. Therefore, we can see that traditional coverage criteria are sensitive to program structure.

Further, just because a condition is shown to influence the outcome of the decision it resides within, there is no assurance that the condition will influence the *program output*. Consider again the sample code fragment in Figure 2. Based on the definition of MC/DC, `TestSuite1` in Table 2 provides MC/DC over the program fragment in Figure 2; the test cases with `in3 = false` (**bold faced**) contribute towards MC/DC of `in1` or `in2` in `v1`. Nevertheless, if we monitor the output variable `out`, the effect of `in1` and `in2` cannot be observed in the output since it will be masked out by `in3 = false`. Thus, `TestSuite1` gives us MC/DC coverage of the non-inlined program fragment, but a fault on the first line will never propagate to the output. On the other hand, `TestSuite2` will also give MC/DC coverage of the program, but since `in3 = true` in the first two test cases, faults in the first statement can propagate to an output.

Because coverage obligations are posed over individual program elements, and make no demands on what happens after that element is executed, masking can prevent triggered faults from being observed. Masking can prevent an infection from propagating, hindering the oracle’s ability to reveal a fault.

Masking can be partially mitigated through selection of the correct oracle strategy. For instance, by monitoring all internal state variables as well as all the outputs,

TABLE 2: Sample test suites satisfying MC/DC for the code in Figures 2-3

$$\begin{aligned} \text{TestSuite1} &= \{(T, F, F), (\mathbf{F}, T, F), (F, F, T), (T, T, T)\} \\ \text{TestSuite2} &= \{(T, F, T), (F, T, T), (F, F, T), (T, F, F)\} \end{aligned}$$

masking between statements is not an issue [3], [5], [40]. In the case of Figure 2, if we monitor the value of `v1` during testing, failures introduced by `in1` or `in2` can be detected without changing test suites. However, monitoring and specifying expected values for all variables is generally prohibitively expensive (or outright infeasible). A subset of variables could be used, if carefully chosen, but this selection is also non-trivial to make.

An alternative approach is to strengthen the coverage criteria with conditions on execution along the path from the program element of interest to the output (or other chosen oracle variables). Such path conditions can ensure the *observability* of such elements when we test.

3.2 Observability

The observability of a program is the degree to which it is possible to infer the internal state of a system given the information that we can monitor from the program—generally through program output [5]. We say an expression in a program is *observable* in a test case if we can change only the expression’s value—keeping the rest of the program fixed—and see the influence of this change in the result of the test case. Otherwise, if this update has no visible influence, we say the expression is *not observable* in that test case. As we will see, observability is closely related to strong mutation.

For example, consider the program fragment in Figure 2. We could replace the expression defining variable `v1` with a fixed value of `false`. Normally, execution of the first test case in `TestSuite1` in Table 2 would cause `v1` to evaluate to `true`. However, as the effect of executing `v1` is masked by `in3` in expression `out`, we would not notice the substitution—we lack *observability* when executing this test case. On the other hand, if we executed the first test case from `TestSuite2` instead, we would detect the substitution. In that case, we can establish observability.

In theory, masking can be overcome by requiring observability from a test suite—in addition to the existing test obligations of a host coverage criterion. Informally, we can obtain observability of test obligations by requiring that the variable whose assignment contains a particular element of interest remains unmasked through a path to a variable monitored by the test oracle.

Although this notion of observability was previously defined as an explicit extension to MC/DC [10], such requirements can be imposed on any existing criterion over Boolean expressions. The path conditions of observability establish a masking-clear path from an expression containing a program element of interest—one with obligations defined over it—to a monitored variable. In this study, we apply observability to Branch, Condition, Decision, and MC/DC Coverage.

To formally define how observability is established, we can view a deterministic program P containing expression e as a transformer from inputs I to outputs $O : P : I \rightarrow O$. We write $P[v/e_n]$ for program P where the computed value for the n^{th} instance of expression e is replaced by value v . Note that this is not a substitution. Rather, we replace a single instance of expression e rather than all instances, which is more akin to mutation. We state e is observable in test t if $\exists v. P(t) \neq P[v/e_n](t)$.

This formulation is a generalization of the semantic idea behind masking MC/DC [24], lifted from decisions to programs. In masking MC/DC, the main obligation is that, for each condition c in given decision D , there are a pair of test cases t_i and t_j ensuring that c is observable in D 's outcome for both outcomes (true and false): $((D(t_i) \neq D[\text{true}/c](t_i)) \wedge ((D(t_j) \neq D[\text{false}/c](t_j)))$.

One can directly lift MC/DC obligations to observable MC/DC obligations by moving the observability obligation from the decision to the program output. Given test suite T , the OMC/DC obligations are:

$$\begin{aligned} & (\forall c \in \text{Cond}(P)). \\ & ((\exists t \in T. (P(t) \neq P[\text{true}/c](t))) \wedge (\exists t \in T. (P(t) \neq P[\text{false}/c](t)))) \end{aligned} \quad (1)$$

where $\text{Cond}(P)$ is the set of all conditions in program P . This formula can be straightforwardly generalized from conditions to several different program structures: decisions to perform Observable Decision Coverage, branches for Observable Branch Coverage, or Boolean variable assignments if we wish to pair Observability with Condition Coverage.

There are strong connections between MC/DC, observability, and mutation testing. From the definition above, it is clear that masking MC/DC corresponds to weak or firm mutation (depending on whether or not a decision is observable) for mutations that replace each condition with constant *true* and *false*. Similarly, Observable MC/DC corresponds to strong mutation for these mutants. Observability offers the means to ensure a path from an expression to the program output, ensuring that the effect of a fault is detected. By explicitly defining a path constraint, Observable criteria offer feedback to the test generation process. In fact, one could use the same path constraints along with Boolean-based mutation operators to satisfy a subset of strong mutation.

This relationship shows a connection between equivalent mutants and “dead code”. If an MC/DC (resp. Observable MC/DC) obligation cannot be satisfied, it means that the expression can be simplified in such a way that the program behaves equivalently, as has been examined in the literature on vacuity [41].

3.3 Tagged Semantics

The semantic definition for observability, defined above, is unwieldy for test generation and test measurement. The analysis would require two versions of the program running in parallel to check that the results match. Then,

for test measurement, the test suite must be executed separately for *each pair* of modified programs.

In order to define an observability constraint that efficiently supports monitoring and test generation, we can approximate semantic observability using a tagged semantics approach [42]. Each variable corresponding to a Boolean expression or atomic value in the program is assigned a tag, the observability of which is tracked through the execution of the a program. If a tag is propagated to the output—or any “monitored” internal variable—the corresponding path condition is considered to be fulfilled. More precisely, we track pairings of tag and concrete outcome. If a tagged variable appears more than once in a decision, a tag is assigned to each occurrence uniquely. We then examine the number of all possible pairs that have reached as output in some test in order to evaluate the coverage level for a test suite.

Formal tagging semantics have been defined for a set of expressions, an imperative command language, as well as a simple dataflow language (shown in Table 3). A reduction semantics with evaluation contexts (RSEC) [43] is used for presentation, and the K tool suite [44] is used to check for consistency. The rules, which run over *configurations* containing K (the syntax being evaluated) and a set of configuration parameters being labeled, operate by applying rewrites at positions in syntax where the evaluation context allow. A *context* can be a program or program fragment with a *hole* (represented by \square)—a placeholder where a rewrite can occur. In their definition, maps are assumed to have operations—lookup (σx) and update $\sigma[x \leftarrow \nu]$, the empty map \emptyset , and lists with concatenation $x.y$ and cons $\text{elem} :: x$, and operators. Additional syntax, which will be formatted as gray background to distinguish from user-level syntax, may be introduced during rewriting.

In Table 3, expressions yield (Val, TS) pairs, where TS is a set of tags, and are evaluated in a context containing environment ε of type $Env = (id \rightarrow (Val \times TS))$. The expressions are standard, except the $\text{tag}(E, T)$ which adds a tag to the set of tags associated with the expression E . For any structural coverage, it is assumed that each Boolean variable is wrapped in a *tag* expression. Masking is defined by operators: 1) *and*—given $(a \text{ and } b)$, for a is not masked out, b has to be *true*, so the tag assigned to a propagates only when b is *true* (and vice-versa); 2) *or*—given $(a \text{ or } b)$, for a is not masked out, b has to be *false*, so the tag assigned to a propagates only if b is *false* (and vice-versa); 3) *ite*—given $(\text{if } a \text{ then } b \text{ else } c)$, for b is not masked out, a must be *true*, therefore b 's tag propagates when a is *true*; similarly, c 's tag propagates when a is *false*; 4) relation expressions such as $a > b$, a and b are never masked out by each other; these will not be shown in Table 3.

The imperative language semantics define the way tags broadcast through commands: tags need to propagate through all variables assigned in either branch in conditional statements, for the value of a variable can be influenced by not being assigned by the condition.

TABLE 3: Syntax and tagging semantics for imperative and dataflow programs

Expression syntax, context, and semantics:

$E ::=$	$Val \mid Id \mid E op E \mid not E \mid$	$E ? E : E \mid tag(E, T) \mid (Val, TS) \mid addTags(E, TS)$
$Context ::=$	$\square \mid Context op E \mid E op Context \mid not Context \mid$	
	$Context ? E : E \mid addTags(Context, TS) \mid$	
	$(\kappa : Context, \epsilon : Env, \dots)$	
lit	$n \Rightarrow (n, \emptyset)$	
var	$\langle \epsilon : \sigma \rangle [x] \Rightarrow \langle \epsilon : \sigma \rangle [(\sigma x)] \quad if \ x \in dom(\sigma)$	
op	$(n_0, l_0) \oplus (n_1, l_1) \Rightarrow (n_0 \oplus n_1, l_0 \cup l_1)$	
and₁	$(tt, l_0) \ and \ (tt, l_1) \Rightarrow (tt, l_0 \cup l_1)$	
and₂	$(tt, l_0) \ and \ (ff, l_1) \Rightarrow (ff, l_1)$	
and₃	$(ff, l_0) \ and \ _ \Rightarrow (ff, l_0)$	
or₁	$(ff, l_0) \ and \ (ff, l_1) \Rightarrow (ff, l_0 \cup l_1)$	
or₂	$(ff, l_0) \ and \ (tt, l_1) \Rightarrow (tt, l_1)$	
or₃	$(tt, l_0) \ and \ _ \Rightarrow (tt, l_0)$	
ite₁	$(tt, l_0) ? e_t : e_e \Rightarrow addTags(e_t, l_0)$	
ite₂	$(ff, l_0) ? e_t : e_e \Rightarrow addTags(e_e, l_0)$	
tag	$tag(t, (v, l)) \Rightarrow (v, l \cup \{(t, v)\})$	
adt	$addTags((v, l_0), l_1) \Rightarrow (v, l_0 \cup l_1)$	

Imperative command syntax, context, and semantics:

$S ::=$	$skip \mid S ; S \mid if E then S else S \mid$	
	$Id := E \mid while E do S \mid end(List Id, TS)$	
$Context ::=$	$\dots \mid Id := Context \mid if Context then S else S \mid$	
	$Context; S \mid \langle \kappa : Context, \epsilon : Env, C : TS \rangle$	
asgn	$\langle \epsilon : \sigma \rangle [x := (n, l)] \Rightarrow \langle \epsilon : \sigma[x \leftarrow (n, l)] \rangle [skip]$	
seq	$skip; s_2 \Rightarrow s_2$	
cond₁	$\langle C : c \rangle [if (tt, l) then s_1 else s_2] \Rightarrow$	
	$\langle C : c \cup l \rangle [s_2; end(V, c)]$	
	where $V = (Assigned s_1). (Assigned s_2)$	
cond₂	$\langle C : c \rangle [if (ff, l) then s_1 else s_2] \Rightarrow$	
	$\langle C : c \cup l \rangle [s_1; end(V, c)]$	
	where $V = (Assigned s_1). (Assigned s_2)$	
while	$while(e) s \Rightarrow if (e) then(s; while(e) s) \ else skip$	
endcond₁	$\langle C : c' \rangle [end(nil, c)] \Rightarrow \langle C : c \rangle [skip]$	
endcond₂	$\langle \epsilon : \sigma, C : c' \rangle [end(x :: V, c)] \Rightarrow \langle \epsilon : \sigma', C : c' \rangle$	
	$[end(V, c)] where (\sigma x) = (n, l) \ and$	
	$\sigma' = \sigma[x \leftarrow (n, l \cup c')]$	
prog	$s \Rightarrow \langle \kappa : s, \epsilon : \emptyset, C : \emptyset \rangle$	

Dataflow program syntax, context, and semantics:

$EQ ::=$	$Id = E \mid id = pre(E)$	
$Prog ::=$	$(I, Env, List EQ)$	
$Context ::=$	$\dots \mid Context; List EQ \mid Context :: List EQ \mid$	
	$EQ :: Context \mid Id = Context \mid$	
	$Id = pre(Context) \mid \langle \kappa : Context, \tau :$	
	$List Env, O : List Env, \epsilon : Env, S : Env \rangle$	
comb	$\langle \epsilon : \sigma \rangle eqs_0. ((x = (n, l)) :: eqs_1) \Rightarrow$	
	$\langle \epsilon : \sigma[x \leftarrow (n, l)] \rangle eqs_0.eqs_1$	
state	$\langle S : \sigma \rangle eqs_0. ((x = (n, l)) :: eqs_1) \Rightarrow$	
	$\langle S : \sigma[x \leftarrow (n, l)] \rangle eqs_0.eqs_1$	
write	$\langle O : \kappa, \epsilon : c \rangle nil; eqs \Rightarrow \langle O : \kappa.[c], \epsilon : c \rangle eqs$	
cycle	$\langle \tau : \sigma_i :: l, \epsilon : _, S : \sigma_l \rangle eqs \Rightarrow$	
	$\langle \tau : i, \epsilon : (\sigma_i \cup \sigma_l), S : \emptyset \rangle eqs; eqs$	
prog	$\langle i, s, eqs \rangle \Rightarrow \langle \tau : i, O : nil, S : s, \epsilon : \emptyset, \kappa : eqs \rangle$	

$C : TS$ is introduced into the expression configuration to store the set of variable tags. Once a statement has been executed, the tags added to C by conditional statements will be removed. An *end* statement is introduced to implement that—it is appended to clear C and propagate the conditional tags to all variables assigned in the conditional body. A helper function (*Assigned s*) will then return the list of variables assigned in s . Given

a program (or program fragment) containing inputs, the rules defined in table 3 will determine the set of tags propagating to output.

Dataflow languages, such as Simulink and SCADE, are popular for model-based development, and assign values to a set of equations in response to periodic inputs. To store system state, state variables ($\frac{1}{z}$ blocks in Simulink) are used. Our dataflow language consists of assignments to combinatorial and state variables, and the semantics are defined over lists (traces) of input variable values. The expression configuration is extended to contain an input trace I , output trace O , and state environments S . Evaluation proceeds by cycles: at the beginning of a cycle, the *cycle* rule constructs the initial evaluation environment.

During a cycle, variable values are recorded using the *comb* and *state* rules. Note that the context does not force an ordering on evaluation of equations; instead, an equation can evaluate as soon as all variables it uses have been stored in the environment. When all equations have been computed, the *write* rule appends the environment to the output list. The *prog* rule, given an input list, an initial state environment, and a list of equations, initializes the configuration for the *cycle* rule. Coverage can be determined by examining the tags stored in the output environment list.

Note that both the tagging semantics are *optimistically inaccurate* with respect to observability; that is, they may report that certain conditions are observable when they are not. This is easily demonstrated by a code fragment:

```
if (c) then out := 0 else out := 0;
```

The semantic model of observability will correctly report that c is not observable; it cannot affect the outcome of this code fragment. However, the tagging model propagates the tags of c to the assignments in the *then* and *else* branches.

We have implemented observable versions of Branch, Condition, Decision, and MC/DC Coverage as part of model-based test generation for the Lustre dataflow language. However, the concept of observability is not restricted to any one language, generation paradigm, or product domain. The semantic model described in this section should be valid for any imperative language, such as C or Java. Ongoing research efforts are in progress to extend these ideas to Java and to assembly language. In addition, Colaco et al. have implemented observability through an extended tag semantics that takes into account certain non-Boolean faults as part of the Scade 6 language [45].

3.4 Implementation: Model-Based Test Generation

In model-based test generation, models are annotated with *trap properties*. A property of interest is negated, then the model checker returns a counterexample—a test input sequence demonstrating that the property can be met. In order to generate tests that meet the conditions

of observability, we need to be able to annotate the program with trap properties that track the tags described above. This is accomplished by conjoining the coverage obligations of the host criterion with a path condition representing the variable in which the test obligation’s target resides. Observability can be attained either *immediately*—within the current computational cycle—or *after a delay*. Path conditions must reflect either case. In this section, we describe this annotation for the Lustre dataflow language [9].

3.4.1 Immediate Non-Masking Paths

A variable x is observable if a computational path can be found from x to a monitored variable z in which x is not masked. If such a path can be taken entirely within one computational step, we call it an *immediate non-masking path*, and variable x is *immediately observable*. Such paths can be defined inductively by examining the variables that use x in their definition. If x is used in the definition of variable y , and x is not masked by other variables within that definition, then x is immediately observable at y . We then consider the variables that use y in their definitions, and apply the same criteria.

We track such notions by introducing additional variables. First, *combinatorial usage expressions*— $x_COMB_USED_BY_y$ —determine whether a variable is masked within a definition. The variable is true if x is not masked by other elements of y ’s definition. Second, *immediate observability expressions*— $x_COMB_OBSERVED$ —which offer a way to check the status of the non-masking path. For each Boolean variable in the program, there could exist one or more immediate non-masking paths.

Consider the code fragment in Figure 4, where out is an output variable, $in1$, $in2$, and $in3$ are input variables, and $v1$, $v2$, and $v3$ are internal variables.

```

v1 = in1 and in2;
v2 = if (in3) then v3 else v1;
v3 = not in2;
out = v1 or v2;

```

Fig. 4: Sample Lustre code

We can generate additional definitions to track the observability of variables as in Figure 5. Variable $v1$ is used by two variables— $v2$ and out —in their definitions and therefore has two potential immediate non-masking paths: directly through the output variable out or through $v2$. Variable $in2$ also has two potential immediate non-masking paths through its use in defining $v1$ and $v3$. All the other variables are each used once, so each has only one immediate non-masking path.

3.4.2 Delayed Non-Masking Paths

Reactive systems compute in cycles, and variable values from the previous cycle can be referred to. As a result, the effect of a variable on output may not be

```

in1_COMB_USED_BY_v1 = in2;
in2_COMB_USED_BY_v1 = in1;
in3_COMB_USED_BY_v2 = true;
v3_COMB_USED_BY_v2 = in3;
v1_COMB_USED_BY_v2 = (not in3);
in2_COMB_USED_BY_v3 = true;
v1_COMB_USED_BY_out = (not v2);
v2_COMB_USED_BY_out = (not v1);

out_COMB_OBSERVED = true;
in1_COMB_OBSERVED = ((in1_COMB_USED_BY_v1
    and v1_COMB_OBSERVED));
in2_COMB_OBSERVED = ((in2_COMB_USED_BY_v1
    and v1_COMB_OBSERVED) or
    (in2_COMB_USED_BY_v3 and
    v3_COMB_OBSERVED));
in3_COMB_OBSERVED = ((in3_COMB_USED_BY_v2
    and v2_COMB_OBSERVED));
v3_COMB_OBSERVED = ((v3_COMB_USED_BY_v2 and
    v2_COMB_OBSERVED));
v1_COMB_OBSERVED = ((v1_COMB_USED_BY_v2
    and v2_COMB_OBSERVED) or
    (v1_COMB_USED_BY_out and
    out_COMB_OBSERVED));
v2_COMB_OBSERVED = ((v2_COMB_USED_BY_out
    and out_COMB_OBSERVED));

```

Fig. 5: Introduced variables to track immediate non-masking paths

observed until several computation cycles after a value is computed. In each of these intermediate computational steps, the system state is stored in a delay variable, until it propagates to an output eventually. We call such a path—propagating influence through a delay variable to an output—a *delayed non-masking path* and the variable is *delay observable*. A delayed non-masking path can be built over multiple immediate non-masking paths: from a variable to a latch—a delay variable—then from the latch to another latch until an output is reached.

Suppose we have a sample code fragment in Figure 6, where $delay1$ and $delay2$ are delay expressions.

```

delay1 = (0 -> pre(in1));
v1 = (if (delay1 > 0) then true else in2);
delay2 = (false -> pre(v1));

```

Fig. 6: Sample Lustre code

As with immediate non-masking paths, we can inductively build paths involving delay expressions. An example can be seen in Figure 7. Variable $v1$, which uses $delay1$ and $in2$ in its definition, is used in the definition of delay expression $delay2$. Therefore, a delayed non-masking path from $delay1$ to $delay2$ is composed of the immediate non-masking path from $delay1$ to $v1$, then a delayed non-masking path from $v1$ to $delay2$.

This annotation gives us the means to track immediate paths to latches. However, it is still necessary to establish the means to knit these paths together to form the sequential path over one or more delays passed on the path to output. To do so, we introduce a *to-*

```

delay1_COMB_USED_BY_v1 = true;
in2_COMB_USED_BY_v1 = (not (delay1 > 0));

in1_SEQ_USED_BY_delay1 = true;
v1_SEQ_USED_BY_delay2 = true;
delay1_SEQ_USED_BY_delay2 =
  (delay1_COMB_USED_BY_v1 and
  v1_SEQ_USED_BY_delay2);
in2_SEQ_USED_BY_delay2 =
  (in2_COMB_USED_BY_v1 and
  v1_SEQ_USED_BY_delay2);

```

Fig. 7: Introduced variables to track delayed non-masking paths

ken mechanism—a special variable to mark the current delay location. Once the token is initialized to a delay variable x , it can non-deterministically move to any other delay location—as long as x can be sequentially used by that location. It can also move to a special TOKEN_OUTPUT_STATE, is a monitored variable is reached or TOKEN_ERROR_STATE is the token can no longer possible be observed through a monitored variable or another delay.

```

v1 = (false -> (not (pre v2)));
v2 = (false -> (pre v1));
v3 = (0 -> (if ((pre v3) = 3)
              then 0
              else ((pre v3) + 1)));
out = (((v1 and v2) and (v3 = 2)) or
        ((not (v1 and v2)) and (not (v3 =
        2))));
```

Fig. 8: Sample Lustre code

We generate token equations to track the path taken through delay variables. Consider the code fragment in Figure 8. We can then generate the token equations shown in Figure 9. In this case, if we are currently at TOKEN_D1, and $v1$ is immediately observable, then we reach the output. Otherwise, if $v1$ can be delay observed through $v2$, then the token moves to TOKEN_D3.

3.4.3 Test Obligations

Test obligations are the goals prescribed by an adequacy criterion, establishing properties deemed important to thorough testing. Consider the expression: $v1 = ((v2 \text{ and } in1) \text{ and } delay2)$. If we wanted to satisfy MC/DC coverage, we would need to establish a set of test cases where each condition ($v2$, $in1$, and $delay2$) is true and false, where the entire decision $v1$ evaluates to true and false, and where each condition is not masked within that decision. These obligations can be established as Boolean properties over the conditions. For example, we could achieve both outcomes for condition $v2$ and show non-masking with these two properties: $v2_AT_v1_TRUE = ((v2 \text{ and } in1) \text{ and } delay2)$ and $v2_AT_v1_TRUE = (((\text{not } v2) \text{ and } in1) \text{ and } delay2)$. If we can show that

```

token_next = (if ((pre token) =
  TOKEN_INIT_STATE) then token_first
  else (if ((pre token) =
  TOKEN_ERROR_STATE) then
    TOKEN_ERROR_STATE
  else (if ((pre token) =
  TOKEN_OUTPUT_STATE) then
    TOKEN_OUTPUT_STATE
  else (if ((pre token) = TOKEN_D1) then
    (if v1_COMB_OBSERVED then
      TOKEN_OUTPUT_STATE
    else (if ((token_nondet = TOKEN_D3)
      and v1_SEQ_USED_BY_v2)
      then TOKEN_D3 else TOKEN_ERROR_STATE))
  else (if ((pre token) = TOKEN_D2) then
    (if v3_COMB_OBSERVED then
      TOKEN_OUTPUT_STATE
    else (if ((token_nondet = TOKEN_D2)
      and v3_SEQ_USED_BY_v3)
      then TOKEN_D2 else TOKEN_ERROR_STATE))
  else (if ((pre token) = TOKEN_D3) then
    (if v2_COMB_OBSERVED then
      TOKEN_OUTPUT_STATE
    else (if ((token_nondet = TOKEN_D1)
      and v2_SEQ_USED_BY_v1)
      then TOKEN_D1 else TOKEN_ERROR_STATE))
  else TOKEN_ERROR_STATE)))));
```

Fig. 9: Example token equations.

each obligation—each property—is satisfied by at least one test case, we can show that the criterion is satisfied.

Observability-based test obligations conjoin the base obligations of the host criterion (e.g., MC/DC) with the path conditions required to establish either an immediate non-masking path or a delayed non-masking path from the expression where the base obligation is established to a monitored variable. An extension of MC/DC obligation $v2_AT_v1_TRUE$ to the equivalent Observable MC/DC obligation is shown in Figure 10.

```

v2_AT_v1_TRUE = ((v2 and in1) and delay2);
v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE =
  v2_AT_v1_TRUE and (v1_SEQ_USED_BY_delay1
  and token=delay1);
v2_AT_v1_TRUE_CAPTURED =
  v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE ->
  (v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE or
  pre(v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE));
obligation_0 = ((v2_AT_v1_TRUE and
  v1_COMB_OBSERVED) or
  (v2_AT_v1_TRUE_CAPTURED and token =
  TOKEN_OUTPUT_STATE));
```

Fig. 10: Sample test obligations

Expression $v2_AT_v1_TRUE$ is a base obligation from the host criteria, defining an MC/DC obligation in expression $v1$. For delayed non-masking paths, we have to define the instant in which the expression would be immediately observable at a delay (the moment of *capture*). We then must latch this fact for the remainder of execution, in case

the execution path hits a monitored variable. Expressions `v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE` and `v2_AT_v1_TRUE_CAPTURED` define this concept of capture for delayed non-masking paths. Finally, the full obligation is defined in expression `obligation_0`. In the obligation, the subexpression before the `or` operator defines immediate observability, and the second subexpression defines delayed observability. If either path is observed, then the obligation is met.

4 CASE STUDY

We wish to assess the quality—in terms of fault finding—of test suites generated to satisfy both observable and traditional versions of the studied coverage criteria. We also want to evaluate the effect of observability on the effectiveness of test suites. Thus, we address the following questions:

- 1) Which criterion has the highest average likelihood of fault detection?
- 2) Are test suites generated to satisfy observable variants of coverage criteria more effective than the test suites generated to satisfy the original criterion?

The first question allows us to establish a baseline for discussion, and a general ranking of criteria. Which criterion—whether observable or traditional—returns the best results, on average? In the second case, we wish to understand whether observability generally offers a *beneficial* effect—does it consistently improve the likelihood of fault detection?

Additionally, we are interested in the nature of the tests generated to satisfy observable and traditional coverage criteria, and the effect of adding observability constraints to a coverage criterion:

- 3) What impact does observability have on the generation cost, the average size of the generated test suites, and the average percentage of satisfied obligations for each criterion?
- 4) Across studied criteria, does observability have a consistent effect on factors such as likelihood of fault detection, oracle and structure sensitivity, and satisfiability of obligations?

Question 3 allows us to examine how the addition of observability impacts generation cost, suite size, and the ability of the test case generation process to satisfy the imposed test obligations. Question 4 allows us to examine the *impact of the choice of criterion*. Does it matter whether we start with MC/DC or Branch Coverage? Does observability consistently impact test suites?

In order to answer these questions, we have performed the following experiment:

- 1) **Gathered case examples:** We have assembled two sets of software models, written in the Lustre language (Section 4.1).
- 2) **Generated mutants:** We generated up to 500 mutants, each containing a single fault. (Section 4.2)
- 3) **Generated structural tests:** We generated test suites intended to satisfy Branch, Condition, Decision,

TABLE 4: Rockwell (non-inlined) example information

Model	# Inputs	# Internal Variables	# Outputs
DWM1	11	569	7
DWM2	31	115	9
Latctl_Batch	23	128	1
Microwave	13	162	4
Vertmax	40	30	2

TABLE 5: Rockwell (inlined) example information

Model	# Inputs	# Internal Variables	# Outputs	Average Complexity
DWM1	11	21	7	95.89285714
DWM2	31	10	9	21.36842105
Latctl_Batch	23	19	1	5.714285714
Microwave	13	99	4	9.15
Vertmax	40	30	2	720.5

and MC/DC Coverage—as well as observable variants of each—using counterexample-based test generation. (Section 4.3)

- 4) **Reduced test suites:** We generated 50 reduced test suites using the full test suite generated in the previous step. (Section 4.4)
- 5) **Computed effectiveness:** We computed the fault finding effectiveness of each test suite using both an output-only oracle and an oracle considering all program variables (a *maximally powerful* oracle) against the set of mutants. (Section 4.5)

4.1 Case Examples

In this study, we have made use of two pools of systems. The studied systems were originally modeled using the Simulink and Stateflow notations [33], [34]. Then, each was translated to the Lustre synchronous programming language [46] to take advantage of existing automation. In practice, Lustre would be automatically translated to C code. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

Note that Lustre systems, and the original Simulink and Stateflow systems from which they were translated, operate in a sequence of computational steps. In each step, input is received, internal computations are performed sequentially, and output is produced. Within a step, no iteration or recursion is done—each internal variable is defined, and the value for it computed, exactly once. The system itself operates as a large loop.

4.1.1 Rockwell Collins Dataset

The first set of systems consists of four industrial systems developed by Rockwell Collins engineers. Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). In addition, we have used a Microwave System—control software for a generic microwave oven developed as a non-proprietary teaching aid at Rockwell Collins. This set of benchmarks has been used in previous model-based test generation research [1], [3], [5], [11], [40], [47], [48], including previous work studying Observable MC/DC [10].

TABLE 6: Benchmark example information

Model	# Inputs	# Internal Variables	# Outputs	Average Complexity
6counter	1	4	1	3.5
AlarmFunctionalR2012	44	182	5	9.086666667
CarAll	2	8	1	4.125
cd	1	6	1	3.833333333
DockingApproach	13	1410	11	1.853754941
DragonAll	13	22	1	19.47619048
DragonAll2	13	27	1	20.77272727
durationThm1	5	7	1	3.333333333
ex3	2	5	1	3.6
ex8	2	5	1	3.4
fast_1	14	19	1	4.166666667
fast_2	14	30	1	4.37037037
FireFly	9	17	1	9.125
Gas	2	8	1	2.444444444
HysteresisAll	2	5	1	5.4
IllinoisAll	10	16	1	11.85714286
Infusion	20	861	5	2.745823389
MesiAll	4	10	1	5.545454545
Metros1	3	16	1	3.533333333
Microwave01	13	126	1	6.417647059
MoesiAll	5	12	1	4.071428571
PetersonAll	12	28	1	14.65517241
ProducerConsumerAll	4	12	1	3.153846154
ProductionCell	3	15	1	3.214285714
Readwrite	9	24	1	12.04
RtpAll	12	24	1	15.96
Speed2	2	5	1	3.6
Stalmark	1	3	1	21
SteamBoilerNoArr1	33	99	1	14.85
SteamBoilerNoArr2	19	3	1	30.666666667
Swimmingpool	8	21	1	8.1875
Switch	3	2	1	3.333333333
Switch2	3	2	1	3.333333333
SynapseAll	4	10	1	4.555555556
Ticket3iAll	13	20	1	11.45454545
Traffic	1	3	1	5.666666667
Tramway	4	23	1	2.727272727
TwistedCounters	1	4	1	5
Two Counters	1	3	1	2
UMS	5	39	1	2.837837838

Previous work has found that, due to masking, the structure of the model can have a significant impact on the resulting efficacy of generated test suites for MC/DC [1], [2]. In theory, observability can assist in overcoming masking. To study this, we have generated two variants of each of the Rockwell Collins systems:

- **Maximally Non-Inlined:** Each expression is as simple as it can possibly be, with sub-expressions split into independent intermediate variable calculations.
- **Maximally Inlined:** Each expression is as complex as it can possibly be, with no intermediate sub-expressions used.

We repeat the entire experiment with both variants, in order to more thoroughly study the interaction between program structure and observability.

Information related to the non-inlined version of each system is provided in Table 4, and information related to the inlined versions is provided in Table 5. In both cases, we list the number of input variables, number of internal variables, and number of output variables. The latter two numbers give an indication of the size of the model, as each internal and output variable corresponds to an expression that must be calculated each computational cycle. For the inlined versions, we also list the average complexity of the inlined expressions—that is, the average number of boolean operations in each expression.

4.1.2 Benchmarks Dataset

While the Rockwell Collins systems allow us to take a detailed look at the effect of program structure, the number of systems is relatively low. In order to more thoroughly analyze the effects of observability, we have also chosen an additional 40 systems from the open-

source Benchmarks dataset. Several of these models have been used in previous work, including a NASA example, *Docking_Approach*, which describes the behavior of a space shuttle as it docks with the International Space Station [11]. Two other systems, *Infusion_Mgr* and *Alarms*—which represent the prescription management and alarm-induced behavior of an infusion pump device—were also used in previous work [3], [11], [49].

The Benchmark Lustre models are available from
[https://github.com/Greg4cr/
Reworked-Benchmarks/tree/SingleNode](https://github.com/Greg4cr/Reworked-Benchmarks/tree/SingleNode).

Information related to each system is provided in Table 6, where we again list the number of input variables, number of internal variables, and number of output variables. In this case, we lack the original models, and cannot control the level of inlining. Therefore, we also list the average complexity of expressions to give an idea of how inlined each model is.

4.2 Mutant Generation

The following mutation operators were used in this study:

- **Arithmetic:** Changes an arithmetic operator (+, -, /, *, mod, exp).
- **Relational:** Changes a relational operator (=, \neq , $<$, $>$, \leq , \geq)².
- **Boolean:** Changes a boolean operator (\vee , \wedge , XOR).
- **Negation:** Introduces the boolean \neg operator.
- **Delay:** Introduces the delay operator on a variable reference (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value).
- **Constant:** Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating boolean constants.
- **Variable Replacement:** Substitutes a variable occurring in an equation with another variable of the same type.

The mutation operators used in this study are discussed at length in [50]. This method is designed such that all mutants produced are both syntactically and semantically valid. That is, the mutants will compile, and no mutant will crash the system under test.

Note that the type of mutants used in the evaluation in this report are similar to those used by Andrews et al., where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [27]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [26].

In order to control experiment costs, we do not use all possible mutants for each model. Instead, we employ

2. Some definitions of this operator also replace the entire expression with true or false. We do not do this in this experiment.

the following rule-of-thumb—if a model has fewer than 500 possible mutations, we use all possible mutations. If over 500 mutations are possible, we choose 500 of them for use in the experiment. In order to select mutants, we first gather a list of all possible mutations. Then, we use the proportions of each mutation type in the full set to select the number of mutants for the reduced set of 500, or a little bit greater than 500 due to some calculating error. Mutants of each type are then chosen randomly until the determined number are chosen for that type. This process prevents biasing towards particular types of mutations. Instead, the proportion of each fault type is maintained, despite not using the full set of mutations.

4.3 Test Data Generation

In this research, we explore four structural coverage criteria: Condition Coverage, Decision Coverage, Branch Coverage, and Modified Condition/Decision Coverage (MC/DC) [6], [19]. These criteria are defined in Section 2.2. For each criterion, we generate tests for both the traditional criterion as well as a version requiring observability. We refer to the observable versions of each criterion as Observable Condition Coverage (**OCondition**), Observable Decision Coverage (**ODecision**), Observable Branch Coverage (**OBranch**), and Observable MC/DC (**OMC/DC**).

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying the four coverage criteria and their observable variants [17], [51]. In this approach, each coverage obligation is encoded as a temporal logic formula in the model, and a model checker is used to produce a counterexample illustrating how the coverage obligation can be covered. This counterexample offers test input—a series of values for each input variable for one or more test steps. By repeating this process for each coverage obligation for the system, we can use the model checker to derive test sequences intended to achieve the maximum possible coverage of the model.

We have extended our model-based test generation framework³ to also generate test cases for observable criteria. This framework makes use of the JKind model checker [36], [52] as the underlying generation engine because we have found that it is efficient and produces tests that are easy to understand [53].

The test generation framework is available from
<https://github.com/MENG2010/lustre>.

4.4 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as

each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [54], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naïvely generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different reduced test suites for each case example.

Reduction is performed using a simple greedy algorithm. We determine the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been examined in the full set of tests.

4.5 Computing Effectiveness

In order to compute effectiveness of the generated test suites, we produce *traces* of execution by executing each test case against the original program and each mutant—recording the value of all variables at each step.

In our study, we use what are known as *expected value oracles* as our test oracles [3]. Consider the following testing process for a software system: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgment; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. Past experience with industrial practitioners indicates that such oracles are commonly used in testing critical systems, such as those in the avionics or medical device fields.

We explore the use of two formulations of expected value oracle: an *output-only (OO) oracle strategy* defines expected values for all outputs, and a *maximum (MX) oracle strategy* that defines expected values for all outputs and all internal state variables. The OO oracle strategy represents the oracle most likely to be used in practice. Both oracle strategies have been used in previous work, and we use both to allow for comparison [1], [3].

To give an example, consider the model defined in Figure 1. This model has a single output variable, *out1*. Therefore, the output-only oracle strategy would define an expected value for *out1*. The model also has three internal state variables—*v1*, *v2*, and *v3*. The maximum oracle strategy would define expected values for all three of those variables and the output variable *out1*.

To produce an oracle, we use the values of the monitored variables from the traces gathered by executing test cases on the original program, and we compare

3. Used in past projects, such as [1], [3], [11].

TABLE 7: Average percent of mutants killed for each pairing of criterion and oracle.

	Rockwell (I)		Rockwell (NI)		Benchmarks	
	MX	OO	MX	OO	MX	OO
OMC/DC	95.61%	87.03%	98.85%	85.88%	89.62%	65.14%
MC/DC	94.85%	84.94%	94.87%	53.89%	88.36%	57.13%
OCondition	88.38%	68.50%	98.95%	85.61%	86.22%	62.93%
Condition	71.00%	55.71%	93.38%	47.64%	79.37%	50.50%
ODecision	83.38%	60.33%	98.37%	83.44%	86.21%	64.70%
Decision	85.03%	53.48%	93.32%	45.73%	78.81%	49.26%
OBranch	81.92%	57.10%	96.84%	70.19%	85.47%	62.70%
Branch	84.20%	47.91%	87.17%	32.05%	73.19%	46.50%

those values to those recorded for each mutant. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or “killed”). For all studied systems, we assess the fault-finding effectiveness of each test suite and oracle combination by calculating the ratio of mutants killed to total number of mutants.

5 RESULTS AND DISCUSSION

In this section, we will address our research questions and discuss the implications of the results. As a reminder, we are interested in the following:

- 1) Which criterion has the highest average likelihood of fault detection? (Section 5.1)
- 2) Are test suites generated to satisfy observable variants of coverage criteria more effective than the test suites generated to satisfy the original criterion? (Section 5.2)
- 3) What impact does observability have on the generation cost, the average size of the generated test suites, the average percentage of satisfied obligations for each criterion? (Section 5.4)
- 4) Across the studied criteria, does observability have a consistent effect on test suites in terms of factors such as likelihood of fault detection, oracle and structure sensitivity, and satisfiability of obligations? (Section 5.5)

5.1 Overall Efficacy

Table 7 lists the average percentage of faults detected by test suites generated for each of the eight coverage criteria, separated by oracle strategy, for the Rockwell and Benchmarks datasets. From these results, we can see that—on average—test suites generated to satisfy OMC/DC tend to kill a larger percent of mutants than test suites satisfying all other coverage criteria. For both variants of the Rockwell systems—with any oracle strategy—test suites generated to satisfy OMC/DC kill the most mutants. The sole exception is for the non-inlined variant—with the maximum oracle strategy—where OCondition suites outperform OMC/DC by 0.1%. For the Benchmark models—with any oracle strategy—OMC/DC-satisfying suites have the highest average possibility of revealing faults.

Test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (MX oracle strategy) and 87.03% (OO oracle strategy) for the inlined Rockwell models, 98.85% (MX)/85.88% (OO) for the non-inlined Rockwell models, and 89.62% (MX)/65.14% (OO) for the Benchmarks models.

We can examine this question further through statistical analysis. To address this, we first formulate our hypothesis as follow:

H_1 : For each system in our study—with any oracle strategy—the OMC/DC criterion produces test suites with the highest likelihood of fault detection.

The paired null hypothesis is,

H_0 : For each system in our study—with any oracle strategy—the OMC/DC criterion produces test suites with a likelihood of fault detection drawn from the same distribution as another criterion’s suites.

We have performed a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [55], a non-parametric hypothesis test used to determine whether two independent samples were selected from populations having the same distribution, to verify our hypothesis. Since we cannot generalize across non-randomly selected case examples, we apply the statistical test over pairs of coverage criteria (i.e., any of the coverage criteria versus the rest of the coverage criteria respectively, therefore, we have 56 pairs of metrics in total), for each pairing of model and oracle type, with $\alpha = 0.05$.

The statistical results are presented in Table 8. In this table, we list the percentage of cases for each dataset where we can reject H_0 —that is, where we can confirm that OMC/DC outperforms the compared criterion. We also list the percentage of cases where the reverse is true—where we can state that the other criterion outperforms OMC/DC with significance. For example, for the Rockwell (Non-inlined) models, with an output-only oracle strategy, OMC/DC outperforms all criteria except OCondition in 100% of cases, with statistical significance.

For Benchmarks, with any oracle strategy, the percentage of cases where OMC/DC suites outperform suites satisfying other coverage criteria is always higher than the percentage of suites satisfying other criteria outperforming OMC/DC suites. That is, OMC/DC always has a highest average likelihood of fault detection. This is also true in all situations for both variants of the Rockwell models with an output-only oracle strategy. Results are a little less clear-cut for the Rockwell models when paired with a maximum oracle strategy, where other criteria occasionally tie or outperform OMC/DC. For instance, for the inlined variants, ODecision, OBranch, and Branch suites outperform OMC/DC suites as often as OMC/DC suites outperform their counterparts.

Intuitively, these results makes sense. There is a clear boost in performance from the addition of observability.

TABLE 8: Percent of cases where OMC/DC suites outperform suites satisfying other criteria with significance, and where suites satisfying other criteria outperform OMC/DC suites.

		MX Oracle		OO Oracle	
		More Effective	Less Effective	More Effective	Less Effective
Benchmarks	ODecision	45.00%	10.00%	45.00%	15.00%
	OCondition	52.50%	15.00%	47.50%	15.00%
	OBranch	37.84%	28.95%	45.95%	21.05%
	MC/DC	45.00%	15.00%	55.00%	5.00%
	Decision	57.50%	2.50%	67.50%	2.50%
	Condition	65.00%	5.00%	72.50%	2.50%
	Branch	65.79%	7.69%	71.05%	7.69%
Rockwell (Inlined)	ODecision	40.00%	40.00%	80.00%	20.00%
	OCondition	60.00%	0.00%	100.00%	0.00%
	OBranch	20.00%	20.00%	80.00%	20.00%
	MC/DC	40.00%	20.00%	80.00%	20.00%
	Decision	40.00%	20.00%	80.00%	20.00%
	Condition	80.00%	20.00%	80.00%	0.00%
	Branch	20.00%	20.00%	80.00%	20.00%
Rockwell (Non-inlined)	ODecision	80.00%	20.00%	100.00%	0.00%
	OCondition	20.00%	20.00%	40.00%	20.00%
	OBranch	40.00%	60.00%	100.00%	0.00%
	MC/DC	80.00%	0.00%	100.00%	0.00%
	Decision	100.00%	0.00%	100.00%	0.00%
	Condition	100.00%	0.00%	100.00%	0.00%
	Branch	60.00%	40.00%	100.00%	0.00%

As Table 7 shows, the observable versions of criteria almost always outperform both their non-observable counterpart and *all other non-observable criteria*, except the original MC/DC. MC/DC suites outperform all of the other non-observable versions of the studied criteria, and is the only non-observable criterion to produce suites that occasionally outperform the observable counterparts. The addition of observability boosts the efficacy of the generated test suites, generally with the end result that Observable MC/DC produces the most effective test suites. OMC/DC does not always produce the best suites, but it is the safest choice of the studied criteria.

Across the board, efficacy tends to be higher for the maximum oracle strategy, and the gap between observable and non-observable criteria tends to be less. This can be explained by examining the concept of masking. With an output-only oracle strategy, input must trigger a fault, and the effect of a fault must not be masked by expressions on the path to the output. Observability is intended to overcome masking, and clearly does assist—given the results for output-only oracles. However, with a maximum oracle strategy, we already have expression-level observability. Masking along the path to the output does not need to be overcome. The observable criteria generally produce more effective suites even in these cases, but the possibility for improvement is smaller.

In general, however, maximum oracles are prohibitively expensive to employ [3]. A tester would need to specify expected values for all variables, for each test step. This is not usually a realistic goal. Output-only oracles are the most common, and OMC/DC appears to be the most effective criterion when paired with this common oracle strategy.

5.2 Efficacy Impact of the Addition of Observability

In Table 9, we present the average improvement in efficacy when moving from a traditional criterion—such as MC/DC—to its observable counterpart over all models for each dataset. Choosing the test input that would

TABLE 9: Average improvement in the likelihood of fault detection, after adding observability constraints

		MX Oracle	OO Oracle
Benchmarks	MC/DC	1.79%	53.38%
	Decision	12.78%	88.03%
	Condition	11.12%	132.79%
	Branch	26.44%	163.10%
Rockwell (Inlined)	MC/DC	0.80%	2.77%
	Decision	-1.80%	14.81%
	Condition	42.92%	32.18%
	Branch	-3.08%	22.13%
Rockwell (Non-inlined)	MC/DC	4.58%	351.12%
	Decision	5.95%	392.44%
	Condition	6.46%	384.93%
	Branch	12.81%	389.99%

reveal a fault for a given test oracle is an undecidable task. However, the intent of observability is to increase the likelihood of detection by overcoming masking. The results show the merit of this idea—there is generally an increase in efficacy. Regardless of the underlying coverage criterion, observability seems to have a positive impact on the likelihood of detecting faults.

This is especially true when an output-only oracle—the most common oracle strategy [3]—is used. When using an output-only oracle, masking is a major problem. Our results show that observability can overcome masking. This can be clearly seen in the Rockwell (non-inlined) models, where the addition of observability improves efficacy up to 392.44%. Results are more subdued for the inlined variants—up to a 32.18% improvement.

We can see from these results that the structure of the system—how code is written—has some impact on the impact of adding observability. The Rockwell examples offer two extremes—either entirely inlined or with the simplest possible expressions. At the later end, there is tremendous improvement from adding observability. If there are a large number of simple expressions, then masking along the path to the output is far more likely than if there are a smaller number of expressions. As a result, observability has a major impact, propagating the effect of a fault to the output variables. On the other hand, if there are a small number of expressions, then the

path to output will be shorter. Therefore, observability will have a smaller impact.

In addition, past work has shown that the test cases generated for heavily inlined systems may be more effective from the start [1], [2]. Criteria such as MC/DC require that an independent impact be shown for each condition within a decision. That is, if MC/DC is fulfilled, then a condition will not be masked within the expression that it appears in. Its impact can be masked on the path to output, but will affect the outcome of the decision that it falls within. If a model is more heavily inlined, then the requirements of standard MC/DC are more strenuous—Independent impact must be shown for more complex expressions. At the same time, the path to output is shorter, limiting further opportunities for masking. Therefore, the test cases may be more effective from the start, and the further impact of observability may be more limited.

We can see some evidence from this that observability helps bridge the gap from output-only oracle to maximum oracle—without adding additional human oracle cost [56], and the gap from non-inlined to inlined program structure. The Benchmarks examples are varied in terms of structure. As a result, the impact of adding observability falls between the two extremes of the Rockwell models—with improvements of up to 163.10% for the output-only oracle strategy.

As noted earlier, improvements tend to be smaller when employing a maximum oracle strategy. For non-inlined implementation of Rockwell models, we see average improvements of up to 12.81%. For the inlined variants, we see up to a 42.92% average improvement, and even see small performance downgrades of up to 3.08%. For the Benchmarks dataset, we see average improvements of up to 26.44%.

Adding observability improves efficacy over satisfaction of traditional criteria, with average improvements of 11.94% with a maximum oracle and 125.98% with the output-only oracle (with per-model improvements of up to 1760.52%).

We can establish evidence by performing statistical analysis, employing the same test used previously. We formulate our hypotheses as follow:

H_2 : For each system and oracle, the observable version of a criterion produces test suites with a higher likelihood of fault detection than the traditional variant.

The paired null hypothesis is:

$H_{\theta 2}$: For each system in our study—with any oracle strategy—the observable version of a criterion produces test suites with a likelihood of fault detection drawn from the same distribution as the traditional variant.

The statistical results are presented in Table 10, where we list the percent of cases where we can reject $H_{\theta 2}$ —we can provide evidence that the observable criterion produces more effective test suites—along with the per-

centage of cases where we can state with significance that the reverse is true—that the traditional criterion is more effective. For example, for the Benchmark models—with a maximum oracle strategy—suites satisfying the OM-C/DC criterion outperform MC/DC-satisfying suites with significance in 45% of cases, while the reverse is true for only 15% of cases. For the remaining 40% of the models, neither outperforms the other with significance.

Almost universally, the observable variant outperforms the traditional variant—with significance—in more cases. The only two situations where this is reversed are for Decision Coverage and Branch Coverage on the inlined Rockwell models, paired with a maximum oracle strategy. As highlighted above, this is the exact situation where we would expect the least benefit from the addition of observability. However, with the more realistic output-only oracle strategy, the observable variant of the criterion produces more effective suites in the vast majority of cases.

5.3 Factors That Influence Efficacy

Observability tends to improve the efficacy of test suites. However, it does not do so in all cases, and the gains in efficacy are not consistent across all models. Therefore, it is worth examining the factors that influence the efficacy of observability. We can illustrate some of these factors by looking at situations when observability had a minimal—or, worse, a negative—impact on efficacy (listed in Table 11).

As the percentage of fulfilled obligations decreases, the efficacy of the resulting suite decreases as well. The observable versions of criteria impose much more difficult obligations to satisfy, so some drop is not surprising—either from provably infeasible obligations or obligations that the test generator is unable to address. However, if the loss in obligation satisfaction is major, then we will observe some loss in performance. The worst case of this is the Docking_Approach example, where—for OCondition Coverage—we lose 89% satisfaction of the obligations. This resulted in a 31% loss in efficacy.

At first glance, the structure of the model—the level of inlining—appears to have some impact. With the Rockwell models, we never see a downgrade in performance for the non-inlined variants. We do see occasional efficacy losses for inlined models (including some of the “worst” examples from the Benchmarks set). However, we do not believe that this is due to inlining alone—for instance, Docking_Approach is not inlined—but because inlining is a factor informative of *model complexity*.

Inlined models tend to see less improvement from observability because they have more complex expressions. Regardless of the length of the path to output, complex expressions suffer more from masking, making it harder to guarantee a clear path to output. In turn, this potentially leads to lower levels of overall satisfaction for the observable variants. However, even though Docking_Approach is not inlined, it does have a deep

TABLE 10: Cases where observable criterion produces suites outperforming non-observable variant with significance, and when the non-observable variant is more effective.

		MX Oracle		OO Oracle	
		Observable	Traditional	Observable	Traditional
Benchmarks	MC/DC	45.00%	15.00%	55.00%	5.00%
	Decision	57.50%	5.00%	67.50%	2.50%
	Condition	65.00%	7.50%	67.50%	7.50%
	Branch	60.53%	5.26%	71.05%	2.63%
Rockwell (Inlined)	MC/DC	40.00%	20.00%	80.00%	20.00%
	Decision	20.00%	40.00%	60.00%	20.00%
	Condition	60.00%	20.00%	80.00%	20.00%
	Branch	0.00%	40.00%	80.00%	20.00%
Rockwell (Non-inlined)	MC/DC	80.00%	0.00%	100.00%	0.00%
	Decision	100.00%	0.00%	100.00%	0.00%
	Condition	100.00%	0.00%	100.00%	0.00%
	Branch	60.00%	0.00%	100.00%	0.00%

TABLE 11: Downgrade or lowest upgrade in efficacy when transitioning from traditional to observable criteria.

		MX Oracle		OO Oracle	
		MC/DC	Decision	Condition	Branch
Benchmarks	MC/DC	-11.7%, PetersonAll		-47.41%, Car_All	
	Decision	-9.35%, MoesiAll		0.00%, 6counter/Metros1/ProductionCell/Stalmark/Switch/Switch2/Ticket3iAll/Tramway/TwistedCounters/UMS	
	Condition	-31.62%, DockingApproach		-22.75%, DockingApproach	
	Branch	-3.64%, Rtp_All		-8.17%, Rtp_All	
Rockwell (Inlined)	MC/DC	-1.42%, DWM1		-0.91%, DWM1	
	Decision	-6.43%, DWM1		-0.71%, DWM2	
	Condition	-7.67%, DWM1		-0.46%, DWM1	
	Branch	-11.65%, DWM1		-6.53%, DWM2	
Rockwell (Non-inlined)	MC/DC	0.04%, Latctl_Batch		12.42%, DWM2	
	Decision	0.92%, DWM2		27.77%, DWM2	
	Condition	0.67%, DWM2		18.47%, DWM2	
	Branch	0.00%, Latctl_Batch/Vertmax		40.69%, Latctl_Batch	

state space—a series of gated conditions—which results in a longer path to establish to ensure observability. Therefore, we get lower satisfaction of the obligations for the observable variant than the original, which must simply satisfy obligations on individual expressions. The problem, then—inlined or not—is establishing a masking-free path from the expression to the output.

Non-inlined models can offer complex observability requirements because the path length is long—a failure must propagate through a long series of expressions to impact the output. Each individual expression is simple, but there are a large number of them to pass through unmasked. Therefore, the *path length* can be informative of the difficulty of achieving observability—impacting both obligation satisfaction and efficacy. In non-inlined models, the individual statements are simple. This results in trivial satisfaction of traditional coverage criteria, and weaker tests. Even if tests trigger a fault, they tend to be masked on the path to output. As a result, there tends to be a greater performance boost from observability.

Inlined models tend to have a shorter path-to-output, but each expression is more complex. Therefore, at each expression from activation to output, a failure could be easily masked. *Statement complexity*—which can be judged by the level of inlining—impacts obligation satisfaction as well as the efficacy gap between observable and traditional variants. Suites satisfying the traditional criterion must satisfy much more difficult obligations, and there are fewer opportunities for masking on the path to output. Therefore, the efficacy of the suites satisfying the traditional criterion tend to be relatively effective even without observability. Observability can

boost efficacy, but the difficulty of finding a path through the more complex expressions can also cause issues.

The above only discussed combinatorial paths—from expression to output in a single computation cycle. Complexity must also be considered over multiple computation cycles, as observability can be established *after delays*. One additional factor impacting the path to output are the *number of delay expressions*. Failures can be propagated across computation cycles. However, the use of such expressions introduces an additional source of complexity to a model, and test obligations that require a delay observable path can be harder to satisfy.

In cases where the loss in performance—or gain—are small, one factor that may contribute is the test suite reduction process. Tests are chosen randomly for the reduced suite, based on their ability to cover obligations. In general, efficacy may be essentially identical between the observable and non-observable suites, and poor test cases choices push the average slightly lower—but not in a statistically significant manner. This would explain most of the inlined Rockwell scenarios, as well as Branch-satisfying suites on the Rtp_All system from the Benchmarks dataset. This is a case where Branch and OBranch attain generally the same results—the *if* statements in the model are easily observable—but the average for OBranch is slightly lower due to poor test selection during suite reduction.

Factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the

TABLE 12: Average change in generation cost, test suite size, and percentage of satisfied obligations when moving from a criterion to the observable version.

		Size of Test Suites	Obligation Satisfaction	Generation Cost
Benchmark	MC/DC	23.59%	-22.37%	346.75%
	Decision	63.66%	-12.38%	173.43%
	Condition	52.61%	-20.02%	247.35%
	Branch	69.68%	-8.02%	129.39%
Rockwell (Inlined)	MC/DC	25.51%	-4.20%	2422.91%
	Decision	23.02%	-5.72%	422.04%
	Condition	49.97%	-4.17%	251.08%
	Branch	7.73%	-3.29%	551.48%
Rockwell (Non-inlined)	MC/DC	307.88%	-6.49%	996.48%
	Decision	392.46%	-8.11%	1285.85%
	Condition	376.25%	-6.30%	1081.58%
	Branch	343.54%	-4.82%	116.99%

TABLE 13: Median time (in seconds) required to generate test suites for each criterion and its observable version.

		Observable	Traditional
Benchmarks	MC/DC	24.81	5.55
	Decision	8.16	2.98
	Condition	13.78	3.97
	Branch	5.55	2.42
Rockwell (Inlined)	MC/DC	880.96	34.92
	Decision	38.26	7.32
	Condition	98.97	28.19
	Branch	37.27	5.72
Rockwell (Non-inlined)	MC/DC	195.49	17.83
	Decision	92.64	6.68
	Condition	104.48	8.84
	Branch	9.72	4.48

length of the combinatorial path from expression to output, and the length of the delayed path from expression to output.

5.4 Impact of Observability on Generation Cost, Test Suite Size, and Obligation Satisfaction

Each test case can satisfy more than one test obligation, depending on the input applied and the execution path taken by applying such input. Observable criteria require the same number of test obligations as their host criterion, but impose more difficult requirements for fulfillment. Rather than simply examining the number of test obligations required by a criterion, we can look at the size of the minimal test suites required to satisfy such criteria and the time cost to generate such test suites as an indicator of the difficulty of meeting testing goals.

In Table 12, we present the average change in generation cost, size of test suites, and percentage of fulfilled obligations when observability is required for each coverage criterion. Table 13 presents the median time required to generate test suites for each criterion. From these two tables, we can see that an observable criterion tends to require significantly more time to generate test suites than when the corresponding host criterion is used on its own. Such criteria yield more complex obligations, resulting in a more complex generation process where the model checker must spend more time identifying a solution that yields a non-masking path.

However, from Table 13, we can also see that test generation can still be completed in a reasonable time frame

for observable criteria. For the Benchmarks dataset, the median generation time still tops under thirty seconds. The worst median—for Observable MC/DC over the inlined Rockwell models—is still under fifteen minutes. Compared to the cost of manual test case creation, the addition of observability is minor.

The addition of observability requires a longer test generation process, with average increases ranging from 129.39%-2422.91%.

As we can see from Table 12, regardless of the underlying coverage criterion, we see an increase in the number of test cases required for the observable version of the test suite. Fundamentally, observable criteria require more test cases to fulfill their obligations than the traditional variants. Because of the highly specific path to output required for each obligation, there is less overlap between test cases in terms of the obligations satisfied.

Program structure seems to have an impact on the magnitude of the size increase. Moving to an observable criterion results in a massive increase in test suite size for the non-inlined Rockwell models—307.87-392.46%—while there is only a modest increase of 7.73-49.97% for the inlined models. On a per-model basis for the Benchmarks examples, many of the models with smaller increases in suite size tend to also be heavily inlined.

This observation makes sense given the discussion above. The obligations for non-observable criteria are formed over individual expressions. If those expressions are simple, the obligations too will be simple. As a result, each test case may cover a variety of obligations with ease. If the model is more heavily inlined, then each obligation will be more complex, and more specialized. There will be less overlap in coverage between test cases [1]. The more heavily inlined the model, the larger the test suite tends to be.

Therefore, model structure has a major impact on the size of the test suite for suites satisfying the *non-observable* criteria. Inlined models start with larger test suites. Then, regardless of the model structure, the addition of observability, increases the size further.

The primary factor influencing the size increase from observability is the length of the path. Each expression encountered along the path imposes additional conditions on maintaining a non-masking path. Therefore, the longer the path length, the more complex the requirements are on the test case. As a result, we see a similar effect to changing the program structure. The individual test cases are more specialized, and there is less of a chance of overlap in covered obligations. This further explains the larger increase in size for non-inlined models. Non-inlined models have simpler expressions, but much more of them. As a result, the satisfaction complexity is in satisfying path constraints rather than in fulfilling the original expression-level obligations.

The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the path from each expression to the output.

Previous work has shown that observability imposes an additional complexity burden on the test case generator, generally resulting in some loss in obligation satisfaction [10], [11]. The results of this study further confirm this. Table 12 shows that—on average—there is a loss in obligation satisfaction regardless of the criterion. For the inlined Rockwell models, this average loss ranges from 3.29-5.72%. For the non-inlined versions, this ranges from 4.82-8.11%. Then, for the Benchmarks dataset, the average loss ranges from 8.02-22.37%.

As discussed earlier, a loss in obligation satisfaction is to be expected—the test obligations require to ensure observability are far more complex than the equivalent obligations when observability is not required. This loss can occur for two reasons. First, if there is no masking-free path, then the obligation will be unfulfillable. This means that observability cannot be established, and thus, a fault in that statement *cannot* influence the output. Generally, this indicates dead code—code that, intentionally or not, cannot affect program output. Occasionally, this is a byproduct of either code reuse—where existing code is reused wholesale—or defensive programming.

However, in some cases, obligations may be too complex for the test generator to fulfill. In such cases, the generator will eventually return an “unknown” verdict. This is an indication that the generator was unable to meet the obligation, and was unable to conclusively determine that it could not be met (the case above). If the obligations are too complex, then the test generator can return weaker test suites because it eventually gives up on finding solutions that fulfill these obligations.

To better understand the reasons we lose coverage, we have listed the average percent of obligations fulfilled and the percent of obligations that resulted in “unknown” verdicts—where the test generator gave up on finding a solution—for each dataset in Table 14. First, we can see again that the observable variants see a lower rate of obligation fulfillment than the traditional criteria. Again, this is expected. In the case of the Rockwell models, we see that there are no situations where the test generator returned an unknown verdict. This means that any loss in such situations is due to *provable* unfulfillable obligations—dead code. This reduction in fulfillment is acceptable, as such obligations can never be fulfilled.

However, for the complex Benchmarks models, we do see some loss due to the test generator. On average, 4.11% (OBranch), 8.46% (OCondition), 3.90% (ODecision), and 7.41% (MC/DC) of obligations return “unknown” verdicts during test generation. We wish to avoid such situations, as they are situations where we cannot prove that the obligation cannot be fulfilled—the

test generator just did not find a solution in time. Some of these obligations may have test cases meeting them. Many will not, but we lack proof in either case.

In the Benchmarks dataset, even the traditional criteria have obligations that result in unknown verdicts—on average, 0.09% (Branch), 0.33% (Condition), 0.09% (Decision), and 0.66% (MC/DC) of the obligations time out. However, these percentages are far lower than for the observable variants. This speaks to the complexity of establishing observability, which is often far beyond that of covering the obligations of the host criterion.

The two driving factors in these unknown verdicts are the length of the combinatorial path from expression and output and the number of delay expressions—the length of the delayed path—between the expression and the output. Both increase the complexity of finding a masking-free path between the expression that is the source of the base obligation and an output variable. If the path is more complex, the generator will have a harder time satisfying the test obligations.

Although paths are shorter in inlined models, the individual expressions are more complex than in non-inlined models. Although expressions are simple in non-inlined models, the paths are longer than in inlined models. As a result, the level of inlining does not play a major role in the loss in obligation satisfaction. The level of correlation between inlining and loss in satisfaction is relatively low. The length of the path—whether delayed or immediate—is of far more importance.

The addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to solve.

5.5 The Effect of Observability

For our final research question, we wish to take a look at the effect of observability *itself*. Regardless of the underlying host criterion, does observability have a consistent impact on suite efficacy, oracle sensitivity, structure sensitivity, and obligation fulfillment?

5.5.1 The Choice of Host Criterion

The choice of coverage criterion is often made based on the perceived strength of that criterion. MC/DC is more strenuous to fulfill than Branch Coverage, and therefore, suites satisfying it *should* be more effective. While there are exceptions, this generally bears out in practice. In our study, MC/DC satisfaction results in stronger test suites than Branch Coverage satisfaction.

However, one question we are curious about is—when observability is required, *does the choice of host criterion matter?* Does observability consistently improve results, and is there still reasonable differentiation in the final results to see an impact from the choice of host criterion.

TABLE 14: Average % of obligations fulfilled, and the average % of the unfulfilled obligations that were due to an “unknown” verdict being returned by the test generation for each dataset.

	Branch		OBranch		Condition		OCondition		
	% Fulfilled	% Unknown							
Benchmark	94.77%	0.09%	87.88%	4.11%	96.22%	0.33%	75.78%	8.46%	
Rockwell (inlined)	97.88%	0.00%	94.94%	0.00%	98.88%	0.00%	94.92%	0.00%	
Rockwell (non-inlined)	100.00%	0.00%	95.18%	0.00%	99.83%	0.00%	93.58%	0.00%	
Decision		ODecision		MC/DC		OMC/DC			
Benchmark	% Fulfilled	% Unknown							
Rockwell (inlined)	94.60%	0.09%	81.17%	3.90%	86.84%	0.66%	67.08%	7.41%	
Rockwell (non-inlined)	98.93%	0.00%	93.50%	0.00%	96.66%	0.00%	92.70%	0.00%	
	99.95%	0.00%	91.85%	0.00%	99.48%	0.00%	93.07%	0.00%	

From the results in Tables 7 and 9, we can still see that the choice of criterion matters. Observability generally results in better test suites, but there is no real consistency in the magnitude of that impact across criteria, oracles, and system structures. The choice of criteria does impact the end result. OMC/DC satisfaction does tend to result in better test suites than OBranch satisfaction. The gap between criteria is often narrower for the observable variants than their traditional variants, but there is still a gap. Therefore, we can conclude that the choice of host criterion still influences the final result.

With traditional coverage criteria, weaker criteria may be used because they offer *enough* benefit, but are less expensive to fulfill. This is particularly true when test cases are written by human developers. Branch Coverage is easier to understand and explain than MC/DC, and proving that your test cases meet the more strenuous requirements of MC/DC requires more time and effort. If satisfaction of Branch Coverage can be achieved within the time period allotted to testing and offers benefits to the testing process, it may be better to make use of it than to spend the same amount on time attaining partial coverage of MC/DC. Even in the case of automated generation, it may be reasonable to choose to maximize Branch Coverage over attaining partial coverage of MC/DC. If the test generator is unable to satisfy the requirements of MC/DC, then attaining a higher level of Branch Coverage could lead to better efficacy.

However, this same trade-off does not necessarily function in an equivalent manner once observability is required. As we can see from the discussion in Section 5.4, the added complexity of observability vastly outweighs the complexity added by the use of a criterion such as MC/DC over Branch Coverage. If the test generation framework employed in this study can satisfy Branch Coverage for a model, it can usually attain similar levels of MC/DC. There is a far more perceptible drop when moving to any of the observable criteria. A gap still exists between Observable Branch and Observable MC/DC, but the leap from non-observable to observable is much greater.

It follows then that—rather than asking which criterion to employ—the more important questions is whether to require observability. In the context of manual test creation, employing observability without tool support is likely to be too expensive to consider in any situation except when safety is absolutely crucial. In the case of automated generation, observability is—at

TABLE 15: Average improvement in mutation detection when changing from OO to MX oracle strategy.

	Benchmarks	Rockwell (Inlined)	Rockwell (Non-inlined)
OMC/DC	96.77%	10.73%	16.31%
ODecision	93.33%	49.94%	19.38%
OCondition	96.81%	34.53%	16.83%
OBranch	102.69%	59.89%	40.63%
MC/DC	208.76%	13.03%	352.92%
Decision	233.13%	78.20%	384.62%
Condition	297.46%	31.45%	375.41%
Branch	278.98%	97.67%	412.78%

least for the studied programs—reasonable to require. Although there are situations where the loss in coverage due to unknown verdicts is unacceptably high, for most of the studied programs there were clear benefits.

These results also show that—as long as the test generator can handle the complexity of observability at all—the additional loss from choosing a more complex host criterion is minor. Therefore, we would recommend the use of stronger criteria such as MC/DC over weaker ones when observability can be handled by the test generator. That said, if the combined complexity of observability and criterion is too much for generation to handle, then a tester could first change the host criterion—then drop the observability requirement.

The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself.

Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity.

5.5.2 Oracle Sensitivity

In normal situations, the results of testing are sensitive to the choice of variables monitored as part of the test oracle. We can see this in comparing the results of the maximum and output-only oracle strategies for suites satisfying the traditional non-observable criteria. When results are checked with the maximum oracle strategy, efficacy tends to be much high. This is because masking can prevent program elements from influencing other variables. With any oracle strategy other than the maximum oracle strategy, suite efficacy depends on the selection of variables monitored by the oracle [3]. This complicates the testing process, as it is not obvious *which* variables should be monitored, and coming up with

expected values for any variables other than the output variables can be very difficult.

In theory, observability should be a powerful tool in overcoming oracle sensitivity. By requiring a masking-free path from any targeted expression to the output, we should be able to increase the efficacy of using an output-only oracle strategy. In Table 15, we present the average improvement in fault finding when moving from an output-only oracle strategy to the maximum oracle strategy for each coverage criterion, and for each of the three datasets.

From these results, we can see that for the non-inlined Rockwell systems, oracle sensitivity is *greatly* reduced when we require observability—for instance, Branch-satisfying suites improve by 412.78% when changing oracle strategies, but OBranch-satisfying suites only improve by 40.63%. As discussed earlier, non-inlined systems tend to have a large number of simple expressions and long paths to output. These results make sense. The maximum oracle strategy monitors every single expression in the program. Therefore, the size of the maximum oracle is much larger than the output-only oracle, as it is much easier to detect faults. When paired with an output-only oracle strategy, suites satisfying traditional criteria will suffer greatly from masking. Observability overcomes this masking by requiring that each expression be able to influence the output.

We do not see the same magnitude of effect for the aggressively inlined versions of the Rockwell models. Except in the case of Condition Coverage, there is a reduction in oracle sensitivity, but the impact is less. Again, however, these results make intuitive sense. An inlined implementation has fewer expressions. Therefore, the maximum oracle is also smaller—with fewer points of observation. The observable versions of criteria still produce suites that are less sensitive to the choice of oracle strategy, but there is also potentially less oracle sensitivity to overcome in the first place.

The Benchmark models again fall between the two extremes. The suites satisfying the observable criteria are less sensitive to the choice of oracle strategy than suites satisfying traditional counterparts. Suites satisfying traditional Branch Coverage improve by 273.10% from the shift in oracle strategy, while suites satisfying OBranch Coverage only improve by 102.51%.

Observability reduces sensitivity to the choice of oracle strategy by ensuring a masking-free path from expression to monitored variables.

5.5.3 Structural Sensitivity

Traditional coverage criteria—particularly MC/DC—are known to be sensitive to program structure [1], [2]. With an output-only oracle strategy, suites generated using the inlined version of the program will be far more effective at finding faults than suites generated using the

TABLE 16: Average change in efficacy when switching from non-inlined to inlined versions of Rockwell models.

	Max Oracle	OO Oracle
OMC/DC	-3.32%	1.50%
ODecision	-15.23%	-25.41%
OCondition	-10.71%	-18.56%
OBranch	-16.70%	-20.47%
MC/DC	0.18%	354.62%
Decision	-7.83%	344.09%
Condition	-22.54%	332.22%
Branch	-5.83%	332.24%

non-inlined version of the program. Because individual expressions are more complex in the inlined program, their test obligations are more complex. There are also, often, fewer opportunities for masking on the path to output, as there are fewer expressions along that path. Observability should help overcome that sensitivity to structure. Although the individual expressions are simpler, overcoming masking along the path should result in a more robust test suite.

In our experiments, this seems to be the case. Table 16 lists the average change in efficacy when switching from non-inlined to inlined version of the Rockwell models. On average, we see that—for the traditional suites—there is a major improvement in efficacy when we change program structures. For suites satisfying the observable variants, we actually see a slight downgrade in performance.

For the traditional criteria, if we use a maximum oracle strategy, we see a downgrade in performance when changing program structure instead of the upgrade we saw with an output-only oracle strategy. This is because, with a non-inlined program, the size of the maximum oracle is very large. Each of the many simple expressions is monitored and checked. With an inlined program, the maximum oracle is much smaller—there are fewer expressions. With a maximum oracle strategy, changing to an inlined program structure is somewhat detrimental to performance. With traditional criteria—as the maximum oracle is generally prohibitively expensive—we would recommend inlining code to improve the performance of the output-only oracle strategy.

The above results make sense then as, with observability, we essentially see the same effect. There is no benefit from changing program structure, as the increased complexity of individual statements is replicated in the masking-free path to output required to attain observability. Instead, there is a slight downgrade in performance because the individual statements are more complex. When observability is required, a simpler program structure may be slightly preferable.

Observability reduces sensitivity to the program structure by capturing the complexity benefits of inlining in the path from expression to output.

6 THREATS TO VALIDITY

External Validity: Our study has focused on a small number of systems but, nevertheless, we believe the systems are representative of the critical systems domain, and our results are generalizable to that domain.

We have used one method of test generation—counterexample-based generation. There are many methods of generating tests and these methods may yield different results. Counterexample-based testing is used to produce coverage-directed test cases because it is a method used widely in testing safety-critical systems.

For each model and criteria, we have built 50 reduced test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have produced consistent results [2].

Construct Validity: In our study, we primarily measure fault finding over seeded faults, rather than real faults encountered during development. However, Andrews et al. showed that seeded faults lead to similar conclusions to those obtained using real faults [27] for the purpose of measuring test effectiveness and Just et al. found a positive correlation between mutant detection and fault detection [26]. We have assumed these conclusions hold true in our domain/language, where examples of real faults are rare.

To control experiment costs, we limited the number of mutants used per model to 500. When more than 500 mutants exist, a random selection was used to avoid bias in mutant selection. While the selection of specific mutants is randomized, the distribution is matched to the full distribution of possible mutants in the model. In our experience, mutants sets greater than 100 result in similar fault finding; we generated up to 500 to further increase our confidence that no bias was introduced.

Lau and Yu [57] and Kaminski et al. [58] have defined fault hierarchies for Boolean expressions, outlining cases where detection of one fault could guarantee detection of other, redundant faults. The mutation operators we have employed could produce redundant mutations, but we have not removed those in our experiments. We do not know which faults are actually redundant in our evaluation. However, we have performed a worst-case analysis, and found that there is generally a low correlation between the percent of redundant faults and the percent of detected faults. Therefore, we do not believe that redundancies have had a significant impact on the results of our study.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. Notably, we have avoided statistical inference across case examples.

7 RELATED WORK

In this section, we will discuss our prior work on observability, the role of adequacy criteria in test case generation, other notions of observability, and other topics related to this work.

7.1 Prior Work on Observability

This work is an extension of our prior work defining and exploring the concept of observability [10]–[12]. We first proposed the concept of observability as an extension of the MC/DC coverage criterion [10]. An extended study found that OMC/DC was more effective—and overcame many of the weaknesses of—traditional coverage criteria [11].

In a recent study, we extended the original tagging semantics of MC/DC in order to generate path conditions as part of Dynamic Symbolic Execution [12]. This work used OMC/DC purely as a test generation target rather than a general adequacy measurement approach. A source of optimistic inaccuracy in the original definition of OMC/DC was addressed by requiring value inequality of expressions from two branches when propagating if conditions. This approach was also able to explicitly terminate when there is no feasible paths. In the regular model-based test generation approach used in this and the other past work, a timeout is usually estimated and manually set in order to terminate the generation process. The DSE-based approach, as a result, could complete generation in a more efficient manner.

This work extends previous efforts by decoupling the notion of observability from MC/DC and exploring its application as a generic addition to any coverage criterion. While we found that MC/DC was still the most effective host criterion in many applications, this was not a universal case. This decoupling allows us to explore the impact of choosing a host criterion and to explore the efficacy of observability as a general construct of adequacy criteria. Our experimental work also considers a far wider range of programs than previously explored in order to better understand the general efficacy of observability-based coverage criteria.

7.2 Adequacy Criteria Efficacy in Test Generation

Automated test generation relies on the selection of a measurable test goal. Adequacy criteria, such as the coverage criteria that are the focus of this study are commonly used for this purpose. However, coverage is merely an approximation of a harder to quantify goal—“finding faults”. The need to rely on approximations leads to two questions that researchers have examined multiple times. First, *do such proxies produce effective tests?* If so, *which criteria should be used to generate tests?*

Answers to these two questions are—to date—inconclusive. Some studies have noted positive correlation between coverage level and fault detection [21], [54], [59], while other work paints a negative portrait

of coverage [60]. Our prior work in search-based test generation for Java programs has found that coverage level is more strongly indicative of efficacy than factors such as suite size [21]. However, in our prior studies of model-based generation, tests generated specifically to achieve coverage were often outperformed by randomly-generated tests [11], [47], [48].

Results to date are promising, given the complexity of some of the faults detected [21], [61]–[63]. However, automated generation does not yet produce human competitive results [64]. Ultimately, if automated generation is to have an impact on testing practice, it must produce results that match—or, ideally, outperform—manual testing efforts. The efficacy of suites generated for many coverage criteria is limited by issues such as masking. Choices about how code is written [1], [2] and the selection of test oracle [3], [5], [40] impact the efficacy of some criteria. The notion of observability was designed to address both issues.

7.3 Coverage Criteria in Lustre and Function Block Diagram

Researchers studying coverage criteria for Lustre [65] and Function Block Diagrams (FBD) [66] implicitly investigated observability by examining variable propagation from the inputs to the outputs.

Some of the structural coverage criteria proposed specifically for Lustre are based on *activation conditions* that are defined as the condition upon which a data flow is transferred from the input to the output of a path. When the activation condition of a path is true, any change in input causes modification of the output within a finite number of steps [65]. Coverage metrics for FBD are based on a *d-path condition* that is similar to activation conditions in Lustre [66].

These coverage criteria in Lustre and FBD are different from the notion of observability in several respects. First, these metrics check if specific inputs affect the outputs and measure the coverage of variable propagation on all possible paths. Observability, on the other hand, checks if each test obligation from the host criterion affects the monitored variables, and determines if a path exists which propagates the effect of the obligation. Second, observability requires a stronger notion of how a decision must be exercised.

7.4 Observability in Hardware Testing

Observability has been studied in testing of hardware logic circuits. Observability-based code coverage metric (OCCOM) is a technique where tags are attached to internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (corrupted state) [42], [67]. A variable is tagged when there is a possible change in the value of the variable due to an fault. The observability coverage can be used to determine whether erroneous effects that are activated by the inputs can be observed at the outputs.

The key differences between our notion of observability and OCCOM are twofold: (1) our notion of observability investigates variable value propagation, while OCCOM investigates fault propagation and (2) OCCOM has pessimistic inaccuracy because of tag cancellation. When both positive and negative tags exist in the same assignment (e.g., different tags in an ADDER or the same tags in a COMPARATOR cancel each other out), no tag is assigned [67] or an unknown tag “?” [42] is used. Variables without tags or with unknown tags are not considered to carry an observable error.

Since we do not make a distinction between positive and negative tags, we do not have tag cancellation or the corresponding pessimistic inaccuracy. Extended work in [68] may fix pessimistic inaccuracy by producing test vectors with specific values, but is highly infeasible.

7.5 Mutation Coverage

As discussed in Section 3.2 there are close connections between observability and strong mutation coverage [25]. In the general case, strong mutation coverage is very difficult to achieve and expensive to measure [28], though recent efforts have made it somewhat more efficient [69]. Therefore, weak mutation coverage is often used instead, as a high level of weak mutation coverage can be more easily reached. Observability, as proposed in this work, offers a means to increase strong mutation coverage of faults in Boolean decisions.

7.6 Dynamic Taint Analysis

Our ideas for examining observability via tag propagation were derived from dynamic taint analysis (also called dynamic information flow analysis). In this approach, data is marked and tracked in a program at runtime, similar to our tagging semantics. This technique has been used in security as well as software testing and debugging [70], [71]. Taint propagation occurs in both explicit information flow (i.e., data dependencies) and implicit information flow (control dependencies). Although the way in which markings are combined varies based on the application, the default behavior is to union them [71]. Thus, dynamic taint analysis is conservative and does not consider masking. More accurate techniques for information flow modeling define path conditions quite similar to those used in this paper to prove *non-interference*, that is, the non-observability of a variable or expression on a particular output [72].

7.7 Dynamic Program Slicing

Dynamic program slicing [73] computes a set of statements that influence the variables used at a program point for a particular execution. This can identify all variables that contribute to a specific program point, including output. However, similarly to dynamic taint analysis, it does not consider masking. Checked coverage uses dynamic slicing to assess oracle quality, where

oracles are program assertions [74]. Given a test suite, it yields a percentage of all statements that contribute to the value of any assertion (i.e., are observable at that assertion) vs. the total number of statements covered by the test suite. This work is designed to assess the *oracle*, not the test suite.

7.8 Checked Coverage

Recent work presents a stronger notion of coverage, checked coverage, which counts only statements whose execution contributes to an outcome checked by an oracle [75], [76]. The ideas of checked coverage and observability are conceptually very similar. With observability, one is trying to see whether a condition—or other code structure—propagates to a point of observation. With checked coverage, one restricts measured code coverage to program statements that are found in a backwards dynamic slice. Thus, in observability, metrics markings are *forward propagated*, while in checked coverage, markings are *back propagated*. Checked coverage is computed over program statements rather than conditions, so is not as precise as our work on observability. Also, it assumes *a priori* that one has a test from which to perform a backwards dynamic slice, so it is not suitable in its current form for test case generation.

8 CONCLUSION AND FUTURE WORK

Many test adequacy criteria are highly sensitive to how statements are structured or the choice of test oracle. This sensitivity is caused by the fact that the obligations for structural coverage criteria are only posed over specific syntactic elements—statements, branches, conditions. Such obligations ensure that execution *reaches* the element of interest, and exercises it in the prescribed manner. However, no constraints are imposed on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered. To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that has the potential to eliminate masking. Observable coverage criteria combine the test obligations of their host criterion with an additional path condition that increases the likelihood that a fault encountered when executing the element of interest will propagate to a variable monitored by the test oracle. We hypothesize that this additional observability constraint will improve the effectiveness of the host criterion—no matter which criterion is chosen—particularly when used as a test generation target, paired with a common output-based test oracle strategy.

Our study has revealed that test suites satisfying Observable MC/DC are generally the most effective criterion. Overall, we found that adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%. Some of the factors that can harm

efficacy include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output. The addition of observability results in an increase in the size of test suites and a decrease in the number of fulfilled obligations. The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself. Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity. Finally, our hypothesis has proven accurate—observability reduces sensitivity to the choice of oracle and to the program structure.

Based on our results, *observability* is a valuable extension—regardless of the chosen host criterion. The addition of observability increases test efficacy and produces test suites that are robust to changes in the structure of program or the variables under monitored by test oracle. While our results are encouraging, there are areas open for exploration in future research:

- **Extension to other coverage criteria:** A variety of coverage criteria have been proposed for logical expressions, some potentially more effective than MC/DC [77]. We will explore the effect of extending such criteria to offer observability.
- **Oracle data selection:** We used two types of oracles representing different extremes. Maximum oracles monitor all internal and output variables, and output-only oracles monitor only the output variables. However, we have found that some level of oracle sensitivity could be overcome with intelligently constructed oracles [3]. We intend to further consider whether such oracles could be more effective in situations where observability constraints are too difficult for the test generator.
- **Selection of solver used for test generation:** While conducting our study, we found that the model checker had difficulties with satisfying the observability constraints for some models. Further, we witnessed varying efficacy performance between the underlying solvers powering our employed test generation approach. We will extend our work in the future to quantify and further explore the choice of solver and its effect on suite efficacy.
- **Method of test generation:** In this work, we have used model-based test generation. In past work, we also used Dynamic Symbolic Execution to generate test suites satisfying Observable MC/DC [12]. In the future, we would like to explore other methods of generating tests for observable criteria, such as search-based generation.

REFERENCES

- [1] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl, “The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage,” *ACM Trans. Softw. Eng. Methodol.*, vol. 25, pp. 25:1–25:34, July 2016.

- [2] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 161–170, ACM, 2008.
- [3] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "Automated oracle data selection support," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.
- [4] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, IEEE Press, 2012.
- [5] M. Staats, M. Whalen, and M. Heimdahl, "Better testing through oracle selection (nier track)," in *Proceedings of the 33rd Int'l Conf. on Software Engineering*, pp. 892–895, 2011.
- [6] J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Engineering Journal*, pp. 193–200, September 1994.
- [7] RTCA, DO-178B: *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
- [8] S. Vilkomir and J. Bowen, "Reinforced condition/decision coverage (RC/DC): A new criterion for software testing," *Lecture Notes in Computer Science*, vol. 2272, pp. 291–308, 2002.
- [9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," *Proceedings of the IEEE*, vol. 79, pp. 1305–1320, September 1991.
- [10] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 2013 Int'l Conf. on Software Engineering*, ACM, May 2013.
- [11] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "The risks of coverage-directed test case generation," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, 2015.
- [12] D. You, S. Rayadurgam, M. Whalen, M. P. E. Heimdahl, and G. Gay, "Efficient observability-based test generation by dynamic symbolic execution," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 228–238, Nov 2015.
- [13] E. Kit and S. Finzi, *Software Testing in the Real World: Improving the Process*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995.
- [14] W. Perry, *Effective Methods for Software Testing, Third Edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006.
- [15] M. Pezze and M. Young, *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.
- [16] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward'14*, (New York, NY, USA), pp. 255–268, ACM, 2014.
- [17] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pp. 83–91, IEEE Computer Society, April 2001.
- [18] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105–156, 2004.
- [19] N. Juristo, A. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Empirical Software Engineering*, vol. 9, no. 1, pp. 7–44, 2004.
- [20] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013.
- [21] G. Gay, "The fitness function for the job: Search-based generation of test suites that detect real faults," in *Proceedings of the International Conference on Software Testing, ICST 2017*, IEEE, 2017.
- [22] G. Fraser and A. Arcuri, "Whole test suite generation," *Software Engineering, IEEE Transactions on*, vol. 39, pp. 276–291, Feb 2013.
- [23] RTCA/DO-178C, "Software considerations in airborne systems and equipment certification."
- [24] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.
- [25] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 2 ed., 2016.
- [26] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, (Hong Kong), pp. 654–665, November 18–20, 2014.
- [27] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608 –624, aug. 2006.
- [28] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2014.
- [29] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11*, (Washington, DC, USA), pp. 612–615, IEEE Computer Society, 2011.
- [30] R. A. D. Millo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, pp. 900–910, September 1991.
- [31] B. Marick, "The weak mutation hypothesis," in *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV4*, (New York, NY, USA), pp. 190–199, ACM, 1991.
- [32] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pp. 152–158, Jul 1988.
- [33] "Mathworks Inc. Simulink." <http://www.mathworks.com/products/simulink>, 2015.
- [34] "MathWorks Inc. Stateflow." <http://www.mathworks.com/stateflow>, 2015.
- [35] Esterel-Technologies, "SCADE Suite product description." http://www.estrel-technologies.com/v2/_scadeSuiteForSafety-CriticalSoftwareDevelopment/index.html, 2004.
- [36] G. Hagen, *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
- [37] W. Howden, "Theoretical and empirical studies of program testing," *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 293–298, 1978.
- [38] E. Weyuker, "The oracle assumption of program testing," in *13th International Conference on System Sciences*, pp. 44–49, 1980.
- [39] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, pp. 507–525, May 2015.
- [40] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 International Conference on Software Engineering*, pp. 870–880, 2012.
- [41] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *Journal on Software Tools for Technology Transfer*, vol. 4, February 2003.
- [42] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM-efficient computation of observability-based code coverage metrics for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 1003–1015, 2001.
- [43] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theor. Comput. Sci.*, vol. 103, pp. 235–271, Sept. 1992.
- [44] G. Rosu and T. F. Serbanuta, "An overview of the k semantic framework," *J. of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397 – 434, 2010.
- [45] J.-L. Colaço, B. Pagano, and M. Pouzet, "Scade 6: A formal language for embedded critical software development," in *International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2017.
- [46] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, 1993.
- [47] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Moving the goalposts: Coverage satisfaction is not enough," in *Proceedings of the 7th International Workshop on Search-Based Software Testing, SBST 2014*, (New York, NY, USA), pp. 19–22, ACM, 2014.
- [48] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering (J. de Lara and A. Zisman, eds.), vol. 7212 of Lecture Notes in Computer Science*, pp. 409–424, Springer Berlin Heidelberg, 2012.
- [49] G. Gay, S. Rayadurgam, and M. P. E. Heimdahl, "Automated steering of model-based test oracles to admit real program behaviors," *IEEE Transactions on Software Engineering*, vol. 43, pp. 531–555, June 2017.

- [50] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," pp. 86–104, 2008.
- [51] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *Software Engineering Notes*, vol. 24, pp. 146–162, November 1999.
- [52] A. Gacek, "JKind - a Java implementation of the KIND model checker," <https://github.com/agacek>, 2015.
- [53] M. Heimdahl, G. Devaraj, and R. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?," in *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*, (Tampa, Florida), March 2004.
- [54] A. Namin and J. Andrews, "The influence of size and coverage on test suite effectiveness," 2009.
- [55] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. pp. 80–83, 1945.
- [56] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation, STOV '10*, (New York, NY, USA), pp. 1–4, ACM, 2010.
- [57] M. F. Lau and Y. T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 247–276, July 2005.
- [58] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002 – 2012, 2013.
- [59] A. Mockus, N. Nagappan, and T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pp. 291–301, Oct 2009.
- [60] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, (New York, NY, USA), pp. 435–445, ACM, 2014.
- [61] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, (New York, NY, USA), ACM, 2015.
- [62] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP)*, ICSE 2017, (New York, NY, USA), ACM, 2017.
- [63] H. Almulla, A. Salahirad, and G. Gay, "Using search-based test generation to discover real faults in Guava," in *Proceedings of the Symposium on Search-Based Software Engineering, SSBSE 2017*, Springer Verlag, 2017.
- [64] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA*, (New York, NY, USA), pp. 291–301, ACM, 2013.
- [65] A. Lakehal and I. Parissis, "Structural test coverage criteria for Lustre programs," in *Proceedings of the 10th Int'l workshop on Formal methods for industrial critical systems*, pp. 35–43, 2005.
- [66] E. Jee, J. Yoo, S. Cha, and D. Bae, "A data flow-based structural testing technique for FBD programs," *Information and Software Technology*, vol. 51, no. 7, pp. 1131–1139, 2009.
- [67] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation," in *Proceedings of the 1996 IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 418–425, 1996.
- [68] F. Fallah, P. Ashar, and S. Devadas, "Functional vector generation for sequential HDL models under an observability-based code coverage metric," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 6, pp. 919–923, 2002.
- [69] F. C. M. Souza, M. Papadakis, Y. L. Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pp. 45–54, May 2016.
- [70] W. Masri, A. Podgurski, and D. Leon, "Detecting and debugging insecure information flows," in *Proceedings of the 15th Int'l Symposium on Software Reliability Engineering*, pp. 198–209, 2004.
- [71] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 Int'l Symposium on Software Testing and Analysis*, pp. 196–206, 2007.
- [72] M. W. Whalen, D. A. Greve, and L. G. Wagner, *Model Checking Information Flow*. Springer-Verlag, Berlin Germany, March 2010.
- [73] H. Agrawal and J. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*, vol. 25, pp. 246–256, 1990.
- [74] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Proceedings of the Fourth IEEE Int'l Conf. on Software Testing, Verification and Validation*, pp. 90–99, 2011.
- [75] D. Schuler, *Assessing Test Quality*. PhD thesis, Universitat des Saarlandes, May 2011.
- [76] A. Murugesan, M. W. Whalen, N. Rungta, O. Tkachuk, S. Person, M. P. E. Heimdahl, and D. You, "Are we there yet? determining the adequacy of formalized requirements and test suites," in *NASA Formal Methods* (K. Havelund, G. Holzmann, and R. Joshi, eds.), (Cham), pp. 279–294, Springer International Publishing, 2015.
- [77] Y. T. Yu and M. F. Lau, "A comparison of mc/dc, mumcut and several other coverage criteria for logical decisions," *Journal of Systems and Software*, vol. 79, no. 5, pp. 577 – 590, 2006. Quality Software.



Ying Meng just received her M.S. in Computer Science from University of South Carolina. She will continue pursuing her Ph.D. at University of South Carolina. Her research interests mainly focus on software testing, especially automated test generation.



Gregory Gay is an Assistant Professor of Computer Science & Engineering at the University of South Carolina. His research interests include automated testing and analysis—with an emphasis on structural coverage criteria—and search-based software engineering. Greg received his Ph.D. from the University of Minnesota, working with the Critical Systems research group, and an M.S. from West Virginia University.



Michael Whalen is the Director of the University of Minnesota Software Engineering Center and a consultant for Rockwell Collins, Inc. Dr. Whalen is interested in formal analysis, language translation, testing, and requirements engineering. He has developed simulation, translation, testing, and formal analysis tools for Model-Based Development languages including Simulink, Stateflow, SCADE, and RSML-e, and has published more than 60 papers on these topics. He has led successful formal verification projects on large industrial avionics models, including displays (Rockwell-Collins ADGS-2100 Window Manager), redundancy management and control allocation (AFRL CerTA FCS program) and autoland (AFRL CerTA CPD program). He has recently been researching tools and techniques for scalable compositional analysis, testing and system image generation from system architectural models for quadcopters and autonomous helicopters.

To Call, or Not to Call: Contrasting Direct and Indirect Branch Coverage in Test Generation *

Gregory Gay

University of South Carolina
Columbia, SC, United States

greg@greggay.com

ABSTRACT

While adequacy criteria offer an end-point for testing, they do not mandate *how* targets are covered. Branch Coverage may be attained through *direct* calls to methods, or through *indirect* calls between methods. Automated generation is biased towards the rapid gains offered by indirect coverage. Therefore, even with the same end-goal, humans and automation produce very different tests. Direct coverage may yield tests that are more understandable, and that detect faults missed by traditional approaches. However, the added burden for the generation framework may result in lower coverage and faults that emerge through method interactions may be missed.

To compare the two approaches, we have generated test suites for both, judging efficacy against real faults. We have found that requiring direct coverage results in lower achieved coverage and likelihood of fault detection. However, both forms of Branch Coverage cover code and detect faults that the other does not. By isolating methods, Direct Branch Coverage is less constrained in the choice of input. However, traditional Branch Coverage is able to leverage method interactions to discover faults. Ultimately, both are situationally applicable within the context of a broader testing strategy.

CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Search-based software engineering; Software verification and validation;

KEYWORDS

Adequacy Criteria, Automated Test Generation, Branch Coverage

ACM Reference Format:

Gregory Gay. 2018. To Call, or Not to Call: Contrasting Direct and Indirect Branch Coverage in Test Generation . In *SBST'18: SBST'18:IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3194718.3194719>

*This work is supported by National Science Foundation grant CCF-1657299.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SBST'18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5741-8/18/05...\$15.00
<https://doi.org/10.1145/3194718.3194719>

```
public int[] add(int[] values, int valueToAdd) {
    for(int i = 0; i < values.size(); i++){
        if(valueToAdd >= 0){
            values[i] = faultyAdd(values[i], valueToAdd);
        }
    }
    return values;
}

public int faultyAdd(int value, int valueToAdd) {
    if (valueToAdd <= 0){ // FAULT, should be ==
        return value;
    }
    return value + valueToAdd;
}
```

Figure 1: Sample code where coverage can be attained directly or indirectly over method `faultyAdd`.

1 INTRODUCTION

As we cannot know what faults exist in software, dozens of criteria—ranging from the measurement of structural coverage to the detection of synthetic faults [9, 10]—have been proposed to judge the *adequacy* of software testing efforts. Such *adequacy criteria* provide advice to developers, and can be used as optimization targets for automated test generation [8].

Regardless of the process used to create test cases—automated or manual—adequacy criteria offer a measurable goal, a point at which test creation can stop. Consider Branch Coverage—arguably the most common criterion used in research and practice [5]. At various points in a class, the decision of which block of statements to execute depends on the outcome of a *branch predicate*. Such branching points—contained within `if` and `switch` statements and loop conditions—determine the flow of control. Branch Coverage mandates that each predicate evaluate to all possible outcomes, ensuring that the correct statements are executed.

No conditions are placed on *how coverage is achieved*. As a result, even though they may have the same end-goal, humans and automation produce very *different* test cases. Consider the two methods in Figure 1. Method `add` iterates over an array of integers. If the value to add to each is ≥ 0 , then method `faultyAdd` is called to add that amount. Method `faultyAdd` has a fault in it, where—if the value to add is ≤ 0 —we return the original value. The expression should state $= 0$, which means that negative numbers are incorrectly handled. Because the two methods are linked through the call from `add` to `faultyAdd`, Branch Coverage of `faultyAdd` can be attained *indirectly* by providing test input to `add`. Automated test generation algorithms, designed to reward efficient attainment of coverage, may never call `add` directly as its branches can be covered indirectly.

The use of adequacy criteria in automated generation contrasts how such criteria are used by humans. For a human, a criterion such as Branch Coverage typically serves an advisory role—as a way to point out gaps in existing efforts. Yet, in automated generation,

coverage is typically *the goal*, and generators will single-mindedly climb towards that goal. A human tester would not stop after testing `add`, just because Branch Coverage has been attained. They would still write unit tests for `faultyAdd` to ensure that it works in isolation. By stopping after attaining indirect coverage, automated test suites *cannot* discover the fault in Figure 1—the indirect call can only cover the faulty code with a value of 0, but a negative value is needed to trigger a failure. This is impossible without a direct call. While a human may not discover this fault either, they are more likely to attempt such input. Although this is a simple example, more complex representations of the same situation are common during development (we cover real-world examples in Section 4.3).

Indirect coverage of branches also carries a cognitive cost for human developers. In a user study, Fraser et.al found that developers dislike tests that cover branches indirectly, because they are harder to understand and extend with assertions [3]. This imposes a high *human oracle cost* that may outweigh the benefits of automated generation [1]. The understandability benefits of direct coverage may help alleviate the concerns of developers with the readability of generated unit tests [2].

Recent updates to the EvoSuite test generation framework allow the use of both traditional Branch Coverage, where indirect attainment is allowed, and Direct Branch Coverage, where branches must be covered through direct method calls [10]. Direct Branch Coverage should carry a lower human oracle cost, and may detect faults that require direct calls. However, because indirect coverage does not contribute to the total Branch Coverage, the generator must make additional method calls to cover branches that traditional Branch Coverage could handle indirectly. This will likely result in a dip in coverage unless additional time is offered to the generation process. If there is enough of a coverage loss, then generated suites may have lower fault-detection potential as well. Additionally, while direct coverage is required to detect the example in Figure 1, other faults may only—or more easily—emerge by focusing on the interactions between methods. Therefore, it is not clear whether the benefits of direct coverage outweigh the costs of ignoring indirect coverage.

In order to study the costs and benefits of each approach, we have used EvoSuite to generate test suites using both variants of Branch Coverage, with efficacy judged against the Defects4J fault database [7]. By examining the attained coverage and fault-detection capabilities of both variants, we can determine the impact of the choice of fitness function on the generated test suites and explore situations where one form of Branch Coverage may be more appropriate than the other. To summarize our findings:

- Given a two-minute search budget, traditional Branch Coverage discovers 10.40% more faults and has a 13.59% higher average likelihood of fault detection than Direct Branch Coverage. With a ten-minute budget, traditional Branch Coverage discovers 4.32% more faults and has a 7.61% higher average likelihood of fault detection.
- Similarly, traditional Branch Coverage attains an average 7.94-9.00% higher Line Coverage and 9.09-10.20% higher Branch Coverage over the code, as well as 8.06-9.46% higher coverage over the *faulty* lines of code.
- However, each method covers portions of the code and detects faults that the other does not. By examining methods

in isolation, Direct Branch Coverage is less constrained in the input it uses to cover each method. Traditional Branch Coverage is able to leverage the context in which methods interact to detect faults that emerge from those interactions.

We have found that requiring direct coverage imposes a cost in terms of coverage and likelihood of fault detection. As long as the human oracle benefits of Direct Branch Coverage outweigh the need to offer additional time for generation, practitioners may find value in requiring direct coverage. Ultimately, there are clear situations where each form of coverage is more suited to detecting a particular fault than the other. Importantly, both also have important limitations not possessed by the other. This indicates that both variants have value as part of a broader testing strategy, and that future approaches to test generation could leverage the strengths of each approach.

2 BACKGROUND

Test case creation can naturally be seen as a search problem [8]. Of the thousands of test cases that could be generated for any SUT, we want to select—systematically and at a reasonable cost—those that meet our goals [8]. Given a well-defined testing goal, and a scoring function denoting *closeness to the attainment of that goal*—called a *fitness function*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*). Metaheuristics are often inspired by natural phenomena, such as swarm behavior or evolution.

While the particular details vary between algorithms, the general process employed by a metaheuristic is as follows: (1) One or more solutions are generated, (2), The solutions are scored according to the fitness function, and (3), this score is used to reformulate the solutions for the next round of evolution. This process continues over multiple generations, ultimately returning the best-seen solutions. By determining how solutions change over time, the choice of metaheuristic impacts the quality and efficiency of the search.

As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of tests. The most common methods of measuring adequacy involve coverage of structural elements of the software, such as individual statements, branches of the software’s control flow, and complex boolean conditional statements [9]. Each adequacy criterion embodies a set of lessons about effective testing—requirements tests must fulfill to be considered adequate. If tests execute elements in the manner prescribed by the criterion, than testing is deemed “adequate” with respect to faults that manifest through such structures. Adequacy criteria have seen widespread use in software development, and is routinely measured as part of automated build processes [5]¹.

Adequacy criteria offer clear checklists of testing goals that can be objectively evaluated and automatically measured [9]. These very same qualities make adequacy criteria ideal for use as automated test generation targets. In search-based testing, the fitness function needs to capture the testing objective and guide the search. Through this guidance, the fitness function has a major impact on the quality of the solutions generated. Adequacy criteria can be straightforwardly transformed into distance functions that effectively guide to the search to better solutions [8].

¹For example, see <https://codecov.io/>.

3 STUDY

While coverage criteria mandate an end-goal for testing, they impose no restrictions on how that goal is attained. Most test generation approaches count *indirect* coverage of the code in called methods towards the total. However, we could restrict counted coverage to that attained through direct calls to methods [10]. Direct Branch Coverage may offer benefits in terms of the understandability of test cases, and may contribute to fault discovery. However, it is not clear whether those benefits outweigh the potential loss in coverage—and potentially fault-detection capability—that would result from the additional demands imposed on the generation framework.

In order to study the costs and benefits of both forms of Branch Coverage, we have used EvoSuite to generate test suites using both variants, with efficacy judged against the Defects4J fault database [7]. By examining the coverage and fault-detection capabilities of suites generated using both forms of coverage, we can determine the impact of this choice on the automated test generation process and explore situations where one form of Branch Coverage may be more appropriate than the other. In particular, we wish to address the following research questions:

- (1) Given a fixed time budget, which form of Branch Coverage detects the most faults, and which has the highest likelihood of fault detection?
- (2) Given a fixed time budget, does the additional difficulty of attaining Direct Branch Coverage result in a lower final level of attained coverage?
- (3) How does an increased search budget impact the performance gap between the two forms of Branch Coverage?
- (4) Are there particular types of faults that certain forms of Branch Coverage are better suited to detect?

We have performed the following experiment:

- (1) **Collected Case Examples:** We have used 353 real faults, from five Java projects, as test generation targets (Section 3.1).
- (2) **Generated Test Cases:** For each class, we generated 10 suites satisfying each form of Branch Coverage. We performed this task using a two-minute and a ten-minute search budget per CUT (Section 3.2).
- (3) **Assessed Fault-finding Effectiveness** (Section 3.3).
- (4) **Recorded Generation Statistics:** For each suite, fault, and budget, we measure factors that allow us to compare suites, related to coverage, suite size, and suite fitness (Section 3.3).

3.1 Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [7]². Currently, the “stable” dataset consists of 357 faults from five projects: Chart (26 faults), Closure (133), Lang (65), Math (106), and Time (27). Four faults from the Math project were omitted due to complications encountered during suite generation, leaving 353 that we used in our study.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system.

²Available from <http://defects4j.org>

Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings. For each fault, Defects4J provides access to the faulty and fixed versions of the code and developer-written test cases that expose the fault.

3.2 Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [4, 11]. In this study, we used EvoSuite version 1.0.3 and its implementations of Branch Coverage and Direct Branch Coverage.

A test suite satisfies Branch Coverage if all control-flow branches are taken by at least one test case—the test suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one whose execution evaluates the predicate to `false`. To guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how “close” the targeted predicate is to being true. The fitness value of a test suite is measured by executing all of its tests while tracking the branch distances $d(b, Suite)$ for each branch.

$$F_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite)) \quad (1)$$

Note that $v(\dots)$ is a normalization of the distance $d(b, Suite)$ between 0-1. The value of $d(b, Suite)$, then, is calculated as follows:

$$d(b, Suite) = \begin{cases} 0 & \text{if the branch is covered,} \\ v(d_{min}(b, Suite)) & \text{if the predicate has been ex-} \\ 1 & \text{ecuted at least twice,} \\ & \text{otherwise.} \end{cases} \quad (2)$$

The cost function used to attain the distance value follows a standard formulation based on the branch predicate [8].

The fitness function for Direct Branch Coverage is the same as that used for Branch Coverage [10], but only methods directly invoked by the test cases are considered for the fitness and coverage computation of branches in public methods. Private methods may be covered indirectly (as they cannot be called directly).

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario, where tests guard against future issues.

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits each fitness function, and is comparable to similar testing experiments [11]. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget.

Generation tools may generate flaky (unstable) tests [11]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky and

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	17	16	36	53	16	138
	600	20	19	35	54	17	145
	Total	21	21	41	57	18	158
DBC	120	19	19	36	47	18	139
	600	19	22	40	52	18	151
	Total	19	22	40	52	18	151

Table 1: Number of faults detected by each Branch Coverage variant. Totals are out of 26 faults (Chart), 133 (Closure), 65 (Lang), 102 (Math), 27 (Time), and 353 (Overall).

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	45.00%	4.66%	34.00%	27.94%	34.82%	22.07%
	600	48.46%	5.79%	40.15%	32.75%	39.26%	25.61%
	% Change	7.69%	24.19%	18.10%	17.19%	12.77%	16.05%
DBC	120	34.23%	5.11%	30.00%	24.51%	31.11%	19.43%
	600	40.77%	6.09%	38.77%	28.63%	40.37%	23.80%
	% Change	19.10%	19.12%	29.23%	16.80%	29.76%	22.45%

Table 2: Average likelihood of fault detection (proportion of suites that detect the fault to those generated), broken down by coverage type, budget, and system. “% Change” indicates how results change when moving to a larger search budget.

non-compiling tests. On average, less than one percent of the tests are removed from each suite [4].

3.3 Data Collection

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **effectiveness** of each fitness function, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault. We also collected the following for each test suite:

Achieved Branch and Line Coverage: Using the Cobertura tool, we have measured the Line and Branch Coverage achieved by each suite over each CUT.

Patch Coverage: A high level of coverage does not necessarily indicate that the lines relevant to the fault are covered. We also record Line Coverage over the program statements modified by the patch that fixes the fault—the lines of code that differ between the faulty and fixed version.

Test Suite Size: Suites containing more tests are often thought to be more effective [6]. Even if two suites achieve the same coverage, the larger may be more effective simply because it exercises more combinations of input.

Test Suite Length: Each test consists of one or more method calls. Even if two suites have the same number of tests, one may have *longer* tests—making more method calls. In assessing suite size, we must also consider test length.

4 RESULTS & DISCUSSION

4.1 Comparing Fault Detection Capabilities

In Table 1, we list the number of faults detected by each variant of Branch Coverage (BC is traditional Branch Coverage, DBC denotes Direct Branch Coverage), broken down by system and search budget. Due to the stochastic search, a higher budget does not guarantee detection of the same faults found under a lower search budget. Therefore, we also list the total number of faults detected by each coverage type. In total, traditional Branch Coverage detects 158 faults (44.76% of the examples), while Direct Branch Coverage only

detects 151 (42.78%). From these results, we can see that our initial hypothesis—that Direct Branch Coverage will be more difficult to satisfy due to the requirement for direct coverage—has some truth to it. At the two-minute budget, BC detects 10.40% more faults than DBC. This gap narrows at the ten-minute budget, where BC only detects 4.32% more faults.

One suite generated by EvoSuite may not always detect a fault detected by another suite—even if the same criterion is used. To more clearly understand the effectiveness of each fitness function, we must not track only whether a fault was detected, but the likelihood of detection—the proportion of detecting suites to the total number of suites generated for that fault. The average likelihood of fault detection is listed for each coverage type, by system and budget, in Table 2. We also list the change in likelihood between budgets. We largely observe the same trends as above. Given a fixed budget, traditional Branch Coverage has an overall average likelihood of fault detection of 22.07% given a two-minute search budget and 25.61% given a ten-minute budget. Direct Branch Coverage follows with a 19.92% chance of detection given a two-minute budget and a 22.78% chance given a ten-minute budget. Again, traditional Branch Coverage outperforms direct coverage, with a 13.59% higher overall chance of detection with two minutes and 7.61% given ten minutes.

Given a fixed time budget, traditional Branch Coverage outperforms Direct Branch Coverage, detecting 10.40%/4.32% more faults with a 13.59%/7.61% higher average likelihood of detection (two/ten-minute budget).

We can perform statistical analysis to assess our observations. We formulate hypothesis H and its null hypothesis, H_0 :

- H : Given a fixed budget, suites generated to satisfy traditional Branch Coverage will have a higher likelihood of fault detection than suites generated to satisfy Direct Branch Coverage.
- H_0 : Observations of fault detection likelihood for both criteria are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate H_0 without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [12]. Due to the limited number of faults for the Chart and Time systems, we have analyzed results across the combination of all systems (353 observations per budget, per criterion). We apply the test for each pairing of fitness function and search budget with $\alpha = 0.05$.

The application of this test results in a p-value of 0.13 at the two-minute budget, and 0.27 at the ten-minute budget. Therefore, we fail to reject the null hypothesis in both cases. Although Branch Coverage has a higher average performance:

Traditional Branch Coverage fails to outperform Direct Branch Coverage with statistical significance.

Further, while traditional Branch Coverage is more effective than DBC, the gap between the two narrows as the search budget increases. The differences in the number of faults detected (Table 1) and likelihood of detection (Table 2) both decrease at the ten-minute budget. We can also see this from the “% Change” rows in Table 2.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	55.26%	17.00%	73.00%	64.62%	74.00%	48.00%
	600	70.28%	23.00%	79.00%	67.19%	85.00%	54.00%
	% Change	27.18%	35.29%	8.22%	3.98%	14.87%	12.50%
DBC	120	49.77%	14.00%	64.00%	61.37%	71.00%	44.00%
	600	60.23%	18.00%	71.00%	65.16%	79.00%	49.00%
	% Change	21.02%	28.57%	10.94%	6.12%	11.28%	11.36%

Table 3: Average Branch Coverage attained by generated suites, broken down by coverage type, budget, and system.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	65.19%	27.00%	79.00%	70.77%	83.00%	56.44%
	600	75.37%	34.00%	82.00%	72.39%	89.00%	61.16%
	% Change	15.59%	25.93%	3.80%	2.29%	7.23%	8.93%
DBC	120	56.67%	24.00%	71.00%	68.88%	81.00%	52.29%
	600	65.01%	28.00%	75.00%	71.11%	84.00%	56.11%
	% Change	14.11%	16.67%	5.63%	3.24%	3.70%	7.69%

Table 4: Average Line Coverage attained by generated suites, by coverage type, budget, and system.

Direct Branch Coverage benefits far more from the increase in search budget than traditional Branch Coverage does for several systems—DBC sees an average overall improvement of 22.45%, while Branch Coverage only improves by 16.05%.

These results confirm that the “direct” coverage requirement of Direct Branch Coverage does impose additional burden on the test generation framework. There is a dip in average performance at both budget levels, but not a statistically significant difference in either case. The performance gap is not enough of a deterrent to recommend the use of traditional Branch Coverage in situations where a testing practitioner could derive human oracle benefit from the understandable test cases generated using Direct Branch Coverage.

As the gap between traditional and Direct Branch Coverage narrows at a higher search budget, we recommend its use—while allocating a longer budget—in situations where DBC may yield understandability benefits.

Additionally, although Branch Coverage detects more faults, it does not necessarily detect the *same* faults. Branch Coverage detects ten faults not detected by Direct Branch Coverage—again maintaining a slight edge. However, Direct Branch Coverage is able to uniquely detect three faults that are missed by traditional Branch Coverage. There is some variation in the performance of each technique between systems. For the Chart system, traditional Branch Coverage earns far better results, with 18.86–31.46% higher likelihood of detection. In general, indirect Branch Coverage maintains the edge, albeit with closer margins. However, there are also a few cases where Direct Branch Coverage has a slightly higher chance of fault detection—namely, for the Closure system at both budget levels (5.18–9.66% improvement) and Time at the ten-minute level (a modest 2.82% improvement). This indicates that:

Both techniques, regardless of overall performance, have some level of situational applicability.

4.2 Comparing Suite Characteristics

In Table 3, we list the average level of Branch Coverage attained by the final generated suites for each system, budget, and coverage

	Branch Cov. to Fault Detection	Line Cov. to Fault Detection
	120	600
BC	0.37	0.35
DBC	0.31	0.29

Table 5: Correlation of coverage to likelihood of fault detection.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	62.76%	19.72%	73.03%	63.88%	84.60%	50.31%
	600	70.80%	25.36%	75.22%	65.80%	91.60%	54.04%
	% Change	12.81%	28.60%	3.00%	3.00%	8.27%	7.41%
DBC	120	55.83%	16.28%	63.99%	61.24%	81.52%	45.96%
	600	64.26%	20.63%	68.73%	64.88%	85.70%	50.02%
	% Change	15.10%	26.72%	7.41%	5.94%	5.13%	8.83%

Table 6: Average patch coverage (coverage over the patched lines of code), by variant, budget, and system.

variant. We do the same for Line Coverage in Table 4. Like with fault detection, traditional Branch Coverage has an edge over Direct Branch Coverage given a fixed budget. Overall, traditional Branch Coverage attains an average of 7.94% higher Line Coverage and 9.09% higher Branch Coverage than Direct Branch Coverage given a two-minute budget. With a ten-minute budget, traditional Branch Coverage attains 9.40% higher average Line Coverage and 10.41% higher Branch Coverage. Again, this effect can be explained by the additional work required to gain coverage if indirect calls do not count towards the total.

A gap in the level of coverage does not always predict a gap in terms of fault-detection capabilities. For Closure and Time, the two systems where Direct Branch Coverage outperformed traditional BC, the average attained Line and Branch Coverage are still lower than that attained by traditional BC. To further examine this effect, we have measured—for both metrics and budgets—the correlation between attained Line Coverage and attained Branch Coverage using the Kendall rank correlation. The resulting τ values are listed in Table 5, where we can see that, at most, attained coverage has a moderate-to-low correlation to the likelihood of fault detection for both versions of Branch Coverage. Lower coverage for Direct Branch Coverage does not entirely explain lower fault-detection efficacy.

Not all coverage is relevant to detecting faults. We can also analyze the “patch coverage”—the coverage attained over the lines of code related to the fault [4, 11]. The resulting patch coverage is listed in Table 6 for each coverage type, budget, and system. Overall, this table offers similar results, with traditional BC attaining 9.46% higher average coverage over patched lines with a two-minute budget and 8.06% with a ten-minute budget.

We can also see from Tables 3–6 that—unlike with the likelihood of fault detection—Direct Branch Coverage often benefits less than traditional BC from an increased search budget. In fact, the gap in overall attained coverage actually increases with the larger budget. However, the performance gap in attained *patch coverage* drops slightly (9.46% to 8.06%) as the budget increases. While this is not of the same significance as the improvement in fault-detection from a higher budget, it does suggest that increasing the budget tends to help Direct Branch Coverage cover the fault.

Branch Coverage attains an average 7.94/9.40% higher Line Coverage and 9.09/10.41% higher Branch Coverage than Direct Branch Coverage (two/ten-minute budget), as well an average 9.46/8.06% higher Patch Coverage.

	Budget	Chart	Closure	Lang	Math	Time	Overall
Covered by BC, not DBC	120	13.52%	5.78%	11.39%	4.26%	5.22%	6.87%
	600	14.62%	8.17%	10.61%	3.75%	6.70%	7.66%
	% Change	8.14%	41.35%	-6.85%	-11.97%	28.35%	11.50%
Covered by DBC, not BC	120	5.29%	2.37%	3.21%	2.21%	2.76%	2.71%
	600	4.24%	2.24%	3.68%	2.33%	1.71%	2.63%
	% Change	-19.85%	-5.49%	14.64%	5.43%	-38.04%	-2.95%

Table 7: Average percent of code that one method covers and the other does not—broken down by budget and system.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	42.85	17.86	73.76	30.36	53.77	36.35
	600	52.52	27.36	82.84	32.74	61.99	43.71
	% Change	22.57%	53.19%	12.31%	7.84%	15.29%	20.25%
DBC	120	48.12	17.69	76.13	29.47	59.77	37.31
	600	65.11	27.12	90.89	34.36	70.87	47.10
	% Change	35.31%	53.31%	19.39%	16.59%	18.57%	26.24%

Table 8: Average suite size—in number of tests—broken down by coverage type, budget, and system.

	Budget	Chart	Closure	Lang	Math	Time	Overall
BC	120	326.46	113.58	565.86	127.91	226.59	225.33
	600	505.14	224.35	629.55	146.35	329.26	305.13
	% Change	54.73%	97.53%	11.26%	14.42%	45.31%	35.41%
DBC	120	366.97	119.96	533.94	178.40	291.18	244.36
	600	599.96	233.01	680.33	209.08	385.12	347.13
	% Change	63.49%	94.24%	27.42%	17.20%	31.58%	42.06%

Table 9: Average suite size—in number of method calls—broken down by coverage type, budget, and system.

Stating that traditional Branch Coverage outperforms Direct Coverage in terms of total coverage does not offer the complete picture—the two metrics also cover *different* targets. In Table 7, we list the average percent of the code that is covered by BC and not DBC as well as the average percent of the code that is covered by DBC and not BC. We can see that each metric covers targets that the other does not. An average of 6.87-7.66% of the lines covered by traditional Branch Coverage are not touched by DBC. However, the reverse is also true. On average, 2.71% of the code is covered by Direct Branch Coverage and remains untouched by BC within the two-minute budget. At the ten-minute budget, DBC covers 2.63% of the program that is never covered by BC.

It is clear that the requirement for direct coverage imposes additional burdens on the test generator. Given a fixed budget, Direct Branch Coverage will attain a lower final level of coverage. To a certain extent, this can be alleviated by offering additional time for generation. However, we can also see that the differences in the final results are not due simply to this burden. The requirement for direct coverage changes how the test generation framework creates test cases, directing the search in different directions. Not only is the total coverage different, but the targets covered differ as well. In many cases, traditional Branch Coverage benefits from the context offered by indirect method calls. There are also cases where Direct Branch Coverage benefits from being forced to make direct calls.

Traditional Branch Coverage and Direct Branch Coverage each cover different targets, again suggesting that each technique has situational applicability.

One other factor that can be used to analyze test suites is the size of the resulting suites. Suite size has been a focus in recent work, with Inozemtseva et al. finding that the size has a stronger correlation to efficacy than coverage level [6] and Gay has found the opposite [4]. In Table 8, we measure size in terms of average

number of unit tests per suite. In Table 9, we measure size in terms of the length—the average number of method calls per suite.

Direct Branch Coverage results in suites that are 2.60% larger at the two-minute budget and 7.80% larger at the ten-minute budget. These suites also have somewhat longer tests, in terms of number of calls—8.50% and 13.76% longer at the two and ten-minute budgets. These results reflect the requirement for direct coverage. As we cannot cover methods indirectly, test cases will call methods and may need more test cases to achieve the same results.

Although this is a natural result of requiring direct coverage, it is also a potential source of concern. Each test added to a suite carries a human oracle cost. Direct Branch Coverage should, in theory, carry a lower *total* cost by producing more understandable test cases [10]. However, this benefit may be reduced by also requiring that more test cases be produced in the first place. In most cases, the increase in suite size is relatively modest—and unlikely to outweigh the potential benefits. However, we cannot confirm this at this time. In the future, we would like to more closely examine the human oracle costs and benefits of each approach.

Direct Branch Coverage produces test suites with 2.60-7.80% more tests and 8.50-13.76% more method calls.

4.3 Comparing Situational Applicability

Our results indicate that each version of Branch Coverage was able to detect faults that the other was not and covered code that the other did not. There are clearly differences between the two forms of coverage that are not merely a result of the search budget, but come down to fundamental differences in how each variant is driven to attain coverage. By examining these situations, we can come to understand the situations where each technique excels and each technique falls short.

Fundamentally, freely allowing a generation framework to count indirect coverage in its total—as is the current standard practice—will bias the generator *towards* indirect coverage. Counting indirect coverage will rapidly accelerate the attainment of coverage. Covering a method through indirect calls removes the need to form input for that method directly. This will result in tests that can differ greatly from those created by humans. Coverage is attained through indirect calls, but these indirect calls may constrain the range of input that is used to call a particular method. In such cases, tests generated using traditional Branch Coverage could miss a fault that a human—or tests generated through Direct Branch Coverage—could detect, by not attempting input that would be tried through direct coverage.

This is the same scenario alluded to in Section 1. Although that example was relatively trivial, similar situations exist in the case examples studied. For instance, we can see such a situation in fault 106 for the Closure project³. The faulty version of this class lacks a check for a null object. This method, `canCollapseUnannotatedChildNames()` is called in several other places. It has no formal parameters. Rather, the execution path depends on the current state of class attributes. As a result, the probability of detecting the fault strongly depends on the context that the method is called in. If the method is called indirectly, the object being examined is unlikely to be null,

³<https://github.com/rjust/defects4j/blob/master/framework/projects/Closure/patches/106.src.patch>

as it has been manipulated within another method first. As the code of interest checks for `! = null`, this does mean that indirect coverage will be attained. However, to detect the fault, we actually want the object to be null. In tests that cover the same code, Direct Branch Coverage will also try to make the object not null. However, the need for a direct call *also* means that we see more test cases with a null object. Traditional Branch Coverage calls the method fewer times, with a smaller range of input values.

A similar example can be seen in Math fault 102⁴. The affected method `chiSquare(double[] expected, long[] observed)` is called elsewhere in the class, making indirect coverage possible. Both Branch and Direct Branch Coverage attain full coverage of the patched code in all cases. However, coverage is less important than choosing the right input. Indirect coverage results in a smaller range of input being passed to the affected method, and a lower likelihood of fault detection. Direct Branch Coverage calls the method in a wider variety of configurations.

Indirect coverage can limit the range of input used to cover a method, missing faults detected through direct calls.

Coverage is a prerequisite to fault detection, but it is not enough to ensure that faults are detected [4, 5]. Context matters—*how* a method is covered is more important than *whether* it was covered. As calls come through another method, indirect coverage limits the range of input passed to the affected method. In addition to the potential understandability benefits, requiring direct coverage gives the generator more freedom to choose how each method is covered.

Direct Branch Coverage is still outperformed by traditional Branch Coverage in many situations. This is because Direct Branch Coverage *entirely* ignores the context offered by indirect coverage. Even though the coverage attained through indirect calls is not counted, those calls are still being made. Methods do not exist in a vacuum, even if we pretend they do when measuring coverage. As these methods work together to perform tasks, faults may be caught by examining how the methods *interact* that are missed by looking at each method in isolation.

Interaction context helps us find faults in two common situations. In the first case, indirect coverage of the faulty method *allows* us to detect the fault while direct coverage does not. An example of this can be seen in Chart fault 5⁵. The affected method `addOrUpdate(XYDataItem item)` is part of a series of `addOrUpdate(...)` methods that take in various numeric primitives and, ultimately, cast them into an instance of `XYDataItem` that is passed into the affected method. Due to the difficulty of creating this non-standard data structure, neither BC or DBC cover this method directly. DBC occasionally detects this fault by accident—through an indirect call. However, because traditional Branch Coverage can explore the affected method indirectly, it is able to more reliably cover and trigger the fault.

Another example can be seen in Closure fault 52⁶. The affected method, `isSimpleNumber(String s)`, is supposed to return `true` if `s` has a length > 0 and has '0' as the first character. In the

faulty version, the requirement of '0' is missing from the check. This is a case where both traditional and Direct Branch Coverage should be on similar footing. However, the affected method is indirectly called by method `getSimpleNumber(String s)`. Coverage of `getSimpleNumber` requires that the input string be a simple number already, which also guarantees indirect coverage of `isSimpleNumber`. This situation gives traditional Branch Coverage an advantage. The context offered by tracking indirect coverage allows EvoSuite to evolve tests that simultaneously consider both methods, rather than requiring each to be covered in isolation. As a result, traditional Branch Coverage reliably produces input that triggers the fault (that has a '0' as the first character).

The second type of situation where this context matters are cases where the faulty method calls another method, and indirect coverage of the second—non-faulty—method helps expose the fault in the calling method. We can see an example of this situation in Math fault 35⁷. The faulty version of the constructor for the class `ElitisticListPopulation` assigns values to the attribute `elitismRate`, whereas the fixed version assigns this value through the method `setElitismRate(double elitismRate)`. The setter method performs bounds-checking, preventing the use of illegal rates—those below 0 or larger than 1. Because traditional Branch Coverage is able to consider indirect coverage of `setElitismRate(...)`, it evolves input that leads to fault detection. Direct Branch Coverage is able to cover `setElitismRate(...)` in isolation, but the lack of guidance when calling the faulty constructor prevents EvoSuite from generating the right input.

By ignoring indirect coverage, Direct Branch Coverage misses faults that emerge when covering method interactions.

Our takeaway from these observations is that both methods of Branch Coverage are flawed. The *intent* behind Direct Branch Coverage is reasonable, but ignoring the context offered by indirect coverage hobbles its performance. Methods do not exist in isolation, and the context offered by their interactions can help expose faults. Even if direct coverage offers human understandability benefits, we should not ignore indirect coverage entirely. Similarly, traditional Branch Coverage is driven towards indirect coverage to the point where the generated tests no longer resemble the tests created by human developers—raising the human oracle cost associated with their use, and potentially missing faults by constraining input choices. It is important to remember that a human tester's job is not done after indirect coverage is attained, and that a generation algorithm could benefit from direct examination of a method.

Fundamentally, coverage is not the true goal of testing. It is a means to judge progress, and a means to automate the creation of input, but what we really want are tests that *detect faults*. Fault detection requires not just coverage, but the right context—the input that will expose the fault. Both traditional and Direct Branch Coverage hold pieces of that context.

We believe that the test generation frameworks of the future should consider means of leveraging the benefits of each approach. For example, a new form of Branch Coverage could require direct

⁴<https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/102.src.patch>

⁵<https://github.com/rjust/defects4j/blob/master/framework/projects/Chart/patches/5/src.patch>

⁶<https://github.com/rjust/defects4j/blob/master/framework/projects/Closure/patches/52/src.patch>

⁷<https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/35.src.patch>

calls to consider each branch to be covered, but—rather than ignoring indirect coverage—the generator could use it as a means of weighting the fitness score. This type of approach may lend Direct Branch Coverage the context that it lacks on its own, and a small increase in generation budget may allow it to overcome the performance loss from the direct coverage requirement.

5 RELATED WORK

Advocates of adequacy criteria hypothesize that we should see a correlation between higher attainment of a criterion and the chance of fault detection for a test suite [5]. Researchers have attempted to address whether such a correlation exists for almost as long as such criteria have existed. Inozemtseva et al. provide a good overview of work in this area [6]. Branch Coverage is a common target for test generation [8, 10] and measurement [5].

Shamshiri et al. found that a combination of three state-of-the-art tools could identify 55.7% of the faults in the Defects4J database [11]. Their work identifies several reasons why faults were not detected, including low levels of coverage, heavy use of private methods and variables, and issues with simulation of the execution environment. In their work, they only used traditional Branch Coverage to generate suites. Recent experiments by Gay compare a variety of fitness functions in terms of fault detection efficacy—including both Branch and Direct Branch Coverage [4]. They found that Branch Coverage is the most effective fitness function.

User studies conducted by Fraser et.al found that developers dislike tests that cover branches indirectly, because they are harder to understand and to extend with assertions [3]. Similarly, Almasi et al. and others have found that concern over the readability of generated suites has slowed industrial adoption of automated test generation [2]. Direct Branch Coverage is an attempt to address such issues [10]. Our study is the first to directly compare and contrast Direct and traditional Branch Coverage.

6 THREATS TO VALIDITY

External Validity: Our study has focused on a relatively small number of systems. Nevertheless, we believe that such systems are representative of—at minimum—other small to medium-sized open-source Java systems. We have used a single test generation framework, EvoSuite, as it is the only framework to implement both direct and indirect Branch Coverage. While results may differ between generation frameworks, we believe that the underlying trends would remain the same. Several of the observed differences between direct and indirect Branch Coverage are a natural result of the requirements of each method, not how they were implemented in EvoSuite. To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and coverage variant. It is possible that larger sample sizes may yield different results. However, we believe that the 14,120 test suites used in analysis are sufficient to draw stable conclusions.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known a priori.

7 CONCLUSIONS

While adequacy criteria offer an end-point for testing, they do not mandate *how* targets are covered. Branch Coverage may be attained through *direct* calls to methods, or through *indirect* calls between methods. In order to study the costs and benefits of each approach, we have judged the efficacy of test suites generated using both variants of Branch Coverage against a set of real faults.

We have found that direct coverage imposes a cost in terms of coverage and likelihood of fault detection. However, traditional and Direct Branch Coverage cover portions of the code and detects faults that the other does not. By examining methods in isolation, Direct Branch Coverage is less constrained in the input it uses to cover each method. However, traditional Branch Coverage is able to leverage the context in which methods interact with each other to detect faults that emerge from those interactions. There are clear situations where each form of coverage is more suited to detecting a particular fault than the other. Importantly, both also have important limitations not possessed by the other. This indicates that both variants have value as part of a broader testing strategy, and that future approaches to test generation should leverage the strengths of both.

REFERENCES

- [1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, March 2013.
- [2] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP)*, ICSE 2017, New York, NY, USA, 2017. ACM.
- [3] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, Sept. 2015.
- [4] G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the International Conference on Software Testing, ICST 2017*. IEEE, 2017.
- [5] A. Groce, M. A. Alipour, and R. Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! ’14*, pages 255–268, New York, NY, USA, 2014. ACM.
- [6] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.
- [7] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.
- [8] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.
- [9] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.
- [10] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.
- [11] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.
- [12] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.

Mapping Class Dependencies for Fun and Profit

Allen Kanapala¹, Gregory Gay²

¹University of South Carolina Salkehatchie, Allendale, SC, USA

²University of South Carolina, Columbia, SC, USA

kanapalaa@acm.org, greg@greggay.com

Abstract. Classes depend on other classes to perform certain tasks. By *mapping* these dependencies, we may be able to improve software quality. We have developed a prototype framework for generating optimized groupings of classes coupled to targets of interest. From a pilot study investigating the value of coupling information in test generation, we have seen that coupled classes generally have minimal impact on results. However, we found 23 cases where the inclusion of coupled classes improves test suite efficacy, with an average improvement of 120.26% in the likelihood of fault detection. Seven faults were detected only through the inclusion of coupled classes. These results offer lessons on how coupling information could improve automated test generation.

Keywords: Coupling, Search-Based Software Engineering, Software Testing

1 Introduction

In complex systems, *coupled* classes depend on other classes to perform certain tasks [6]. By *mapping* and grouping these dependencies, we may be able to offer valuable information that can improve software quality.

Automated test generation can be performed to control testing costs. However, a question remains—which classes should be targeted for generation? Often, only the classes that are known to be faulty are targeted. However, a class that is coupled to a faulty class may still exhibit unexpected behavior. By generating tests for coupled classes, we may be able to detect faults that would otherwise be missed.

We have developed a prototype framework to investigate the effect of coupling in test generation. The framework maps the dependencies between Java classes into a directed graph. This graph can then be used to generate small, dense groupings of classes centered around selected targets. To understand whether test generation is more effective when including coupled classes, we have performed a pilot case study. Using 588 real faults from 14 Java projects, we have identified groupings of classes, generated test suites for these groupings and the faulty classes alone, and assessed whether the inclusion of coupled classes improves the likelihood of fault detection.

Overall, there is only an average improvement of 3.79% in the likelihood of fault detection when incorporating coupled classes. However, when these additional classes have *any* impact, there is an average improvement of 120.26% and seven additional faults were detected only through coupling. The inclusion of coupled classes could

yield significant efficacy improvements if we can identify in advance where they would be useful, and improve coverage of dependencies. In addition, our optimization process often yields unnecessarily large groupings.

We hypothesize that the ability to map and optimize groups of coupled classes could benefit many areas of software engineering research—particularly when automating tasks. Our framework has offered promising preliminary results. We will further explore how coupling information could improve automated software engineering.

2 Coupling Mapping Framework

We have developed a framework that maps class dependencies into a directed graph¹. We then use this graph to optimize small, highly-interconnected groupings of classes coupled to designated targets using a simple genetic algorithm. The following basic process is used to generate groupings:

1. This framework first maps dependencies between classes. In this case, we consider dependencies to be either method calls or variable references to another class.
2. A directed graph is created, where each class is a node, and each edge indicates a dependency. Any classes that have no dependencies and are not the target of a dependency will be filtered out from consideration at this stage. If no classes are coupled to a changed class, the changed class will still be added to the target list.
3. We generate a population of 1,000 groupings, formed by randomly selecting classes.
4. Each grouping is scored using the fitness function described below, and a new population is formed through retention of best solutions (by default, 10%), mutation (20%), crossover (20%), and further random generation (50%).
5. Evolution continues until the time budget is exhausted—by default, five minutes².
6. The best grouping is returned. The changed classes are added to that grouping.

The fitness function used to score groupings is:

$$F_G = \sqrt{\overline{size}^2 + (\overline{coverage} - 1)^2 + \overline{avg(distance)}^2} \quad (1)$$

That is, we prioritize groupings that are closer to a *sweet spot* of fewer classes (*size*), where the chosen classes are coupled to a large number of other classes (*coverage*), and where more classes are either coupled directly to the changed classes or through a small number of indirect dependency links (average *distance*). This should result in a relatively small grouping of classes that are densely coupled to each other and other classes. \bar{x} is a normalized value $0 \leq \frac{x - \min(x)}{\max(x) - \min(x)} \leq 1$. Scores range from $0 \leq F_G \leq \sqrt{3}$ and lower scores are better.

3 Case Study

Traditionally, in unit test generation research, tests are generated solely for the classes we know to contain faults. However, other classes may depend on the faulty classes, and

¹ Available from <https://github.com/Greg4cr/Coupling-Mapping>

² Experimentation suggested that convergence was often reached before that time.

by targeting these coupled classes, we may be more likely to detect faults. We wish to examine whether we could use knowledge of class dependencies to enhance test generation. Specifically, we wish to address: (1) *Can the inclusion of coupled classes improve the efficacy and reliability of test suite generation?* (2) *Are the groupings produced by our framework small enough to be of practical use?*

We have performed the following experiment: (1) We have gathered 588 real faults, from 14 Java projects. (2) For each fault we generate 10 groupings of coupled classes. (3) For each fault, we generate 10 suites per grouping (and for the set of faulty classes) using the non-faulty version of each class. We allow a two-minute generation budget per targeted class. (4) For each fault, we measure the proportion of test suites that detect the fault to the total number generated.

Defects4J is an extensible database of real faults extracted from Java projects [4]³. Currently, it consists of 597 faults from 15 projects. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault. The Guava project was omitted from this study, as its code uses features not supported by our framework. We have used the remaining 588 faults for this study.

EvoSuite applies a genetic algorithm in order to evolve test suites over several generations, forming a new population by retaining, mutating, and combining the strongest solutions [7]. It is actively maintained and has been successfully applied to the Defects4J dataset [2]. In this study, we used EvoSuite version 1.0.5.

Tests are generated from the fixed version of each class and applied to the faulty version in order to eliminate the oracle problem. Tests are generated targeting Branch Coverage, and EvoSuite is allowed two minutes per class—a time chosen to fit within the strict time constraints of the continuous integration (CI) process that testing is commonly performed as part of. In the CI process, changed code is built, verified, and deployed. As this process may be performed multiple times per day, test generation and execution must take place on a limited time scale. As results may vary, we generate 10 groupings of classes per fault, and we perform 10 test generation trials for each fault, grouping, and budget. Generation tools may generate flaky (unstable) tests [2]. We automatically remove non-compiling test cases. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, less than 1% of tests are removed from each suite.

4 Results & Discussion

In Table 1, we compare the average likelihood of detection between the normal case—where only the faulty classes are targeted—and when we generate for a set of targets including coupled classes. From this table, we can see that there is often *some* improvement, but the overall effect is minimal. The inclusion of coupled classes fails to improve results for six systems. For the others, we see average improvements of up to 13.36%. Overall, the average improvement from including coupled classes is only 3.79%.

To understand when coupled classes can benefit generation, we can filter out situations where their inclusion does not improve results. Table 2 lists the average likelihood

³ Available from <http://defects4j.org>

Project	Detection Likelihood (Changed-Only)	Detection Likelihood (With Coupled)
Chart	40.00%	42.58%
Closure	4.10%	5.10%
CommonsCodec	31.36%	35.55%
CommonsCSV	55.00%	58.50%
Jsoup	19.80%	21.70%
Lang	35.20%	35.50%
Math	28.68%	29.29%
Time	34.40%	35.90%
Overall	22.69%	23.55%

Table 1. Average likelihood of detection when only changed classes are targeted and when coupled classes are included, omitting systems with no observed differences.

of detection for the 23 faults where the inclusion of coupled classes had an impact. These filtered results show that when additional classes have *any* impact, it is a major one. In such cases, the likelihood of detection improves by an average of 120.26%. In fact, seven new faults were *only* detected by including coupled classes.

While the addition of classes can be very powerful, it is also very expensive given that—by default—the same amount of time is devoted to generating test cases for each class. Our results illustrate that we should not generate tests for such classes if there is a low likelihood they will help detect faults. **To decide when to add additional targets, we must understand when their inclusion will be helpful.**

Figure 1 depicts a selection of classes in the CommonsCodec project. Three faults—centering around the Base64 class (faults 12, 15, and 20⁴)—see improved efficacy from the inclusion of coupled classes Base64InputStream and Base64OutputStream. Tests generated solely to target Base64 are able to detect all three faults, but not reliably. The incorporation of these two coupled classes greatly increases the likelihood of detection. Class—BCodec—is also coupled to Base64, but does not contribute to efficacy.

These three examples are interesting because the two additional classes are not just coupled through in-code dependencies, but all three are linked by a common conceptual purpose—encoding binary data by treating it numerically and translating it into a base 64 representation. One option for incorporating coupled classes would be to periodically present human developers with coupling information and ask them to filter groupings. Each time that any class in that grouping is altered, those coupled classes could be included in generation.

Of course, not all situations where coupling assists are as straightforward as the CommonsCodec example. For instance, consider fault 31 for the Math system⁵. Generated tests never detect the issue when targeting faulty class ContinuedFraction.

Project	Detection Likelihood (Changed-Only)	Detection Likelihood (With Coupled)
Chart	15.00%	27.50%
Closure	30.00%	62.50%
CommonsCodec	13.33%	44.00%
CommonsCSV	30.00%	51.00%
Jsoup	15.00%	27.80%
Lang	10.00%	30.00%
Math	6.67%	28.33%
Time	25.00%	44.00%
Overall	18.26%	40.22%

Table 2. Average likelihood of detection—omitting cases where coupled classes have no effect.

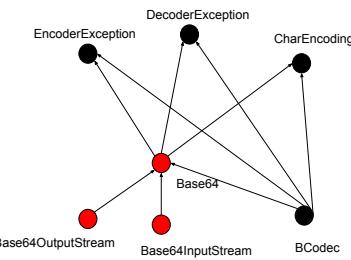


Fig. 1. Partial visualization of coupling. Relevant classes are colored red.

⁴ [https://github.com/Greg4cr/defects4j/blob/master/framework/projects/CommonsCodec/patches/\[12/15/20\].src.patch](https://github.com/Greg4cr/defects4j/blob/master/framework/projects/CommonsCodec/patches/[12/15/20].src.patch)

⁵ <https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/31.src.patch>

Instead, the fault is only detected when tests are generated for Gamma (coupled to Continued Fraction) and GammaDistribution (coupled to Gamma). The reason the coupled classes are useful is likely because they provide guidance to the generator in how to make use of ContinuedFraction. Exposing the fault requires setting up a series of values and calling ContinuedFraction.evaluate(...) on those values. By attaining coverage of Gamma, EvoSuite is able to set up and execute the functionality of ContinuedFraction. Without that guidance, it struggles.

While only a small number of classes are coupled to Continued Fraction, there is not a common conceptual connection like with the CommonsCodec example above. In retrospect, we can explain these situations. However, more research is needed to recognize patterns in when the inclusion of coupled classes is beneficial. Further, asking developers to name useful couplings creates additional maintenance effort, and may not offer sufficient benefit for the time and knowledge required. Therefore, we also need further research into automated means to suggest and prune couplings.

We should also endeavor to make the inclusion of coupled classes more useful by focusing on increased coverage of those dependencies. **The efficacy of generation—when coupled classes are included as targets—may be improved if coverage is ensured of references to changed classes.**

Test generation for each class is an entirely independent process. While the attained Branch Coverage may be relatively high for each targeted class, we have no guarantee that dependencies *between classes* are covered. Steps could be taken to improve coverage of such dependencies by considering coverage of dependencies between classes. Jin and Offutt have proposed coverage criteria for integration testing that could be used to ensure class dependencies are covered [3]. These forms of “Coupling Coverage” could be used to prioritize suites that attain a higher coverage of the specific code segments that require data or functionality from a changed class.

Recent work has found that combinations of coverage criteria can be more effective than individual criteria [2]. For example, combining Branch and Exception Coverage yields test suites that both cover the code and force the program into unusual configurations. “Coupling Coverage” metrics could be thought of as another situationally-appropriate orthogonal criterion. Rather than generating tests using Branch Coverage alone, the generator could combine Branch and “Coupling Coverage” when targeting coupled classes—potentially creating suites that are especially effective at exploiting dependencies between classes, and in turn, at detecting faults.

Regardless of the use, our framework is intended to produce small, effective groups of coupled classes. The size of that group must be small enough to be of practical use. In Table 3, we list the average grouping size for each system. In many cases, we can see that these groupings are larger than would be practical. We used them for this case

	Number of Classes
Chart	18.63
Closure	73.92
CommonsCLI	1.91
CommonsCodec	3.51
CommonsCSV	3.40
CommonsJXPath	20.80
JacksonCore	5.03
JacksonDatabind	34.96
JacksonXML	2.80
Jsoup	26.34
Lang	7.12
Math	12.64
Mockito	34.43
Time	52.87

Table 3. Average number of classes in the groupings.

study, as they are useful for understanding the benefits of such information. **However, we must refine the optimization process to further limit grouping size.**

We found that, in situations where coupling affects the results, only a small number of classes are useful, and they are closely linked to the target classes. Therefore, these groupings could be easily pruned down to a more appropriate size. We will reformulate our fitness function to further constrain grouping size.

5 Related Work

Coupling between classes is a well-established area of research [6]. Similar search-based techniques have been used to suggest refactorings. The CCDA algorithm uses a graph structure and a genetic algorithm to restructure packages based on class dependencies [5]. However, we are aware of no other use of such techniques to optimize groupings for test generation. Past work on integration testing has suggested ways to better ensure that class dependencies are tested [3, 1], but has largely not addressed the question of *which* classes to test. In addition, we are not focused purely on integration testing, but a broader set of scenarios.

6 Conclusions

We have developed a framework to optimize groupings of classes. The results of a pilot study on the applicability of coupling to test generation show potential benefits from generating tests for coupled classes and offer new research challenges.

References

1. Alexander, R.T., Offutt, A.J.: Criteria for testing polymorphic relationships. In: Proceedings 11th International Symposium on Software Reliability Engineering. pp. 15–23 (2000)
2. Gay, G.: Generating effective test suites by combining coverage criteria. In: Proceedings of the Symposium on Search-Based Software Engineering. SSBSE 2017, Springer Verlag (2017)
3. Jin, Z., Offutt, A.J.: Couplingbased criteria for integration testing. *Software Testing, Verification and Reliability* 8(3), 133–154, <https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1689%281998090%298%3A3%3C133%3A%3AAID-STVR162%3E3.0.CO%3B2-M>
4. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), <http://doi.acm.org/10.1145/2610384.2628055>
5. Pan, W., Jiang, B., Xu, Y.: Refactoring packages of objectoriented software using genetic algorithm based community detection technique. *International Journal of Computer Applications in Technology* 48(3), 185–194 (2013), <https://www.inderscienceonline.com/doi/abs/10.1504/IJCAT.2013.056914>
6. Poshyvanyk, D., Marcus, A.: The conceptual coupling metrics for object-oriented systems. In: 22nd IEEE International Conference on Software Maintenance. pp. 469–478 (Sept 2006)
7. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) *Search-Based Software Engineering, Lecture Notes in Computer Science*, vol. 9275, pp. 93–108. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-22183-0_7

Multifaceted Test Suite Generation Using Primary and Supporting Fitness Functions *

Gregory Gay

University of South Carolina
Columbia, SC, United States

greg@greggay.com

ABSTRACT

Dozens of criteria have been proposed to judge testing adequacy. Such criteria are important, as they guide automated generation efforts. Yet, the current use of such criteria in automated generation contrasts how such criteria are used by humans. For a human, coverage is part of a *multifaceted* combination of testing strategies. In automated generation, coverage is typically *the goal*, and a single fitness function is applied at one time. We propose that the key to improving the fault detection efficacy of search-based test generation approaches lies in a targeted, multifaceted approach pairing *primary* fitness functions that effectively explore the structure of the class under test with lightweight *supporting* fitness functions that target particular scenarios likely to trigger an observable failure.

This report summarizes our findings to date, details the hypothesis of primary and supporting fitness functions, and identifies outstanding research challenges related to multifaceted test suite generation. We hope to inspire new advances in search-based test generation that could benefit our software-powered society.

CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Search-based software engineering; Software verification and validation;

KEYWORDS

Automated Test Generation, Search-Based Test Generation, Adequacy Criteria

ACM Reference Format:

Gregory Gay. 2018. Multifaceted Test Suite Generation Using Primary and Supporting Fitness Functions . In *SBST'18: SBST'18:IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3194718.3194723>

*This work is supported by National Science Foundation grant CCF-1657299.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SBST'18, May 28–29, 2018, Gothenburg, Sweden

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5741-8/18/05...\$15.00

<https://doi.org/10.1145/3194718.3194723>

1 INTRODUCTION

With exponential growth in the cost of software testing, we must find means of reducing costs while maintaining quality. Automation of tasks such as unit test creation has a critical role to play in reducing testing costs [2]. Yet, despite advances in automated test generation technology, the efficacy of the produced test suites at detecting faults has yet to match human-produced tests [6, 8, 10].

As we cannot know what faults exist a priori, dozens of criteria—ranging from the measurement of structural coverage to the detection of synthetic faults [9]—have been proposed to judge testing *adequacy*. In theory, if the goals set forth by such criteria are fulfilled, tests should be *adequate* at detecting faults related to the focus of that criterion. Adequacy criteria are important for search-based generation, as they are the most common basis for the fitness functions that judge solutions and guide the search.

Yet, the current use of adequacy criteria in automated generation sharply contrasts how such criteria are used by humans. For a human, coverage typically serves an advisory role—as a way to point out gaps in existing efforts. Human testers build suites in which adequacy criteria contribute to a *multifaceted* combination of testing strategies. Yet, in automated generation, coverage is typically *the goal*, and a single fitness function is applied at one time. Yet, search-based techniques need not be restricted to one criterion—one fitness function—at a time. The test obligations of multiple criteria can be combined into a single score or simultaneously satisfied by multi-objective optimization algorithms. Such *multifaceted* suites have the potential to be more effective than those generated using a single fitness function, as they trade a laser-focus on that one criterion for reasonably high coverage of a varied set of goals [9].

In previous work, we have explored the efficacy of both individual fitness functions [6] and combinations of functions [7] at detecting real faults, given a fixed search budget. Our observations have revealed that, while certain fitness functions are more effective than others, almost all functions are *situationally* adept at detecting certain types of faults. Further, we have found that both high coverage of class structure *and* high satisfaction of the goals of the chosen fitness function are both needed to detect faults. Combining situationally-adept functions like Exception or Output Coverage—particularly functions that lack their own means to increase structural coverage—with a strong structure-focused function such as Branch Coverage—which is unable on its own to favor targeted fault types—yields significant improvements in the likelihood of fault detection. Well-chosen fitness functions, when used in combination, are able to guide the test generation framework towards test suites that combine the strengths of each function, overcome their weaknesses, and produce a testing strategy that is more effective than any single function—even without an increase in search budget.

Therefore, we propose that the key to improving the fault detection efficacy of search-based test generation approaches lies in a *human-like* approach to test creation—the application of a targeted, multifaceted approach to generation where multiple testing strategies are selected and simultaneously explored. We hypothesize that effective test generation strategies will pair **primary** fitness functions that effectively exploit the structure of the class under test with lightweight **supporting** fitness functions that target particular aspects of the class under test likely to trigger an observable failure.

We propose that the hypothesis of effective multifaceted generation based on primary and supporting fitness functions should be explored by the search-based software testing community, and that numerous research challenges connected to this hypothesis remain to be solved. Advances are needed in terms of how criteria are optimized, the identification of new supporting fitness functions tied towards particular types of software faults, selection of a multifaceted function portfolio for new classes and systems, and approaches that reduce the difficulty of generation under a limited budget.

This report summarizes our findings to date, details the hypothesis of primary and supporting fitness functions, and identifies outstanding research challenges related to the topic of multifaceted test suite generation. We hope to inspire new advances in search-based test generation that have the potential to impact industrial practices and benefit our software-powered society.

2 PRELIMINARY RESULTS

We have previously performed empirical studies on the ability of individual criteria to produce test suites that detect real faults [6]¹. After assessing such suites on 593 faults from 15 open-source Java projects, we have found that:

- Branch Coverage² detects more faults and demonstrates a higher likelihood of detection than other functions, given a fixed search budget.
- Regardless of overall performance—most functions have situational applicability, where suites detect faults no other function can detect. Exception³, Output⁴, and Weak Mutation Coverage⁵ show situational applicability, even if their average efficacy is lower than Branch Coverage.
- Factors that indicate a high level of efficacy include high structural coverage over the code and high coverage of the chosen function’s test obligations. In situations where achieved structural coverage is low, the fault does not tend to be found.
- The factor that differentiates occasionally detection and *consistent* detection of a fault is satisfaction of the chosen function’s test obligations. The best suites are ones that *both* explore the code and fulfill their own testing goals, which may be—in cases such as Exception Coverage—orthogonal to structural coverage.

We have also performed exploratory studies of the fault-detection capabilities of combinations of criteria [7]. We have observed:

¹Due to space constraints, we do not fully define each fitness function here. Full definitions can be found in: [6]

²Branch Coverage requires that each control-altering decision outcome be exercised.

³Exception Coverage rewards suites that cause more exceptions to be thrown.

⁴Output Coverage rewards coverage of type-specific abstract output values

⁵Weak Mutation Coverage rewards coverage of synthetic faults.

- A combination of all eight studied functions performs well, but the difficulty of simultaneously satisfying all functions prevents it from outperforming every individual function under a fixed search budget. However, for all systems, at least one targeted combination is more effective than every individual function.
- The most effective combinations vary by system, but all pair a structure-focused function—such as Branch Coverage—with supplemental strategies targeted at the class under test.
 - Across the board, effective combinations include Exception Coverage. Method Coverage⁶ also generally offers an efficacy boost. Both can be added to a combination with minimal effect on generation complexity.
 - Additional targeted criteria—such as Output Coverage for code that manipulates numeric values or Weak Mutation Coverage for code with complex logical expressions—offer further efficacy improvements.

3 PRIMARY AND SUPPORTING FITNESS FUNCTIONS

Fitness functions represent strategies that can be used to manipulate the search. In single-objective generation, the chosen fitness function will determine the focus, strengths, and weaknesses of the resulting suite. If multiple fitness functions are simultaneously applied—whether combined into a single score or simultaneously optimized by a multi-objective algorithm—the resulting test suite will be the product of the interaction of the chosen strategies. In theory, the simultaneous use of a *portfolio* of fitness functions during generation could produce test suites that are able to detect a variety of faults—trading precise focus on one fitness function for reasonable coverage of multiple functions [9].

In practice, selecting this portfolio of fitness functions requires careful consideration. Given a limited fixed time limit for generation, the framework will more easily achieve high coverage of a single function than high coverage of multiple functions, as the combination will require that more goals be met—and, at times, that conflicting goals be met. This explains why the eight-way combination of functions used by the EvoSuite framework is often outperformed by individual functions [6, 7]. The difficulty of optimizing for so many functions in the same time window allotted to one function led to weaker results. Given a sufficiently-long generation period, that eight-function combination may produce stronger test suites. However, our observations also indicate that careful selection of a fitness function portfolio can yield superior results *without increasing the search budget*. We hypothesize that effective automated generation may be performed through targeted selection of this portfolio.

In general, faults cannot be detected without executing the affected lines of code. This is why structure-based criteria such as Branch Coverage and Line Coverage dominated our ranking of individual fitness functions. Yet, past research also indicates that *coverage alone is not enough* [8]. Merely executing a line of code in *any* manner is not sufficient to detect a fault. *How* that line of code is executed matters. In our experiments, a consistently high likelihood of fault detection requires both coverage of the code structure *and* coverage of a fitness function’s goals. In the case of functions

⁶Method Coverage requires that each method be called by test cases.

Fitness Function Portfolio	
Primary Fitness Function(s) Structure-focused, may be more complex to calculate. Use only a small number (1-2) at one time. Example Options: <ul style="list-style-type: none"> • Branch Coverage • Line/Statement Coverage • Block Coverage • Modified Condition/Decision Coverage • Def-Use Coverage 	Supporting Fitness Functions Scenario-focused, typically simple to calculate. Number to use based on complexity of overall portfolio. Example Options: <ul style="list-style-type: none"> • Weak/Strong Mutation Coverage • Exception Coverage • Output Coverage • Method Coverage • Readability • Input Diversity

Figure 1: Example primary and secondary fitness functions.

such as Branch or Line Coverage, these two are the same. However, targeted fitness functions such as Exception or Output Coverage lack an in-built means to drive structural coverage. This may limit the efficacy of such functions as the sole target of test generation, but illustrates a potential advantage of multifaceted generation. A structure-focused function could be used to explore execution paths, while targeted fitness functions could be simultaneously used to *shape* that exploration process—tuning the resulting test suite.

Exception Coverage is effective because it rewards suites that trigger more exceptions—which often are the observable manifestation of a fault. However, it lacks any feedback mechanism to drive generation towards exceptions. Branch Coverage is effective at exploring the structure of a class, but lacks the context needed to drive execution to an observable failure. By uniting the two, Branch Coverage provides the means to explore the class under test—leading to the discovery of additional exceptions. Exception Coverage provides a weighting mechanism to Branch Coverage—increasing the odds of detecting a fault. We hypothesize then, that the key to effective testing is the identification of the correct portfolio of **primary and supporting fitness functions** for the class or system under test.

Primary fitness functions are designed to explore the structure of the class under test. Common examples of such criteria include Branch, Line, or MC/DC Coverage. As structural coverage is a prerequisite to fault detection, primary functions should be the focus of the generation portfolio—perhaps even weighted more heavily than other fitness functions. As such criteria tend to be more complex to compute, requiring more execution time to calculate than other criteria, a limited number of primary functions should appear in the portfolio—typically only one. However, as we have observed scenarios where Branch and Line Coverage were more effective combined than in isolation, multiple primary functions may be considered.

Supporting fitness functions are intended to control *how* code is executed, and should be targeted towards scenarios or faults of interest. For example, Exception Coverage rewards suites that trigger more exceptions [9]. Output Coverage rewards suites that produce particular defined value types for output that belongs to particular data types [1]. Mutation Coverage rewards suites that detect synthetic faults [5]. Such fitness functions—on their own—do not necessarily have the means to drive structural coverage. However, when paired with primary criteria, they can manipulate the overall search strategy, increasing the likelihood of detecting certain types of faults. As such fitness functions must rely on complex primary functions for code exploration, supporting criteria should be *lightweight*. They should not unduly increase the time required to calculate fitness. The number to be used should be based on the complexity of the overall portfolio, and should remain relatively low.

We hypothesize that a well-chosen portfolio of primary and supporting functions will be able to shape test generation in a more human-like manner—resulting in test suites tuned towards particular types of systems, faults, or testing scenarios. However, a number of research challenges related to the selection and optimization of primary and supporting fitness functions remain unsolved.

4 RESEARCH CHALLENGES

4.1 How to Optimize The Chosen Functions

The portfolio of fitness functions can be optimized in multiple ways. Some techniques, such as EvoSuite, combine multiple fitness functions into a single score [9]. Individual fitness functions can be weighted, but an improvement in *any* of the chosen functions is considered to be an improvement in the overall score. A loss of score for one function is acceptable if balanced by enough improvement in another. Other techniques, such as jMetal [4], attempt to simultaneously optimize separate fitness functions. Such approaches attempt to find an optimal balance between each distinct fitness functions.

A portfolio of primary and supporting functions can be explored using either approach. However, each will produce distinct test suites. Merging each function into a single overall score could result in a suite that heavily favors a particular function. However, this may actually be desirable—for example, we may prefer a suite that attains higher structural coverage and lower coverage of a supporting function over a suite that attains perfect balance of the primary and supporting criteria. Different means of optimizing the portfolio should be explored to better understand how to generate suites.

4.2 Identification of the Criteria Portfolio

The identification of the correct portfolio of fitness functions for a system is not a trivial task. In our experience, the most effective portfolio varies from system to system, and depends on the purpose of the system and the type of mistakes that developers tend to make on that project [7].

Because they add little difficulty to generation, we have observed that Exception and Method Coverage have the most consistent effect on suite efficacy. However, targeted supporting criteria often had beneficial effects as well. For example, generation for the *Apache Commons Math* project typically benefited from the inclusion of Output Coverage. This project offers a variety of tools for numeric analysis, and Output Coverage's focus on different abstract types of numeric values naturally led to an increased rate of fault detection.

One way to choose the criteria portfolio for a new class could be through the use of reinforcement learning (RL) [11] as part of the generation process. Each round of generation, the RL algorithm could choose a new portfolio. After each choice, the algorithm would receive a reward chosen from a probability distribution dependent on the portfolio selected. Over time, it would attempt to maximize the total expected reward, identifying the portfolio most adept at improving a chosen “reward function.” Then, the chosen combination could be used from the start of generation—without reinforcement learning—when testing that class in the future.

If developers seek to maximize coverage of a particular adequacy criterion—for instance, developers of avionics applications must satisfy MC/DC Coverage to attain safety certification [8]—then coverage of that criterion could serve as the reward function. The

RL algorithm would suggest a portfolio adept at quickly attaining MC/DC. If a particular criterion is chosen as the reward function, then the RL algorithm could suggest a portfolio that both meets a chosen testing goal and is still able to detect a variety of faults. This process, in particular, could be quite useful for maximizing criteria that are too complex to serve as ideal fitness functions. For example, Strong Mutation Coverage—which requires input that reveals the presence of synthetic faults at the output level—is difficult to optimize as a direct fitness function as it offers little feedback to the search. However, the RL process could suggest a portfolio adept at achieving high levels of Strong Mutation Coverage—made up of individual fitness functions that do offer feedback to the search.

4.3 Discovery of New Supporting Functions

If the structural coverage enabled by the use of a primary fitness function enhances the efficacy of supporting functions, then new fitness functions can be formulated and experimented with without the need for an in-built coverage mechanism. This opens the opportunity to craft new fitness functions around particular testing scenarios, fault types, or other measures that could enhance the fault-detection capabilities of the generated test suites.

For example, when generating test suites, a set of classes must be chosen as targets. Often, generation is performed solely on classes whose code was directly changed. However, the likelihood of fault detection could be increased by also targeting classes that are *coupled*—dependent—on those changed classes. This is particularly true during regression or integration testing. A supporting fitness function could reward test suites that cover connections between the current generation target and particular classes of interest. By relying on structural coverage from a primary criterion, this function could be efficiently calculated as a count of dependencies covered. This would do little to increase the difficulty or cost of generation.

Similarly, many approaches to test generation reward input diversity [3]. Such approaches ensure a wide spread of input choices over the possible option space, theorizing that ensuring diversity will improve the likelihood of fault detection. As a supporting fitness function, a simple diversity measurement could weight the selection of input chosen by the primary fitness function. This would help ensure that a variety of input choices are used while still ensuring a high level of code coverage. This simple metric could be added to a portfolio of other supporting functions as a low-cost means to further increase the likelihood of detection.

4.4 Improving Generation Efficiency

Fundamentally, it is more difficult to generate a suite that optimizes multiple fitness functions than a suite that optimizes a single function. At the least, the task presented to the generation framework is more difficult. Improvements in the efficiency of generation are needed. Such improvements will benefit generation for both single functions and multifaceted portfolios. In that case, more time could be allocated to multifaceted generation without unduly delaying the testing process. In addition, improvements in the efficiency of calculating fitness for individual criteria will also benefit the ability to generate for multiple criteria.

Another potential avenue for improvement could be similar in form to the *archiving* of test obligations performed by EvoSuite.

When obligations are fully satisfied for criteria such as Branch Coverage in EvoSuite, they can be removed from the overall fitness evaluation. This enables improvements in efficiency, as fitness becomes progressively faster to calculate. A similar process could be used to stagger the inclusion of additional criteria. A small “core” portfolio could be considered at the beginning of the generation process. As obligations are covered, additional criteria could be incorporated. Over time, this process would reshape the population of test suites to steadily cover additional facets, and could reduce the complexity of fitness calculation at any one step in that process.

5 CONCLUSIONS

We propose that the key to improving the fault detection efficacy of search-based test generation approaches lies in a *human-like* approach to test creation—the application of a targeted, multifaceted approach to generation where multiple testing strategies are selected and simultaneously explored. We hypothesize that effective test generation strategies will pair strong *primary* fitness functions—criteria that effectively exploit the structure of the class under test—with lightweight *supporting* fitness functions that target particular aspects of the system under test likely to trigger an observable failure.

We hope to inspire new advances in multifaceted test generation that could impact industrial practices and benefit our software-powered society. In particular, advances are needed in how we select a portfolio of fitness functions, new fitness functions are needed that target a variety of testing scenarios, and improvements must be discovered in regards to generation difficulty and time requirements.

REFERENCES

- [1] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, New York, NY, USA, 2014. ACM.
- [2] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.
- [3] T. Chen, H. Leung, and I. Mak. Adaptive random testing. In M. Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 3156–3157. Springer Berlin / Heidelberg, 2005.
- [4] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011.
- [5] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.
- [6] G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the International Conference on Software Testing*, ICST 2017. IEEE, 2017.
- [7] G. Gay. Generating effective test suites by combining coverage criteria. In *Proceedings of the Symposium on Search-Based Software Engineering*, SSBSE 2017. Springer Verlag, 2017.
- [8] G. Gay, M. Staats, M. Whalen, and M. Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, PP(99), 2015.
- [9] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.
- [10] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.
- [11] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.