Automatic Test Case Generation and Test Suite Reduction for Closed-Loop Controller Software

Christian Murphy, Zoher Zoomkawalla, Koichiro Narita
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{cdmurphy, zoher, knarita}@cis.upenn.edu

***Abstract*—Domains such as embedded systems, medical devices, process automation, etc. make use of controller software to make important decisions that can affect people's lives and well-being. Although safety-focused processes such as model-driven development can be used to assure a certain degree of quality in these applications, ultimately software testing still remains the primary mechanism by which faults are detected.**

However, a variety of challenges arises in identifying test cases for controller software, particularly in closed-loop systems that incorporate feedback from the entity being controlled, potentially leading to exponential growth in the number of paths through the code and difficulty in identifying sequences of inputs to put the application into the desired states for testing.

In this paper, we present an approach to efficiently generating a set of test cases that will cover all reachable states in closed-loop controller software, describe how it is possible to reduce the number of test cases without losing any coverage of states, and present evidence that, compared to other approaches, the technique significantly reduces the number of test cases (down to less than 1% in our experiments) needed to achieve the same level of coverage, with almost no negative effects on the test suite's fault-finding capabilities.

I. Introduction

Software is increasingly being used to make control decisions in domains such as embedded systems, medical devices, process automation, cyber-physical systems, etc. Typically, a sensor is used to detect the status of whatever is being controlled (often referred to as the "plant"), the software makes some decision according to some rules or protocol, and the output indicates what action to take.

When the system incorporates feedback from the plant, the software is said to be a *closed-loop controller*. For example, in medical cyber-physical systems, closed-loop controller software can be found in automated insulin pumps, pacemakers, anesthesia devices, etc. The input to the software consists of some aspects of the patient's physiological condition, and the output is the action to take, such as how much medicine to deliver; the software maintains state, e.g. the values of past readings, and then awaits another input of the patient's condition before deciding on the next action, and the loop continues.

Clearly, the quality assurance of such controller software is of the utmost importance: the cost (either monetary or in terms of health and lives) of faults in the implementations can be enormous. And although safety-focused processes such as model-driven development [10] can be used to a certain extent, ultimately there is actual code running on actual hardware, and software testing still remains the primary mechanism by which faults are detected.

In addition to the importance of testing such software, one challenge that arises in testing closed-loop controllers in particular is that the "input" is actually a sequence of input values that are read from the plant or the environment. To test such software with a high degree of confidence, it is necessary to identify sequences of input values (test cases) that would put the software into each of the states in which it should be tested.

Efficiently generating a set of test cases that cover all possible states can be challenging for three reasons. First, even if it were possible to enumerate all paths through the program, so that all states are eventually reached, this can be incredibly time consuming, even when automated, and would likely yield a huge number of test cases. Second, aside from the fact that it is time consuming, it may be unnecessary, since many test cases ultimately put the program into the same state (or semantically equivalent states), and it may only be necessary to test the application in each state once, regardless of how it was reached. However, automatically identifying even a single path to each state can be as difficult as identifying all of them. Third, even if we can identify one path to each state, often there will be overlap in the paths, such that an individual state may be covered by multiple test cases; if our objective is to find the smallest number of test cases (or, at least, a relatively small number) that will cover all states, we would want to reduce the test set to remove any redundancy.

In this paper, we present an approach to efficiently generating a small set of test cases that will cover all (reachable) states in closed-loop controller software, and then further explain how it is possible to reduce the number of test cases without losing any coverage of states. Additionally, we describe an implementation of our approach for C programs, and present evidence that, compared to simply enumerating all possible paths through the program, the approach significantly reduces the number of test cases needed to achieve the same level of coverage, with almost no negative effects on the test set's fault-finding capabilities.

Although we focus on closed-loop controller software in this paper, and medical device software in particular in our experiments, the results are generalizable to any software for which the input consists of a sequence of input values and the implementation includes repeatedly-called functions that have numerous paths (states) that need to be covered by the test set.

The rest of this paper is organized as follows. Section II presents our approach using a running example, and Section III explains how we implemented the approach for programs written in C. Section IV discusses our empirical studies, in which we measure the reduction in the number of test cases, the time it takes to generate them, and the impact on the effectiveness of the test set. Section V presents related work, Section VI discusses possible future directions, and Section VII concludes.

II. Approach

Because our approach to generating test cases for closed-loop controller programs assumes that we have access to source code, and not a state diagram, we will focus on path coverage as our goal, since covering all reachable paths through the code would imply covering all reachable states.

In this paper, we use *path* to mean the sequence of statements from the beginning of the program's execution to its end, and *sub-path* to mean the sequence of statements from the beginning of a specified *function's* execution to its end.

Our approach to generating test cases for closed-loop controller programs is based on the intuition that, in order to cover all states, it is not necessary to cover all paths through the program, as long as we cover all sub-paths through the various functions. This intuition will be explained below.

The approach consists of two steps:

1) Identify a set of test cases that covers all function sub-paths in the program under test
2) Remove any test cases that only cover sub-paths already covered by some other tests

As a running example in this section, consider the C program illustrated in Figure 1. This program implements a closed-loop controller that takes an integer value as input. The controller produces a 0 and stops if it observes the same input three times in a row; otherwise, it produces a 1 and continues processing. The controller also stops after seeing ten inputs without seeing the same value three consecutive times.

There are five unique sub-paths through the `update` function. Given that it is called up to 10 times, if we wanted to cover each path we would need up to 5^{10} = 9,765,625 test cases! Fortunately, because there are ways to break out of the loop before the function is called 10 times (specifically, on lines 7 and 16), the number of paths through this program is only 143. Thus, to achieve 100% path coverage in this program, we would need 143 test cases, each of which is a sequence of inputs (generated/read on line 24).

As mentioned above, our intuition is that 100% path coverage is not necessary in order to test all states. For instance, we note that some input sequences lead to the same state, even though their paths are different: for instance, the test case input sequences 4-7-3-8 and 6-6-2-8 both put the application in the same state (`consec` = 1, `count` = 4, `last` = 8), but the fact that in the second test case we at one point saw the same number twice in a row, which took us through a different path, no longer matters in the current state.

```
 1  int consec=1, count, last;
 2
 3  int update(int input) {
 4     if (input == last) {
 5        if (++consec == 3) {
 6           //Done! Three in a row!
 7           return 0;
 8        }
 9     }
10     else consec = 1;
11
12     last = input;
13
14     if (++count == 10) {
15        //Done! Count is ten!
16        return 0;
17     }
18     else return 1;
19  }
20
21  main() {
22     int input, output;
23     do {
24        input = ... // get next input
25        output = update(input);
26        printf("Output: %d", output);
27     }
28     while (output);
29  }
```

Fig. 1. Simple controller to detect when the same input value (line 24) is seen three times in a row.

Additionally, some sequences lead to semantically similar states. For instance, 5-1-1-1, 3-8-1-1-1, and 6-6-1-1-1 all cover the state in which a number is seen three times in a row; the fact that we have seen a total of four numbers or five numbers up to that point, or that we at one point did or did not see the same number twice in a row, is not significant.

Because some sequences lead to the same state or semantically similar states that can be eliminated, we can reduce the number of test cases by focusing only on the five unique sub-paths through the "update" function, as shown in Table I.

Because we are testing using sequences of values for `input`, we assume that we cannot directly manipulate the other variables (`last`, `consec`, and `count`) in order to create test cases to satisfy these path conditions. However, by focusing only on covering the five sub-paths through the `update` function, we can reduce the size of the test set from 143 test cases to just five. For instance, the set of test cases

TABLE I
SUB-PATHS THROUGH THE `update` FUNCTION

	Sub-path Line Numbers	**Path Condition**
A	4-5-7	`input == last` and `consec == 3`
B	4-5-12-14-16	`input == last` and `consec != 3` and `count == 10`
C	4-5-12-14-18	`input == last` and `consec != 3` and `count != 10`
D	4-10-12-14-16	`input != last` and `count == 10`
E	4-10-12-14-18	`input != last` and `count != 10`

TABLE II
TEST CASES THAT COVER ALL FIVE SUB-PATHS

Test Case	**Sub-path**	**Input Sequence**
T1	A	1-1-1
T2	B	1-2-3-4-5-6-7-8-9-9
T3	C	1-1
T4	D	1-2-3-4-5-6-7-8-9-1
T5	E	1-2

described in Table II would cover each of the five sub-paths described in Table I.

As pointed out in various other works (e.g., [4], [19], [13], etc.), we can further reduce the number of test cases in our test set by noting that there is overlap in the sub-paths that are covered by these five tests. Although the initial reduction from 143 test cases down to just five is significant, we can apply a test suite reduction strategy and remove any test cases that are unnecessary.

TABLE III
SUB-PATH COVERAGE MATRIX FOR EACH TEST CASE

	A	**B**	**C**	**D**	**E**
T1	X		X		
T2		X			X
T3			X		
T4				X	X
T5					X

The coverage matrix shown in Table III shows, for instance, that in order to get to sub-path D, test case T4 also goes through sub-path E. Thus, since sub-path coverage is our goal, it is unnecessary to have separate test cases for D and E, since a test that covers D will also cover E. So we further reduce the test set by finding the smallest subset that covers all five sub-paths, giving us a final test set comprised of T1, T2, and T4.

We have now reduced our original 143 test cases down to just three, and we still manage to cover all of the sub-paths through the `update` function, and thus cover all of the states that we want to test.

III. IMPLEMENTATION

Here we describe the implementation of the two steps to our approach as described in Section II. We chose C for our implementation because of the availability of tool support, but the approach itself is not language-dependent, and implementations could conceivably be built for other languages.

A. Test case generation

The first step in our approach is to generate a set of test cases that will cover all sub-paths through the program, based on the implementation of the code. We started with the test case generation tool KLEE [3], which uses symbolic execution to identify path conditions for all reachable paths, and then uses a constraint solver to generate test cases that satisfy those path conditions. KLEE supports the generation of sequences of input values, which is required by the types of programs that we want to test.

By default, KLEE will attempt to generate test cases that cover *all* paths through the program, though as described above, many of those test cases lead to equivalent or semantically equivalent states. Our goal, of course, is not to achieve 100% path coverage through the whole program, but rather to cover 100% of the sub-paths through the various functions.

Fortunately, KLEE has a runtime option "-only-output-states-covering-new," which reports only test cases that cover sub-paths that have not been previously covered. As we discuss later, this is not an attempt to find a global *minimum* set of test cases that cover all sub-paths, but rather KLEE prunes its search tree by focusing on paths that it has not previously seen.

In the example from the previous section, KLEE generated 143 test cases to cover all paths (as expected), and then the five test cases to cover the five sub-paths through the `update` function (also as expected) when using the runtime option to only cover new states.

B. Test suite reduction

In order to further reduce the test set, such that we remove any test case that only covers sub-paths that are already covered by other tests, we consider the coverage matrix and then apply a solution to the Minimum Set Cover Problem in order to find the minimum subset that covers all sub-paths.

Although this problem is known to be NP-hard, the greedy solution typically gets optimal or close-to-optimal results [5], and we implemented it in Java for the purposes of this work. For the programs we considered in our empirical studies below, solving the problem by enumerating through all 2^n possible combinations was certainly tractable, but we acknowledge that that may not always be the case.

For the example in the previous section, the tool we implemented would take the coverage matrix from Table III and remove test cases T3 and T5. Thus, using a combination of KLEE and our minimization tool, we reduced the total number of test cases from 143 to just three.

IV. EVALUATION

To determine the effectiveness of our solution, we conducted experiments in which we focused on the following three questions:

1) By how much does the new approach reduce the number of test cases?
2) By how much does the implementation of the approach reduce the time to generate tests?
3) What effect does reducing the number of test cases have on the fault-finding capabilities of the test suite?

In order to answer these questions, we applied our approach to five controller programs that are based on implementations of real-world medical protocols, specifically those used at the Hospital of the University of Pennsylvania to determine the amount of insulin to deliver to patients with various medical conditions. The protocols are hereafter referred to as: *PennNeuroICU*, *PennCardiac*, *PennMICU*, *PennHyperGlycemia*, and *PennIntraoperative*.

The implementations were written in C and receive as input the patient's current blood glucose level and then output the rate of insulin delivery according to the specific protocol. The state that is maintained includes the previous glucose level reading and the current delivery rate, both of which are used by the protocols to determine how to adjust the delivery of insulin based on the current input reading. For our purposes, the programs terminate when a simulated time of delivering insulin has elapsed, or when an invalid blood glucose reading is detected (specifically, one that is too high or too low to be handled by the protocol and would, in practice, necessitate emergency intervention). Details of the implementations are provided in Table IV.

TABLE IV
PROGRAMS USED IN EXPERIMENTS

Program Name	Simulation Time	Lines of Code	States
PennNeuroICU	240 min	215	28
PennCardiac	210 min	167	22
PennMICU	270 min	194	32
PennHyperGlycemia	180 min	185	30
PennIntraoperative	210 min	180	38

A. Test Cases

In order to answer the first research question, "By how much does the new approach reduce the number of test cases?", we first used KLEE to generate test cases for *all* paths through the programs (note that the simulation time was bounded for each protocol, so that there would not be infinite paths). We then used KLEE to generate only those test cases that covered new sub-paths, and lastly performed test suite reduction to remove unnecessary test cases.

As shown in Table V, the impact of using the approach is quite significant. For instance, for the *PennNeuroICU* implementation, KLEE generated 101,928 test cases in order to cover all possible paths through the program when executing the protocol for a simulated time of up to 240 minutes of insulin delivery. However, when we only considered the distinct sub-paths through the functions used in the implementation (specifically, the ones used to calculate the initial insulin delivery rate, and then to readjust it), only 31 test cases were needed in order to test the implementation in all of the states of the protocol. In addition, we were able to reduce the test set down to 27 test cases by removing those that did not cover states that were not already covered, reducing the total test set to around a hundredth of a percent of its original size.

In general, across all five applications, the number of test cases needed to cover all sub-paths is less than 1% of the number needed to cover all paths through the program. Furthermore, we can reduce the test sets by a further 10-20% by removing test cases that only cover sub-paths that are already covered by other test cases.

It is worth pointing out that, although we used KLEE to obtain the numbers in the second and third columns of Table V, any other tool that correctly enumerates all paths and correctly identifies distinct sub-paths should produce the same results.

However, the size of the reduced set of generated tests is somewhat an artifact of using KLEE since another tool may have produced a different set of test cases to cover all sub-paths. That is, our approach does not necessarily generate the true *minimum* number of test cases, since we start with a set of test cases that covers all sub-paths and then reduce those, but the initial set is not necessarily attempting to cover all sub-paths in as few test cases as possible. For instance, there could conceivably be one single test case that covers *all* sub-paths, but it may not be revealed by the initial exploration of the code. To find a true minimum number of test cases that cover all sub-paths, we may need to enumerate over all paths through the program, which would of course defeat the point of our approach in terms of time saving.

Regardless, this experiment demonstrates quite clearly that for these types of programs that consist of repeatedly-called functions with potentially huge numbers of paths, an approach that only considers sequences of inputs that cover sub-paths through the various functions in order to test all reachable states will generate a far smaller and more manageable set of test cases.

B. Time

In answering the second research question, "By how much does the implementation of the approach reduce the time to generate tests?", we measured the time it took to run KLEE and our tool as described in the previous experiment. Using a machine running Mac OS X on a 2.2 GHz Intel Core i7 processor with 8GB RAM, we obtained the results shown in Table VI. The third column shows the combined time of using KLEE to generate the test set that only covers new sub-paths, plus the time to reduce it further to remove unnecessary test cases.

As expected, the time it takes for KLEE to generate the smaller set of test cases is significantly reduced, up to over 90% in some cases. In total across all five applications, the time saved by only considering newly covered sub-paths was literally more than three days of computing.

Note that, when considering the number of test cases (Table V) and the time it takes to generate them (Table VI), the results are not at all linear. For instance, it took twice as

TABLE V
NUMBER OF GENERATED TEST CASES.

Program Name	All Paths	Only New Sub-paths	Reduced Set
PennNeuroICU	101,928	31 (-99.969%)	27 (-12.9%)
PennCardiac	3,052	26 (-99.148%)	22 (-15.4%)
PennMICU	61,195	33 (-99.946%)	28 (-15.2%)
PennHyperGlycemia	45,106	30 (-99.933%)	24 (-20.0%)
PennIntraoperative	22,146	56 (-99.747%)	49 (-12.5%)
Total	**233,427**	**176 (-99.924%)**	**150 (-14.8%)**

TABLE VI
TIME TO GENERATE TEST CASES.

Program Name	All Paths	Only New Sub-paths + Reduced Set
PennNeuroICU	63h 3m 15s	15h 31m 8s (-75.4%)
PennCardiac	0h 5m 5s	0h 1m 32s (-69.8%)
PennMICU	21h 30m 12s	1h 12m 34s (-94.3%)
PennHyperGlycemia	10h 15m 35s	0h 53m 48s (-91.2%)
PennIntraoperative	2h 40m 27s	2h 20m 48s (-12.2%)
Total	**97h 34m 34s**	**19h 59m 50s (-79.5%)**

long to generate the test cases for *PennMICU* as it did for *PennHyperGlycemia*, even though *PennMICU* only had about 33% more paths. Additionally, although we reduced the size of the test set by over 99% for each program, the reduction in time varied greatly from 12% all the way up to 94%. Clearly these results are all related to using KLEE in our implementation, and depend on the manner in which it explores the program and solves path conditions, but the point is still made that a large amount of time can potentially be saved by reducing the number of tests cases that are generated.

Moreover, although we only considered test set generation time and not test set execution time, it follows that a test set that is 99% smaller should run about 99% faster, thus saving even more time during development and regression testing.

C. Effectiveness

To answer the final research question, "What effect does reducing the number of test cases have on the fault-finding capabilities of the test suite?", we used mutation analysis to systematically insert faults into the source code and then determined whether removing some test cases would hurt the ability to detect those faults.

In particular, we inserted mutations related to arithmetic operators (e.g., changing "+" to "-"), boolean operators (e.g., changing ">" to "<"), logical operators (e.g., changing "&&" to "||"), and assignment operators (e.g., changing "=" to "+ ="). Each variant of the program had exactly one mutation, i.e., we did not create any program variants with more than one fault inserted.

A fault was considered to be detected (or, the mutant is considered "killed"), if any test case in the test set caused the program to produce the incorrect output, using the original (unmutated) version as the golden implementation. The goal was not so much to determine the fault-finding capability of the tests generated by our approach (though this is discussed below), but rather to see whether removing test cases by reducing the test set would diminish the number of faults that could be found.

As shown in Table VII, our approach to reducing the number of test cases in the test set had almost no negative effect on the fault-finding capabilities: only two faults detected by the full test set were missed by the reduced set. That is to say, for those two mutations, our test set reduction technique removed some test cases that were the only ones among the entire set that detected those defects. Upon further investigation, we determined that this happened because the values chosen by KLEE for those test sets coincidentally revealed those bugs, but the other test cases that covered the same sub-paths did not; how this could happen is discussed below.

We observe here that the number of faults that were detected by the test sets is somewhat small, around 72% in total. This is primarily an artifact of using a white-box testing technique (based on path coverage) to generate tests that are not necessarily attempting to detect the different types of mutations that we inserted. As one would expect, our test cases were effective at revealing mutations that affected decisions in the code (e.g., mutating "<" to ">"), since these resulted in the wrong path being taken, and thus the incorrect behavior from the controller. Additionally, mutations that affected the calculations performed by the controller (e.g., mutating "+" to "-") were often revealed since the same path was covered but the output had the incorrect value.

However, some mutations proved very hard to kill, especially those that generally result in the same path being taken but do not change the output value. In our case, this happens because KLEE only seeks to generate inputs based on the path condition, and does not try to come up with inputs that are "close" to the values against which they are compared. For example, anything that satisfies "x $<$ 10" will satisfy its mutation "x $<=$ 10", and almost any value that violates "x

TABLE VII
FAULT-FINDING CAPABILITIES OF TEST SETS.

Program Name	Faults Inserted	Detected by Full Test Set	Detected by Reduced Test Set
PennNeuroICU	241	189	188
PennCardiac	160	126	125
PennMICU	238	191	191
PennHyperGlycemia	195	158	158
PennIntraoperative	256	125	125
Total	**1090**	**789**	**787**

$<$ 10" will violate its mutation "x $<=$ 10", except for the particular case where x is 10, of course. The same holds true for the situation in which "A && B" is mutated to "A || B". The only way in which these mutants would be killed is when the condition should normally evaluate to false, but the mutated version evaluates to true, e.g. if KLEE coincidentally chose a value of 10 for x in an attempt to violate the condition in the first example. As it turned out, this was the case for the two tests in our experiment that revealed such faults but then happened to be removed by the test set reduction strategy.

We noted anecdotally that, after analyzing the test cases generated by KLEE, when we hand-generated some additional test cases by using boundary analysis (a black-box technique) based on the values specified in the original protocols, we were able to raise the effectiveness up to over 90%, primarily because we were able to detect faults related to changes to boolean operators (such as a "$<$" that had been mutated to a "$<=$") that essentially resulted in off-by-one errors.

This is not to say that black-box tests are more effective than white-box tests, nor is it to say that mutation analysis is not an appropriate means of measuring effectiveness (see [2] for a discussion on that topic). In the case of closed-loop controllers, it would have been very difficult to hand-generate black-box test cases that consist of sequences of inputs that would allow us to reach each sub-path; using our white-box approach was necessary as a starting point. Thus, this is more evidence of the claim that black-box and white-box tests should be used in conjunction with each other.

But more importantly for our purposes, the results of this part of the experiment show that a reduced set of white-box tests can be generated much more quickly than a set that simply covers all paths, with almost no decrease in fault-finding capabilities as measured in this manner.

V. RELATED WORK

There has, of course, been substantial research in the area of automated test case generation in general (e.g., [8], [3], [14], [18], etc.) as surveyed by McMinn [16] in 2004, Ali et al. [1]in 2010, and many others. Some research in this area has sought domain-specific strategies, e.g. for GUIs [17], Web Services [23], etc., and there has also been work in generating test cases for real-time embedded systems [7] [20] [21]; however, none of this previous work has looked specifically at generating test cases for closed-loop controllers as we do here.

Research into the testing of controller software [12] [15] in domains such as automotive [6] [22] has focused on the mechanisms of conducting the testing and not on the automatic generation of test cases from an implementation. Moreover, as with the works cited above, none specifically address the challenges of creating and then reducing a test set for a closed-loop controller.

VI. FUTURE DIRECTIONS

Future work in this area could involve addressing some of the issues raised during our experiments, for instance seeking to determine whether it is possible to quickly find a true minimum set of test cases that will cover all sub-paths. It would, of course, be possible to do this by enumerating all possible paths and then applying a solution to the minimum set cover problem, but the time to compute that set would be enormous. It may instead be preferable to design a tool that would specifically attempt to solve this problem by first trying to achieve sub-path coverage with just one test case, then with two tests, then three, and so on until a minimum number of test cases is found.

Additionally, more work could be done in evaluating the suitability of white-box vs. black-box test cases for such programs. Our goal was to automatically generate test cases based on the code, but we noted above that we could improve the fault-finding capability of the test set by manually adding extra test cases based on the protocol, i.e. the specification. Although this result is not surprising, future work could try to quantify the benefits of doing so, and perhaps look into automatically creating test cases from the free-text specification [11], or from a state diagram or model that comes from those specs [9]. It may also be possible to create a hybrid approach in which a white-box technique like the one presented here is used to identify the path conditions to reach certain sub-paths, but then a black-box technique is used to find condition-satisfying input sequences that will specifically be most likely to reveal defects.

Last, we acknowledge that there may be threats to the validity of our experiment in terms of the target applications and the tool we used, and thus further investigation would be required to determine the effectiveness of the approach when applied to closed-loop controllers in other domains, or for implementations in other programming languages.

VII. CONCLUSION

As the use of software for process control purposes increases in domains like cyber-physical systems and medical

devices, so too does the need to quickly generate test cases that will cover all possible states in which the software should be tested, so as to increase the chance of finding faults in the software and thus its quality.

In this paper, we have presented an automated approach to generating and then reducing such a set of test cases. Using the tool KLEE, we identify only a small set of test cases that covers each sub-path, and then remove test cases that only cover sub-paths covered by other tests, using a solution to the Minimal Set Cover Problem. Our experimental results demonstrate that our implementation could reduce the number of test cases by tens of thousands and test case generation time by dozens of hours in applications in the domain of interest, with almost no negative impact on fault-finding capability.

As the ultimate goal of this research is to identify ways to improve the quality of controller software by making it easier to test and analyze, we feel that this work is an important first step in that direction.

VIII. Acknowledgments

The authors would like to thank Prof. Insup Lee for suggesting this line of work, and for his advice in evaluating it. We also thank Nikos Vasilakis for his initial investigation into generating test cases, Shweta Tyagi for implementing the programs used in the evaluation, Eric O'Brien for assisting with the mutation analysis experiment, and Sanjian Chen for his help with the insulin protocols. This research was supported in part by NSF CNS-1035715.

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. *IEEE Transactions on Software Engineering*, 36(6):742–762, Nov/Dec 2010.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on operating systems design and implementation*, pages 209–224, Dec. 2008.

[4] T. Y. Chen and M. F. Lau. A new heuristic for test suite reduction. *Information and Software Technology*, 40:347354, Jul 1998.

[5] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, Aug. 1979.

[6] M. Conrad and I. Fey. Systematic model-based testing of embedded automotive software. *Electronic Notes in Theoretical Computer Science*, 111:13–26, Jan. 2005.

[7] S. J. Cunning and J. W. Rozenblit. Automatic test case generation from requirements specifications for real-time embedded systems. In *Proceedings of the 1999 IEEE conference on systems, man, and cybernetics*, pages 784–789, 1999.

[8] R. A. DeMilli and A. J. Offut. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sep 1991.

[9] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the First International Symposium of Formal Methods Europe: Industrial-Strength Formal Methods*, pages 268–284, 1993.

[10] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the 2007 Future of Software Engineering*, pages 37–54, 2007.

[11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on foundations of software engineering*, pages 146–162, Nov. 1999.

[12] J. Hawkins, R. B. Howard, and H. V. Nguyen. Automated real-time testing (ARTT) for embedded control systems (ECS). In *Proceedings of the annual reliability and maintainability symposium*, pages 647–652, 2002.

[13] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, Mar 2003.

[14] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, Aug 1990.

[15] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international conference on embedded software*, pages 299–306, 2005.

[16] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, Jun 2004.

[17] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, Feb 2001.

[18] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, Dec 1999.

[19] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, Dec 2002.

[20] T. Tekcan, V. Zlokolica, V. Pekovic, N. Teslic, and M. Gunduzalp. User-driven automatic test-case generation for DTV/STB reliable functional verification. *IEEE Transactions on Consumer Electronics*, 58(2):587–595, May 2012.

[21] W. T. Tsai, L. Yu, X. X. Liu, A. Saimi, and Y. Xiao. Scenario-based test case generation for state-based embedded systems. In *Proceedings of the 2003 IEEE performance, computing, and communication conference*, pages 335–342, 2003.

[22] J. R. Wagner and J. S. Furry. A real-time simulation environment for the verification of automotive electronic controller software. *International Journal of Vehicle Design*, 13(4):365–377, 1992.

[23] Y. Zheng, J. Zhou, and P. Krause. An automatic test case generation framework for web services. *Journal of Software*, 2(3):64–77, Sep 2007.