Evaluating Specification-level MC/DC Criterion in Model-based Testing of Safety Critical Systems

Syed S. Arefin
Microsoft Corporation
Vancouver, BC, Canada
ssarefin@cs.umanitoba.ca

Hadi Hemmati
University of Calgary
Calgary, AB, Canada
hadi.hemmati@ucalgary.ca

Howard W. Loewen
Micropilot Inc.
Winnipeg, MB, Canada
hloewen@micropilot.com

ABSTRACT

Safety-critical software systems in the aviation domain, e.g., a UAV autopilot software, needs to go through a formal process of certification (e.g., DO-178C standard). One of the main requirements for this certification is having a set of explicit test cases for each software requirement. To achieve this, the DO-178C standard recommends using a model-driven approach. For instance, model-based testing (MBT) is recommended in its DO-331 supplement to automatically generate system-level test cases for the requirements provided as the specification models. In addition, the DO-178C standard also requires high level of source code coverage, which typically is achieved by a separate set of structural testing. However, the standard allows targeting high code coverage with MBT, only if the applicants justify their plan on how to achieve high code coverage through model-level testing.

In this study, we propose using the Modified Condition and Decision coverage ("MC/DC") criterion on the specification-level constraints rather than the standard-recommended "all transition coverage" criterion, to achieve higher code coverage through MBT. We evaluate our approach in the context of a case study at MicroPilot Inc., our industry collaborator, which is a UAV producer company. We implemented our idea as an MC/DC coverage on transition guards in a UML state-machine-based testing tool that was developed in-house. The results show that applying model-level MC/DC coverage outperforms the typical transition-coverage (DO-178C's required MBT coverage criterion), with respect to source code-level "all condition-decision coverage criterion" by 33%. In addition, our MC/DC test suite detected three new faults and two instances of legacy specification in the code that are no longer in use, compared to the "all transition" test suite.

KEYWORDS

Safety critical systems, model-based testing, search-based testing, MC/DC, code coverage, constraint coverage, DO-178C, software certification, Aviation, UAV.

ACM Reference Format:
Syed S. Arefin, Hadi Hemmati, and Howard W. Loewen. 2018. Evaluating Specification-level MC/DC Criterion in Model-based Testing of Safety Critical Systems. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183551

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183551

1 INTRODUCTION

Avionics systems are safety-critical systems embedded in an aircraft hardware that controls and monitors the aircraft. The Unmanned Aerial Vehicle (UAV) is an avionics system with no pilot on board. A software, called autopilot, helps automate the process of controlling and guiding the UAV's navigation [2] . Safety critical systems in aviation, railway, and automotive domains often face a formal safety certification process. The safety certification ensures systems' safe and risk-free operations. The certification also ensures that the system will not cause any harm to its user, general public or the environment [21]. To prevent catastrophic effects, the regulatory authorities and the avionics industry have defined DO-178C [23], a rigorous certification standard for avionics systems.

MicroPilot Inc. [20] (our industrial partner in this project) is a commercial UAV manufacturer, which needs this certification for their avionics systems. Their autopilot system is developed using embedded system C code under the Visual Studio development environment. The certification standard provides a list of suggestions both on the process of software development and on the final product metrics. One of the major demands of DO-178C is having a set of explicitly written "software requirements" for the safety critical software and a set of test cases that verifies those requirements (requirement coverage). The test cases are also required to provide high source code coverage (structural coverage), e.g., in terms of statement coverage, decision coverage, and Modified Condition and Decision (MC/DC) coverage [23].

The DO-178C standard recommends following model-driven engineering (MDE) to achieve high requirement coverage, which is explained in a supplement document (DO-331). MDE's proposal for automated test generation is called Model-Based Testing (MBT). MBT accepts a specification model of the software under test as an input and generates test cases that verify all the requirements specified in the model and covered by a model-level coverage criterion (such as "all-transition" coverage on state machine models).

Given that an applicant of DO-178C is very likely to follow DO-331 recommendation and implement MBT to achieve requirement coverage, one of the FAQs noted in DO-331 is: *"May the applicant use the model coverage analysis activity to achieve the structural coverage analysis objectives?"*. The standard's answer to this question, in short, is YES but only if the applicant can show that the structural coverage objectives have been achieved. In other words, there is

no guarantee that a test suite generated by the recommended MBT will cover "enough" of the source code.

To examine how well MBT test suites cover the source code and whether one can use MBT to also achieve high structural code coverage, in this paper, we propose leveraging specification model constraints and targeting MC/DC coverage on the model-level, to achieve higher code coverage. The motivations for this idea is that the constraints in the model, eventually, will be translated into source code level conditions. Therefore, imposing a model constraint-level coverage will result in higher "all condition/decision" coverage of the underlying source code as well. This probably would not happen by a simple "all-transition" coverage on the specification level, which is recommended as a default for MBT in the standard.

To empirically evaluate this idea, we have conducted an experiment in our industry collaborator's site. We have an ongoing research collaboration with MicroPilot since early 2016, where one of our projects is on automated test generation using MBT, to partially satisfy DO-178C requirements. Throughout that project we have developed an in-house MBT tool that accepts manually developed state machine representations of the system as input and automatically generates executable system-level test cases. The default coverage criterion of the MBT tool, according to the standard, is "all-transition" coverage.

In this study, we have implemented the concept of MC/DC on the state machine constraints and compared its source code level coverage and defect detection ability with those of the default criterion. The experiment compares test suites of the two approaches when applied on a state machine representing one representative flight command's behaviour. The results show that, the proposed approach achieves an extra 65% MC/DC model constraint coverage compared to MBT's default criterion. This results in an extra 33.5% coverages of all conditions/decisions in the source code and detects three new faults in the system plus two legacy specifications that are no longer in use.

2 BACKGROUND

In this section, model-based testing, MC/DC code coverage criteria, and search-based test data generation will be very briefly explained.

2.1 Model-Based Testing

The objective of Model-Based Testing (MBT) [13] is to automatically generate executable test cases based on the system specification models. These models typically represent functionality but also can include performance, safety, and security concerns. The MBT process is composed of the following steps:

- The test designer manually models the system under test (and if required its environment), as specification models (e.g., UML state machines).
- The MBT tool automatically generates abstract test cases from the model, by applying a model coverage strategy (e.g., covering all states or transitions in a state machine) to create a set of test paths. These test paths identify the scenarios that are going to be verified.
- The abstract test cases are too generic and need language-specific data to be executable. In addition, to generate executable tests, the MBT tool needs to add specific input data values for each method call in a test path and system settings, which will be discussed in the search-based test generation section.
- Finally, the generated test cases are executed (typically within a test execution framework) and the outputs are analyzed and reported.

2.1.1 Modified Condition/Decision Coverage (MC/DC). One of the well-known code coverage criteria is Decision coverage, which represents what percentage of source code branches has been covered by a test suite. Condition/Decision (C/D) coverage is stronger version of Decision coverage, where not only all decisions are counted by also all individual conditions in a decision. This coverage criterion ensures the possible outcome of each condition of the decision are tested at least once. Modified Condition/Decision Coverage (MC/DC) criterion is a modification of C/D coverage proposed by the safety certification [23]. Unlike C/D it emphasizes on independent effect of each condition of each decision. To meet the MC/DC criterion, a test suite must meet all the followings:

(1) Every point of entry and exit in the program must be invoked at least once.
(2) Every condition in a decision in the program must take all possible outcomes at least once.
(3) Every decision in the program must take all possible outcomes at least once.
(4) Each condition in a decision must show independent effect in the decision's outcome.

To show a condition's independent effect on a decision's outcome one can vary just that condition while keeping all the other possible conditions fixed. Note that MC/DC is a weaker version of all C/D coverage criterion.

2.2 Search-Based Test Data Generation

Search Based Test Data Generation is an approach that transforms test data generation problem into optimization problems [12], where the objective of the test generation is implemented by a fitness function that guides the search. Among the many meta-heuristic search techniques used for test generation, Genetic Algorithms are perhaps the most common [3]. In a Genetic Algorithm, randomly selected candidate solutions are evolved by applying evolutionary operators, such as mutation and crossover, resulting in new offspring individuals, with better fitness values. An example objective function for unit test generation is the whole test suite's code coverage [5].

(1+1) Evolutionary Algorithm (EA) is a very basic and simpler search algorithm, compared to the genetic algorithms. In this algorithm the population size is one and after mutation a new individual is created. The newly created individual competes with it's parent to become the parent of next generation. We will explain this algorithm in more details in Section 4.4.1

Table 1: Modified Condition/Decision Coverage of a Sample Predicate.

	$A > B$ *and* $X > Y$ *and* $P > Q$			Predicate Eval.
	$A > B$	$X > Y$	$P > Q$	
1	TRUE	TRUE	TRUE	TRUE
2	TRUE	TRUE	FALSE	FALSE
3	TRUE	FALSE	TRUE	FALSE
4	FALSE	TRUE	TRUE	FALSE

3 THE MC/DC CRITERION ON STATE MACHINE CONSTRAINTS

As discussed, two typical coverage criteria applied on state-machines for test generation are "all state coverage" and "all transition coverage". The DO-178C's MDE supplement (DO-331) suggests using "all transition coverage" for state machines (Table MB. 6-1 in [1]). However, the same table recommends coverage of all decisions for "logic equations" and Table A7 in DO-178C [23] recommends "all statement coverage", "all decisions coverage", and MC/DC criteria for structural coverage.

In this section, we apply the concept of MC/DC on the state machine constraints. This will result in a higher coverage of high-level logical constraints in the model (which is the direct target of the criterion), as well as higher structural code coverage such as all C/D coverage in the source code.

The concept of MC/DC, as explained in Section 2, can be applied on any decision point which is represented as a boolean expression, composed of a set of atomic conditions (conditions that can not be broken into simpler expressions) "AND"ed and "OR"ed together. This means that for any transition in the state machine that has a guard predicate one can create a set of test inputs that satisfy MC/DC on that decision point. To be more specific, if a guard contains n atomic conditions then it takes $n + 1$ test cases to cover all MC/DC conditions of the guarded transition [17]. For example, a minimum of four test cases is required to meet the MC/DC criterion for the following guard condition ($A > B$ *and* $X > Y$ *and* $P > Q$), as shown in Table 1.

However, a test path typically includes several transitions where each may have a guard predicate. Note that conditions in different guards in a path can NOT be simply jointed by 'AND" operator to create one total predicate for a path. Since these guards will be evaluated at different times (from different states), it is very common to have contradictory conditions. For instance, guard 1: [$Speed == 0$] and guard 2: [$Speed > 0$]. Therefore, to generate a full test suite out of all these predicates, we make a Cartesian product out of all MC/DC test suites of all guards in the path.

Now if a test path contains $P = \{p_1, p_2, p_3, ..., p_n\}$ guard predicates, then there will be $(|p_1|+1)*(|p_2|+1)*(|p_3|+1)*...*(|p_n|+1)$ number of MC/DC combinations, where $|p_n|$ represents the number of conditions in predicate p_n. To give an example, if a test path has two guarded transitions as G1: [C1 *or* (C2 *and* C3)] and G2: [C4 *and* (C5 *or* C6)], then the total MC/DC test paths for this path is (3+1)*(3+1)=16.

4 EMPIRICAL EVALUATION

In this section we evaluate the proposed approach in the context of MicroPilot case study.

4.1 Objective

The goal of this study is to investigate the effect of applying model-level MC/DC coverage on test effectiveness.

4.2 Subject of study

The subject program of this study is MicroPilot's Autopilot software. The autopilot software is embedded in different types of UAVs such as Fixed-Wing, Helicopter, Multi-rotor Blimp, and VTOL (Vertical Takeoff and Landing). It helps automate different activities in the process of controlling and guiding of a UAV, such as taking off, keeping a safe altitude, climbing or descending to a predefined altitude, heading to an assigned waypoint location, landing on an assigned location, etc. An autopilot system is also programmed to handle critical situations such as heavy air turbulence, GPS failure, sensor failure, engine failure etc. The system can lead itself based on predefined flight commands or by a Ground Control System (GCS).

The implementation of MicroPilot's autopilot system is started nearly 20 years ago using ANSI C standard. Over time, the code-base has been evolved extensively and is still being updated day by day. At MicroPilot, each system feature undergoes a rigorous unit and system-level testing. In this study, we focus only on the system-level test generation, though.

Running a system-level test case with the actual hardware is very expensive. So, those test cases are executed in a controlled simulated environment. An actual implementation of autopilot software can run against the simulator, which simulates the sensors and other environmental elements to observe how the autopilot behaves in certain circumstances.

In the current practice of system testing at MicroPilot, test engineers manually write the test cases and execute them under the simulator with the help of a test execution framework. After a test passes on the simulator it goes to the real hardware execution.

MicroPilot autopilot software has three types of inputs that can be monitored and manipulated in the simulator: a) flight command parameters, b) autopilot settings, and c) the sensor inputs. The autopilot can perform several flight commands. Some flight commands contain parameters (e.g., climb(x) and flyto(x,y)) and some do not (e.g., takeoff(), circuit() and flare()).

The autopilot is highly configurable by a set of system variables. These settings are different in different types of vehicles. For example, a takeoff rotor speed needs to be defined in a Helicopter while in Fixed-wing a minimum defined ground speed needs to be defined for a successful takeoff. Users can modify these settings to customize and configure autopilot features they need. There are nearly 29,000 of such input system variables that not only define autopilot features but also define the communication links to the ground control systems. All input system variables have corresponding code-level variables. Any change in the system variables by the simulator immediately changes the code-level variables.

Autopilot senses the environmental data (e.g., temperature, pressure, GPS etc.) using several sensors, which affect the state of the

autopilot in many ways. For example, the speed of an autopilot depends on the GPS sensor and current altitude depends on the pressure sensor. Like input fields, sensor fields can also be monitored and manipulated from the simulator.

The autopilot also keeps another type of variable called state fields, which reveal information about the current state of the system. These fields are accessible through the simulator to monitor the current state but are not supposed to be modified by the user (though the simulator allows this feature for debugging purposes). There are a total of nearly 1,850 state fields in the system. For example current speed, current altitude, pitch, roll, yaw etc. State fields enable test engineers to monitor and test the system's state comprehensively. For example, upon submitting a command $climb(150)$, the system must maintain a certain pitch angle until it reaches target altitude of 150 meters.

4.3 Research Questions

RQ 1. How much does the proposed approach improve model-level constraint coverage? This question investigates the effect of applying a more demanding coverage criterion on the coverage of specification logic (represented by model-level constraints). Covering more of the specification logic, by itself, is a target toward certification. The higher model-level constraint coverage also implies that there might be a good chance of getting higher code-level condition coverage, which will be separately investigated in RQ2.

RQ 2. Does the proposed approach improve code coverage? This question compares the code-level "all condition-decision" coverage of two MBT test suites generated by applying "all transition" and model-level "MC/DC" coverage criteria.

RQ 3. Does the proposed approach detect new bugs? This question looks at the practical implication of applying the more demanding criterion and see if "MC/DC" test suite can detect any new bugs that are undetected by other testings.

4.4 Study Design

4.4.1 Test Data Generation. Each of those combinations described in section 3, is a different execution path of the software under test (SUT). To execute such paths, we need a set of test data. Since the guard variables are not necessarily the SUT's input variables, we need to first identify all input variables. In our case, as explained in Section 4.2, there are three types of input variables that can be used as test data: a) flight command parameters, b) autopilot settings, and c) the sensor inputs.

In general, the input data for transitions on a test path can be generated either transition-by-transition (each transition at a time) or all at once in the beginning of the test path. However, in our autopilot case, the flight command parameters and autopilot settings are set at once before the system starts, assuming the UAV is fully controlled by the autopilot instead of a ground control system. Changing these settings in the middle of the flight is unrealistic and a bad practice. However, the sensor inputs can potentially change over time during a flight, but given some technical limitations with simulator during test executions, we preset all test data including the sensor data at the beginning of each test case execution.

To generate the test data that satisfies the set of MC/DC conditions on the test path, we use a search-based test generation approach. Our main challenge for applying a search-based approach for test generation in this study was its cost. That is due to the fact that firstly the search space is very large. As explained, there are more than 29,000 autopilot settings and hundreds of sensor inputs and command parameter (of types boolean, integer, signed integer and geographical coordinate) for autopilot software. To reduce the search space, we did a manual dependancy analysis and excluded many variable that the expert identified as irrelevant. Ideally, we would like to automate this process using static analysis and slicing techniques.

Secondly, each fitness evaluation is expensive. In the rest of this section, we will explain our fitness function in details, but in short, each individual's fitness evaluation requires running a test case on the simulator. Note that these test cases are system-level test cases and they require setups and have timing constraints that make each test case execution very costly (several minutes per individual).

Therefore, using an expensive search algorithm (with large populations and many generations), such as Genetic Algorithm, was out of question. The alternative less expensive but also less powerful algorithms that we adopted here were (1+1) evolutionary algorithm and random search (as a baseline):

(1+1) Evolutionary Algorithm (1+1) EA uses an evolutionary strategy called mutation rate. The mutation rate controls the internal change an individual (in our case, a test case) goes through. For each gene (in our case, each input test data) in an individual (a test case), a random number in the range of [0-1] is generated. If the number is less than the mutation rate then the gene encounters a change (replacing the value of the input with another valid value), otherwise it is left unchanged.

We have applied a basic (1+1) Evolutionary Algorithm with 50% mutation rate and with maximum of 200 iterations. Due to the large search space and the expensive executions, we opt to use a high mutation rate (50%) that might help exploring the search space in a fast fashion. We also set the initial individual to the default settings. If the individual is fit, that is, the fitness function returns zero, then the individual is taken as the solution and the search ends, otherwise it undergoes a mutation again. This process keeps continuing until a termination criterion (in our case 200 iterations) is reached.

Random Search: To compare the performance of (1+1) Evolutionary Algorithm with a simpler baseline, we also have applied Random Search algorithm with the same number of iterations. The search is unguided and it generates a random individual and checks its fitness (same as (1+1)EA). It returns a solution if found, otherwise keeps iterating until the maximum iterations is reached.

4.4.2 Fitness Function. We have integrated our approach with the Micropilot's autopilot simulator to run the system under test with the input values an individual represents. During the execution of a flight, the simulator logs the values of the specified state fields over time. After the execution, the simulator stores the log in a file, which means that we can calculate the fitness only after the execution is over.

The objective of each search is to cover a path with a specific set of the model constraints. The quality of each individual is measured

by the fitness function. The fitness function used in our experiment is based on the common "branch distance" and "approximation level" concepts [28] in source code-level search-based software testing. Approximation level guides the search to satisfy guards in the order their transitions are visited in the test path. This means that, for instance, an individual that satisfies all the guard constraints except the last one on the path is fitter than the individual that satisfies all the guard constraints except the first one. To implement this, "approximation level" is defined as the number of uncovered guards from the back of the test path. For instance, if a test path contains 4 guarded transitions <[G1]T1, [G2]T2, [G3]T3, [G4]T4> and the test case satisfies the first guard (G1) but fail in the second (G2), then the approximation level for this test case is 3.0, since there are three guards that are remained unsatisfied by this test case.

Branch distance, however, is defined on only one guard (the first guard that the test case fails to satisfy) and measures how far the test case is to satisfy that guard's condition. In general, the guards on state machines are in OCL language. Therefore, to evaluate the fitness function, an OCL-based branch distance would be required [4]. However, in our case, the constraints were all simple boolean expressions and thus we followed a basic branch distance by Tracey et. al. [26].

To calculate a complex guard including ANDs and ORs, we first convert the guard constraint into a disjunctive normal form (DNF: $\vee_{j=1}^{n}(\wedge_{i=1}^{m}(C_{i,j}))$) and split it to multiple sub-constraints (conjunctive clauses: $\wedge_{i=1}^{m}(C_{i,j})$)). For example, a constraint $(x\ or\ y)\ and\ z$ splits into two constraints $(x\ and\ z)$ or $(y\ and\ z)$. So, if any of the sub-constraints evaluates to $true$ then the main guard constraint is also $true$. Therefore, the branch distance of the guard constraint is equal to the minimum branch distance of any of the sub-constraints.

$$B_t = \min_j b_{tj}$$

where b_{tj} is the normalized branch distance of a sub-constraints C_j created by splitting the guard constraint B_t (in DNF), on ORs. If an individual (a test case) covers a b_{tj} then it covers B_t as well.

Since each sub-constraint is a conjunctive clause ($\wedge_{i=1}^{m}(C_{i,j})$), b_{tj} is calculated by applying Tracey et al. [26]'s functions on each individual condition (C_i) in the sub-constraint and then summing them all. Finally, we need to normalize the value between 0 and 1.

Thus the final fitness of an individual is the sum of branch distance and approximation level.

$$F = B_t + AL$$

where, F is the fitness of an individual. AL is the approximation level and B_t is the normalized branch distance at transition t.

Since in our case we typically have multiple targets (several MC/DC combinations to cover), we have also adopted a whole test suite generation approach from [10]. The idea is that while evaluating the fitness of an individual for a target, we not only evaluate that particular target but also all the other targets. This approach is efficient since an individual might be fit for more than one target.

4.5 State machine under test

As part of our collaboration with MicroPilot, the first author of the paper together with a domain expert from the company developed specification models for the main scenario of autopilot (the flight command behaviour). The team is now in the process of building more models. Our state machine for this study (Fig. 1), which is anonymized for the confidentiality reasons, contains 7 test paths shown in table 2 including 12 states, 16 transitions and total 384 MC/DC combinations. Each of those combinations is an abstract test cases and a target for our search-based test data generator.

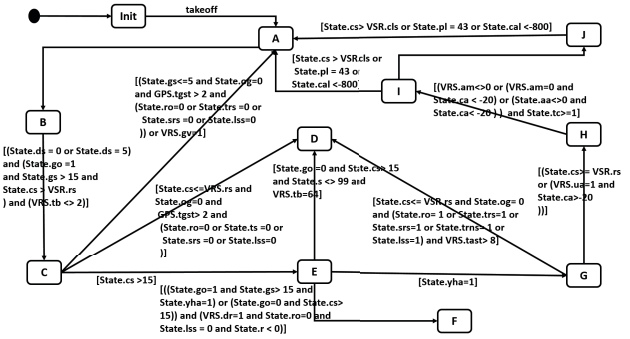

Figure 1: A sample anonymized state machine of the system under test.

Table 2: Test paths of the sample state machine in Figure 1.

SL	Path	No. of test cases using	
		Transition Coverage	Proposed Approach
1	init-A-B-C-D	1	34
2	init-A-B-C-A	1	33
3	init-A-B-C-E-F	1	15
4	init-A-B-C-E-D	1	7
5	init-A-B-C-E-G-D	1	51
6	init-A-B-C-E-G-H-I-J-A	1	122
7	init-A-B-C-E-G-H-I-A	1	122
	Total	**7**	**384**

4.6 Execution setup

We have applied (1+1) EA to generate test data described in section 4.4.1 for a sample anonymized state machine of the system under test shown in Fig. 1. We run (1+1) EA and random search in a highly configured development machine (Intel Core i7 CPU 3.40 GHz 16 GB RAM 64bit OS x64-based processor).

5 RESULTS AND ANALYSIS

In this section, we answer all three research questions asked in section 4.3 and discuss about the limitations of our approach and challenges of applying this technique in practice.

5.1 RQ 1. How much does the proposed approach improve model-level constraint coverage?

To answer this question, we first applied a (1+1)EA to cover model-level MC/DC and compare the model-level coverage with the default "all transition coverage". The results show that our approach (the MC/DC test suite) has generated test cases for 250 targets out of 384 targets. Therefore, it has 65.1% MC/DC coverage of model constraints as shown in Fig. 2. On the other hand, the traditional "all

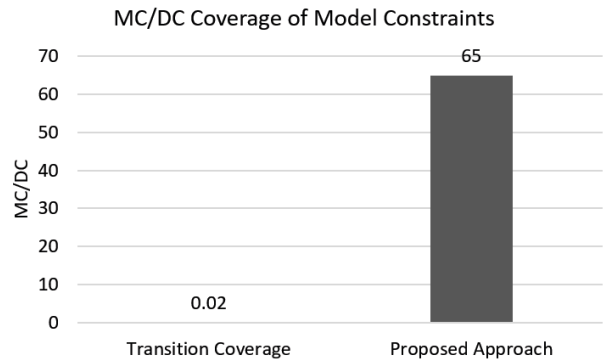

Figure 2: Improved model constraint coverage

transition" coverage generates 7 test cases for the entire state machine and the MC/DC coverage is 0.02%. So, as expected the answer is YES! the proposed targeted approach significantly improves the model constraint coverage.

However, almost 35% of the MC/DC constraints are not covered, by our approach. So in the rest of this RQ we investigate these constraints in more detail. We have analyzed all constraints and categorized them based on the reason for not being covered. The categories are described as follows:

(1) **Satisfied:** The search algorithm successfully managed to find a solution regarding such target.
(2) **Not Satisfied:** There are conditions that cannot be satisfied due to the existence of bugs or limitations. We have further categorized these types of conditions as follows.
   I **Buggy:** Conditions that can not be satisfied due to a bug in the code. The proper data has been generated and the test case has been executed but the test FAILs.
   II **Infeasible:** It is not possible to satisfy such target at all due to conditions that can never be true. Infeasible targets are typically due to contradictory conditions. But in our case all our infeasible targets were due to legacy code that no longer exists. So the feature had been removed but not completely. The pieces that were left in the code were e.g., extra conditions in the code that would not have any effect on behaviour but are not desired because they reduce readability of the code and perhaps create confusion.
   III **Limitation of the simulator:** There are several sensor fields that cannot be simulated properly using the simulator. It is possible that a solution for a constraint is feasible but the search could not generate it because of the simulation's limitation on manipulating a specific sensor data.
   IV **Limitation of our MBT Tool:** There are some conditions that cannot be satisfied because of our current MBT approach, where we, like most other MBT studies, generate all test data in the beginning of each test and do not change them during execution. However, sensor fields in our case study can be potentially changed anytime during the execution. Therefore, if there exist contradictory conditions on a path that depends on a sensor field having different values during one execution, our tool fails to satisfy those paths. A proper way of fixing this issue would be modelling the environment (including sensor fields) separately and have the two models (SUT vs. environment) run concurrently, which is in our future work.
   V **Limitation of the search algorithm and budget:** There is a feasible solution but the particular search algorithm fails to find it with the given budget. For example, a solution can be found using the evolutionary algorithm but not using random search.

To have a closer look at the effect of the search algorithm on model coverage, we compare (1+1)EA's results with a baseline Random search algorithm. As discussed earlier in Section 4.4.1, each fitness evaluation involves integrating the test execution framework and running a test case on the simulator. Due to the cost of each evaluation, we set a termination criterion of maximum 200 fitness evaluations per target. Given this limit, (1+1)EA needed 26,210 fitness evaluations to try all targets. Note that not all targets require maxing out the 200 evaluations. Also, some evaluations may result in a solution for other targets than the one under test, due to the whole test suite generation approach explained in Section 4.4.2.

To compare (1+1)EA with the baseline, we gave the same fitness evaluation budget of 26,210 simulator executions to the Random Search. Note that the clock hour execution time for the two algorithms were slightly different, though. It took (1+1)EA 145 hours to finish the 26,210 executions, whereas random search spent 149 hours to finish the runs.

This is simply because the two techniques don not necessarily try the same solutions and thus the run time per test evaluation may be varied. In particular, in the search space, some invalid input combinations may cause execution errors and thus more expensive runs (to recover the system from the error). In (1+1)EA any such individual will get a high fitness value (=100) thus helping the search to avoid generating similar erroneous individuals. However, since the random search is not guided, it might produce more such erroneous individuals, thus more total execution time.

Table 3 and 4 show the results of (1+1) EA and random search, respectively.

The effectiveness of these two algorithms are the same except the paths 5, 6, and 7. With the same amount of budget, (1+1)EA has finished trying all the targets but the random search has 7 unattempted targets in total. In paths 6 and 7 the targets that are not attempted by random search are identified either as infeasible paths or unsatisfied due to the simulator limitations, by (1+1)EA. Similarly, in path 5, two unattempted targets are categorized as infeasible, by (1+1)EA. However, there is one target that is satisfied by (1+1)EA but not attempted by random search.

In summary, (1+1)EA has attempted covering seven more targets than random search. It also satisfied one more target (250 vs. 249 targets out of 384). The difference between random search and (1+1)EA suggests that a more effective search strategy with potentially more budget may be able to improve (1+1)EA results, in terms of model constraint coverage, as well.

Table 3: Result of (1+1) Evolutionary Algorithm, with 26,210 fitness evaluations budget.

Path#	Targets	Bug	Inf.	Limitation of		Unattempted by EA	Satisfied
				MBT Tool	Sim.		
1	34	5	4	4	8	0	13
2	33	4	4	4	8	0	13
3	15	0	2	6	3	0	4
4	7	0	1	2	1	0	3
5	51	1	5	5	17	0	23
6	122	0	13	0	12	0	97
7	122	0	13	0	12	0	97
Total	384	3(unique)	2 (unique)	21	61	0	250

Table 4: Result of Random search, with 26,210 fitness evaluation budget.

Path#	Targets	Bug	Inf.	Limitation of		Unattempted by Ran.	Satisfied
				MBT Tool	Sim.		
1	34	5	4	4	8	0	13
2	33	4	4	4	8	0	13
3	15	0	2	6	3	0	4
4	7	0	1	2	1	0	3
5	51	1	3	5	17	**3**	22
6	122	0	12	0	12	**1**	97
7	122	0	12	0	10	**3**	97
Total	384	3(unique)	2(unique)	21	59	**7**	249

Table 5: Code coverage of the "all transitions" coverage vs. our proposed model-level "MC/DC" coverage.

Source files	Function Coverage					Condition/Decision Coverage				
	Total	All Transitions		MC/DC		Total	All Transitions		MC/DC	
		#	%	#	%		#	%	#	%
D1	29	0	0%	0	0%	42	0	0%	0	0%
D2	27	0	0%	0	0%	0	0		0	0%
D3	20	0	0%	0	0%	80	0	0%	0	0%
D4	15	0	0%	2	13%	58	0	0%	0	0%
D5	9	0	0%	0	0%	272	0	0%	0	0%
D6	9	0	0%	0	0%	203	0	0%	0	0%
D7	621	1	0%	1	0%	4066	0	0%	0	0%
D8	20	1	5%	1	5%	192	1	0%	1	0%
D9	24	2	8%	2	8%	193	0	0%	0	0%
D10	567	56	9%	75	13%	1992	137	6%	163	8%
D11	418	53	12%	61	14%	6019	535	8%	3168	52%
D12	966	198	20%	211	21%	10022	1071	10%	1143	11%
D13	2437	755	30%	779	31%	41888	10088	24%	11602	27%
D14	2568	820	31%	844	32%	44564	10690	23%	12285	27%
D15	324	168	51%	173	53%	3645	773	21%	873	23%
D16	113	65	57%	65	57%	2201	602	27%	683	31%
D17	271	166	61%	171	63%	3410	773	22%	873	25%
Total	4843	1295	**26%**	1364	**28%**	66242	13206	**19%**	17632	**26%**

5.2 RQ 2. Does the proposed approach improve code coverage?

As mentioned in Table 2, the sample state machine contains 7 test paths, which are covered by 7 test cases using "all transition" coverage. Our approach creates 384 target test cases for these 7 paths and manages to generate concrete test cases for 250 of those targets. We measure the code coverage of the original 7 test cases (generated by "all transition" coverage) as well as 250 test cases generated by our proposed approach (applying model-level MC/DC). To calculate the

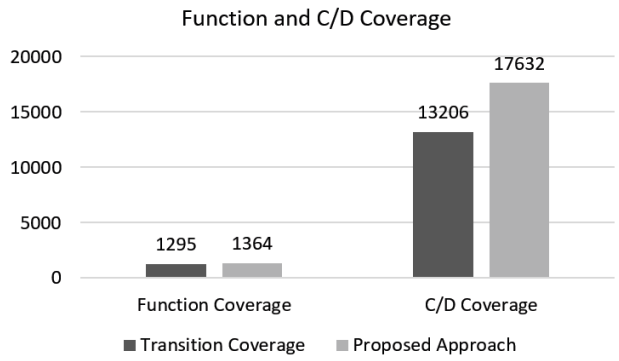

Figure 3: Improved code coverage

code coverage, we used the company's coverage tool called "Bullseye Coverage" [6] tool. The Bullseye Coverage [6] measures source code coverage in terms of "function coverage" and C/D coverage. Ideally, measuring code-level MC/DC would be more interesting since the standard also asks for it. However, C/D coverage gives a more detailed level of coverage (all conditions and decisions coverage not just affecting ones as in MC/DC). In addition, finding a free coverage tool that measures MC/DC for C code was not trivial either. So we opt for using the company's existing tool.

Figure 3 summarizes the function and C/D coverage for our sample state machine. Using the MC/DC model coverage rather than "all transitions", the function coverage went up from 1,295 functions to 1,364 functions. That is an extra 69 (10%) functions. In terms of C/D coverage MC/DC test cases covered 17,632 cases whereas "all transitions" test cases covered 13,206 combinations. That is an extra 4,426 (33.5%) combinations of conditions. Note that this impressive improvements were resulted by only modelling one flight command (i.e., $takeoff()$) of the system. This definitely limits the scope of improvement in overall function and C/D coverage.

Table 5 summarizes both function and C/D coverages for the default ("all transitions") and improved ("MC/DC") coverage, across all files reported by BullseyeCoverage tool. In the overall set up, there are 4,843 functions and 66,242 C/Ds to cover and even our improved test suite covers only 28% and 26% of the functions and C/Ds. Therefore, we expect that a full test suite generated from a complete model of all flight commands will cover more functions and C/Ds and bring even more significant improvements.

5.3 RQ 3. Does the proposed approach detect new bugs?

Though code and model coverage are important from the certification point of view, but without a doubt the most important quality factor of a test suite is its ability to reveal undetected bugs.

In MBT a bug is detected when the behaviour specified in the state machine does not match with the actual behaviour of the running system. In this RQ we compare the two model-level criteria's effectiveness in terms of detecting new bugs. By a new bug we refer to a behaviour which was not expected by the domain expert and was unknown before running the test suite. That means that the company's own unit and system-level testing could not catch that defect.

Our results show that the transition coverage approach did not detect any new bug, but the proposed MC/DC approach has detected 3 new bugs and 2 wrong legacy specification instances. To be more precise, the tool has found one fault regarding autopilot configuration, and two faults regarding incorrect code implementation. Note that in Tables 3 and 4 the detected unwanted legacy codes are not categories as bugs but they are in the "Infeasible" category. These cases are typically conditions in the code related to an old feature which is forgotten to be removed, when the feature's code is deleted. So we do not call them defects since one can not cover the condition and get a failed test (infeasible path), but it is still an unwanted code that reduces code readability.

Below is a high-level analysis of the detected bugs (the details are omitted due to confidentiality reasons):

- **Bug 1: Configuration:** The system under test, autopilot software, can be embedded in different types of UAV hardware (vehicle) such as Fixed-wing, Helicopter, multi-rotor, etc. However, there are some configurations that are specific to the vehicle type. Those configurations are specified using configuration (VRS) fields. These configuration fields are also exposed to the users. A wrong configuration will result in a bug. To be exact, in our sample state machine the condition $VRS.gv = 1$ is wrong.
- **Bug 2: Incorrect transition constraints:** In our case, the conditions from State C to A and C to D are wrong in the code ($State.tgst > 0$ instead of $State.tgst > 2$ as it is specified in the model).
- **Bug 3: Incorrect transition constraints:** Another bug of the same type (incorrect transition constraints). The condition from State G to D is wrong. The code implements $State.tast > 0$ instead of $State.tast > 8$.

Therefore to summarize RQ3, although, we have used only one sample state machine to evaluate our approach, as the results show, the technique was effective to detect new bugs. This approach also detects wrong specifications which is helpful to detect and remove legacy code that are obsolete.

5.4 Discussion

In this section, the main challenges that we have faced during this project will be explained:

Reducing the search space: As discussed, our search space is very large with thousands of configuration and hundreds of sensor variables. Searching the entire search space simply was not an option in our case.

The ideal way to reduce the search space is by knowing the exact set of internal and external variables that may affect the transitions in the test path. A transition may be affected by "Command Input Parameters" and/or "State Fields" (in the guard). Limiting the search space to only input fields that can potentially modify the "Command Input Parameters" and "State Fields" is really useful.

Therefore, one need to do a backward dependency analysis to identify influential autopilot settings and sensor inputs, to keep for each target test case. In this study, using the domain expert knowledge, we manually excluded the obvious irrelevant cases.

However, there might be many more cases that could have been excluded to further reduce the search space.

To improve the results, we tried to do a one-time static analysis on the source code. A static analysis strategy that applies in such scenario is called program slicing [29]. Static program slicing is the extraction of a set of program statements (known as a slice) that may affect the values at some point of interest based on a slicing criterion. Since, the system under test is developed using C language, we looked for existing program slicing tool for C/

The Wisconsin Program Slicing Tool [14] is a commercial tool for slicing C programs. This tool is no longer being distributed and have several limitations. For example, it does not support *struct* typed identifiers, variable length parameter list, signals, and system calls.

The Unravel [19] is an open-source program slicing tool for C that runs under UNIX environment. As pointed by [18], this tool does not support several recent extensions of C language such as *new*, $sizeof$ and *delete* keywords, C templates and fixed-width integers (e.g., $int32\_t$, $in64\_t$).

Frama-C is an up-to-date program slicing tool for C language [16]. It provides a collection of plug-ins that perform static analysis, deductive verification, and testing; and it has been previously applied in safety critical software, as well. Frama-C is based on C Intermediate Language(CIL) [22], which is a high-level representation of C program and includes a set of tools to do easy analysis and source-to-source transformation of a C program.

However, CIL is also unable to comprehend some extension of C language that are widely used in our system under test. For example, CIL does not recognize fixed-width integer types such as $int8\_t$, $int16\_t$, $int32\_t$, $int64\_t$, $uint8\_t$, $uint16\_t$, $uint32\_t$ and $uint64\_t$. In MicroPilot autopilot, there is no generic integer type. Instead all integers are fixed-width integer types. The size of a generic integer type (*int*) depends on the memory and system architecture but the size of a fixed-width integer is always fixed irrespective of memory and architecture. Its not an issue in modern computers, but in an embedded system memory architecture, it is particularly important that a variable will always have the same predefined size. Therefore employing Frama-C requires a lot of modifications in our case.

In the future, we are planning to investigate more dependency analysis approaches such as CodeSurfer [8] to automate this step.

Reducing execution time: In the simulator one hour of real flight takes several minutes based on the computer configuration and the hardware type of the UAV. In our case, we ran the simulator on a highly configured development machine (Intel Corei7 CPU 3.40 GHz 16 GM RAM 64bit OS x64-based processor) where each test execution would take about 500 seconds with a minimum amount of flight plan (i.e., $takeoff$ and *circuit*). The long flight simulation is because before flying a UAV, the landing procedure needs to be defined unless the flight commands are given from the Ground Control System (GCS). This is true even in the case of simulation. Such a time consuming execution imposes a great cost of fitness evaluation in search-based testing.

To reduce test execution time we use a feature of simulator that let us monitor what flight command has been executed in the autopilot at a certain time. We use this opportunity and modify the test code to stop the simulation whenever it finishes executing the flight command of interest. Therefore, if for instance I am testing the behavior of the $takeoff$ procedure, I do not need to wait for the execution of the landing procedure. Applying this idea we managed to reduce the average test execution time from 500 seconds to about 23 seconds, with a minimum amount of flight plan when testing $takeoff$ flight command. Though 23 seconds is way better than the original 500 seconds, it still is an expense that we would like to reduce in future so that more expensive search algorithms can be used for test generation.

5.5 Threats to the Validity

The main threat to the validity of this study is the limited sample size (the specification model for only one flight command). We are working with MicroPilot to build more specification models for other flight aspects to be able to generalize our findings to the entire autopilot system. However, the goal of this study is by no mean over-generalizing the results to other systems and domains. That requires replicating the study. However, we have provided enough information about the details of our experiment so that other researchers can replicate the study.

We have also implemented a fair amount of code to build the MBT tool and generate test data. All these implementations are subject to errors, which may affect the results.

Our search-based approach is very simple (due to the large search-space and expensive test executions). This may underestimate the effectiveness of MC/DC-based approach on code coverage and fault detection, if a more advanced technique is afforded.

Our fault detection analysis is only based on the new faults, however, a more general study would consider any faults (e.g., reintroducing historical defects, or mutation analysis). Our code coverage analysis is also limited to the company's current code coverage tool. Therefore, we reported function coverage and all C/D coverage and not e.g., statement , branch , or MC/DC coverage criteria.

6 RELATED WORK

Kicillof et al. [15] at Microsoft introduced a novel approach to automate low-level test generation by combining model-based testing with white-box parameterized unit testing. The authors uses Pex [25], a symbolic execution tool, also developed at Microsoft, to extract concrete parameter values to improve branch coverage at the code-level. The parameter values are then fed back to the model-based testing tool [24] which generates multiple test cases for each methods with a high branch coverage.

Vishal et al. [27] studied a Constraint-Based Testing (CBT) approach to improve code coverage using SpecExplorer [24]. Constraint based testing is a promising technique in white-box testing that uses constraint solvers to generate test data. SpecExplorer [24] is a white-box model-based testing tool for C-Sharp that uses Z3 SMT Solver [9] to get decision tables. Z3 provides a decision table that includes the code-level branches and appropriate input data, generated by solving the code-level constraints that comes from symbolic execution tool Pex [25]. The decision table is rich in branch coverage, which in turn defines the data model to generate test cases. However, the results show that their approach does not

improve the code and decision coverage but it improves condition and boundary coverage significantly.

Godboley et. al. [11] applied a program transformation technique to improve the MC/DC coverage criterion. Typically Concolic execution does not focus on MC/DC coverage, instead it focuses on branch coverage. The authors' idea is to transform conditional structures with multiple conditions into nested conditional structures with single condition. So that, the Concolic execution can be applied to generate test data covering all the branches of the code. They used Quine-Mc-Mlusky technique to transform conditional structures with multiple conditions to generate empty $if - else$ statements and applied Concolic execution tool, CREST [7], to generate the test data.

Li et. al. [17] conducted an empirical experiment to show the effectiveness of combinatorial testing to improve MC/DC criterion of a test suite. In combinatorial testing, the interaction strength represents the number of input parameters that are considered for the combination. Typically, higher interaction strength often leads to high MC/DC coverage. Higher the interaction strength, the more expensive the test suite is. However, the experiment results show that 5-way test sets are better than 4-way, 3-way, and 2-way but not significantly better. A five-way test set achieves certain MC/DC coverage more slowly than that of 4-way, 3-way, and 2-way. On the other hand, 2-way test set are less expensive but they are not effective to achieve high MC/DC coverage. The authors recommended to use 4-way or 3-way test sets, in practice.

7 CONCLUSION AND FUTURE WORK

Safety-critical software systems such as UAV autopilots are overly complex. Testing the behavior of an autopilot system plays a crucial role in the system's safety and obtaining airworthiness certifications. Safety certifications, such as DO-178C [23], recommend following a model-driven approach for testing high-level requirements of the system. The standard also demands targeting low-level code coverage. In this study, we explored the possibility of improving Model-based Testing's (MBT) low-level code coverage, through replacing MBT's default "all transition coverage" with a more demanding logic coverage (MC/DC) on the model constraint-level.

Our experiment on a representative state machine from our industry collaborator's autopilot software shows that the MC/DC-based approach can significantly improve the code coverage. In addition, the model-level MC/DC test suite detected three new faults and two obsolete legacy code compared to the "all transition" test suite.

In the future, we are planning to improve this approach by a more-cost effective test generation technique. In particular, we are working on reducing the search space by an automated dependency analysis. Together with the company experts, we are also preparing more specification models of different flight commands, to generalize our findings to the entire autopilot system.

ACKNOWLEDGMENT

Part of this work was done when the first two authors were employed by the University of Manitoba. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2011. DO-331, Model-Based Development and Verification Supplement to DO-178C and DO-278A. *RTCA Inc.* (2011).
[2] AFC 2009. Automated Flight Control. In *Advanced Avionics Handbook*. Federal Aviation Administration, Chapter 4.
[3] S. Ali, 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. *IEEE Transactions on Software Engineering* 36, 6 (Nov 2010), 742–762.
[4] S. Ali, M. Zohaib Iqbal, A. Arcuri, and L. C Briand. 2013. Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1376–1402.
[5] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and Jānis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. 263–272.
[6] Bullseye Coverage [n. d.]. Bullseye Coverage. Bullseye Coverage Technology. ([n. d.]). http://www.bullseye.com
[7] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. 443–446.
[8] CodeSurfer Tool [n. d.]. ([n. d.]). https://www.grammatech.com
[9] L De Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.
[10] G. Fraser and A. Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.
[11] S. Godboley, G. S. Prashanth, D. P. Mohapatro, and B. Majhi. 2013. Increase in Modified Condition/Decision Coverage using program code transformer. In *2013 3rd IEEE International Advance Computing Conference (IACC)*. 1400–1407.
[12] M. Harman and B. F. Jones. 2001. Search-based software engineering. *Information and Software Technology* 43, 14 (2001), 833–839.
[13] H. Hemmati, A. Arcuri, and L. Briand. 2013. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 6.
[14] S. Horwitz, T. Reps, and D. Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.
[15] N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman. 2007. Achieving Both Model and Code Coverage with Automated Gray-box Testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*. 1–11.
[16] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Form. Asp. Comput.* 27, 3 (May 2015), 573–609.
[17] D. Li, L. Hu, R. Gao, W. E. Wong, D. R. Kuhn, and R. N. Kacker. 2017. Improving MC/DC and Fault Detection Strength Using Combinatorial Testing. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 297–303.
[18] J. R. Lyle and D. R. Wallace. 1997. Using the Unravel Program Slicing Tool to Evaluate High Integrity Software. In *In Proceedings of 10th International Software Quality Week*.
[19] J. R Lyle, D. R Wallace, J. R Graham, K. B Gallagher, J. P Poole, and D. W Binkley. 1995. Unravel: A case tool to assist evaluation of high integrity software volume 1: Requirements and design. *National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD* 20899 (1995).
[20] Micropilot Inc. [n. d.]. Micropilot Inc.. ([n. d.]). http://micropilot.com
[21] S. Nair, J. L. De La Vara, M. Sabetzadeh, and L. Briand. 2014. An extended systematic literature review on provision of evidence for safety certification. *Information and Software Technology* 56, 7 (2014), 689–717.
[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. 213–228.
[23] 2011. DO-178c Software Considerations in Airborne Systems and Equipment Certification. *RTCA Inc.* (2011).
[24] SpecExplorer [n. d.]. SpecExplorer. Microsoft Inc.. ([n. d.]). https://msdn.microsoft.com/en-us/library/ee620411.aspx
[25] N. Tillmann and J. De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs*. 134–153.
[26] N. J. Tracey. 2000. *A search-based automated test-data generation framework for safety-critical software*. Ph.D. Dissertation. Citeseer.
[27] V. Vishal, M. Kovacioglu, R. Kherazi, and M. R. Mousavi. 2012. Integrating Model-Based and Constraint-Based Testing Using SpecExplorer. In *IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. 219–224.
[28] J. Wegener, A. Baresel, and H. Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.
[29] M. Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. 439–449.