Model-Based Testing of Safety-Critical Avionics Systems

by

Syed Samsul Arefin

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2017

Thesis advisors

Hadi Hemmati, Rasit Eskicioglu

Author

Syed Samsul Arefin

Model-Based Testing of Safety-Critical Avionics Systems

Abstract

Safety-critical systems in aviation domain often go through the formal process of airworthiness certification DO-178C, which ensures system's safe and risk-free operation. Two of the main requirements for this certification are to have traceability between specifications and test cases, and have high code coverage. In this thesis, I have implemented a Model-Based Testing (MBT) tool to automate system-level test generation from specification models. MBT alone provides the traceability requirement. To improve the code coverage, unlike transition coverage based MBT, I target model-constraints in a more demanding manner, Modified Condition/Decision Coverage (MC/DC), and generate an extra set of test cases using an evolutionary algorithm that leads to higher code coverage. The result shows, the proposed approach improves the MC/DC model-constraint coverage by 65% and the code-level C/D coverage by 33% for a sample specification model. The proposed approach also detected three new faults in the system.

Contents

List of Figures

List of Tables

Acknowledgments

I thank my advisors, MicroPilot, my thesis committee members, my parents and all the well-wishers who supported me along the way.

To world peace.

Chapter 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 that is usually flown by a pilot at a ground control station. A UAV can fly autonomously based on predefined flight plans using software called Autopilot. Autopilot helps automate the process of controlling and guiding the aircraft [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. To prevent catastrophic effects, the regulatory authorities and the avionics industry have defined DO-178C [3], a rigorous certification standard for avionics systems. The 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, plus a mapping between each requirement and its test cases to provide a two-way traceability between the requirements and the test cases. The test cases are also required to provide high code coverage in terms of Modified Condition and Decision (MC/DC) coverage [3].

Software testing includes test plans, testing strategy, and specifications of the software to detect possible defects of the system. Software needs to be tested continuously during each stage of the development process. The cost of software testing is significant in terms of time and other resources [4]. It is done in several levels that are run in parallel with the development process. The major levels of testing are:

1. Unit testing
2. Integration testing
3. System-level testing
4. Acceptance testing

The testing of software is almost completely separated from the software development process. Variety of tools and languages are used in different activities of the development process. Even within the software testing, different sets of expertise is required for every testing level [5]. In the unit and integration-level testing, variety of existing tools are used to automate the test generation process based on the used language. For example, LDRA [6] and CTGEN [7] are widely used in generating unit tests for C/C++ language. On the other hand, system-level testing is the most misunderstood and most difficult testing process that tests the complete system as a whole [5]. The

system-level test particularly compares the system's behavior to its original objectives while keeping the system as a black box. In system-level testing, test scripts are specific to the application and are usually done manually. In avionics systems, aircraft test emulators [8] enable system-level test cases to run in a controlled software environment with less cost instead of running it in the real environment. The test goals such as high coverage and traceability are also applicable in system-level testing.

To partially achieve this goal from system-level testing, this thesis focuses on the requirement-to-test mapping (one-way tractability) and apply a black-box model-based testing technique on specification models written by the domain experts. I have introduced a new technique in the model-based testing tool that focuses on testing each condition in the model constrains and generates a huge set of test cases unlike traditional model-based testing. Our approach improves the test suite in terms of their code coverage. To generate test input data, I use evolutionary algorithm and compare the result with a random test data generation strategy.

Model-based Testing (MBT) is a model-driven technique for test automation, which uses specification models of the System Under Test (SUT) to generate test cases. Specification models represent the expected behavior of the SUT and can be seen as the requirements of the system. Specification models are usually written in Unified Modeling Language (UML) behavioral diagrams. Some example of behavioral models are Finite State Machines (FSM) diagram, Sequence Diagrams and Activity diagrams. A testing strategy is then applied on those models to create abstract test cases. Finally, an input test data generation technique such as search-based testing

or symbolic execution is applied in the abstract test cases to make them concrete and executable.

In this thesis, I have worked with the industry partner, MicroPilot [9], a word-leader in building autopilot systems for commercial Unmanned Aerial Vehicles (UAV). I have access to their autopilot embedded system code-base and their domain expertise. Our overall goal is to apply cutting-edge test automation techniques on their autopilot system and empirically investigate their effectiveness in detecting bugs and analyzing pros and cons. The proposed tools and processes of this project will be integrated into the existing testing frameworks and processes of the company to help them to get airworthiness certification.

1.1 Research Problem

Due to the complexity of the system under test and the requirements of the certification (e.g., high code coverage, traceability between the requirements and test cases), developing system-level test cases is a time consuming and difficult task. A specification model of the system represents its behaviors, which can be used to automatically generate system-level behavioral test cases and map them to the requirements on the model level. In this thesis, I propose to systematically generate system-level test cases from the UML state machine model of the system under test. The challenges are:

1. Technical challenges: Although the autopilot system at MicroPilot is built using the C language but it has a language independent representation of system-level test cases which run on the windows-based simulator. Generating system-level test cases requires seamless integration of the simulator and the test execution

framework. Therefore, I need to develop a tool integrating their simulator and the test execution framework to automate the test generation process.

2. Following the standard: I need a high coverage of the model-level requirements. This means that the test data generator needs to solve the conditions on the paths and find the input data that leads the system to cover those paths. It is also expected that high model-level requirement coverage may lead to high code coverage.

3. Test data generation: Given that an avionic system is complex and a very wide range of system parameters affect the behavior of the system, finding the appropriate test data for specific test path would be challenging and costly in terms of time and computational resources.

4. Simulating sensor inputs: The behavior of an avionic system is extensively affected by the environmental parameters, i.e. sensor inputs. Most of the safety-critical features come into action based on those sensor inputs. The problem is not only finding the specific sensor inputs for a test path, but also finding the time when the sensor inputs need to be simulated during a test flight.

5. Execution expenses: System-level testing includes running test flights under the simulator. A test flight includes takeoff and landing procedures and a set of flight command schedules within them. In the real environment the entire procedure may take an hour or more depending on the flight commands. However, in the simulator this time is reduced to a few minutes since it does not depend on the hardware of the UAV. It is still very expensive in terms of time

if it needs hundreds or thousands of try to find a proper set of input values for a specific test path.

6. Generating test code: The system-level test cases have a language independent representation. Therefore, the generated test cases need a transformation to a data structure specific to the system under test and it's test execution framework. Model-to-test (M2T) transformation are well established. There are several well established tools that transform a test into a programming language syntax. For example, [10] provides a necessary code generation framework and facilities that transform test cases into java, C/C++, HTML, properties or XML files. However, the system-level test cases has its own language independent representation, hence, I have to create my own transformation logic. Consequently, any limitation of test execution framework or the simulator or the system-level test representation would certainly affect capability of the proposed tool as well.

Therefore, the objective of this thesis is developing a model-based testing tool that accepts UML state machine models of the autopilot system and efficiently generates system-level behavioral test cases with high coverage as well as detect real faults in the system. I also conduct an empirical study to evaluate the effectiveness of the tool in terms of fault detection.

1.2 Research Methodologies

My research methodology includes both exploratory and analytical aspects. The research work is started with studying the autopilot system and current challenges to qualify for the airworthiness certifications. I then focused on automating system level test automation process while emphasizing the certification requirements, for example improving coverage, traceability and comprehensive testing. I choose to apply the model-based testing techniques so that it enriches the software specifications in the first place and automates the test generation process. It also enriches the traceability between the specifications and test cases. In the beginning of this thesis, I have comprehensively studied the system to do system modeling and write specification models. Those specification models are used as the input of the model-based testing process. The system modelling has been done using Papyrus [11], an open-source modeling tool that runs in the Eclipse environment. I have developed the model-based testing tool using the Eclipse Modeling Framework (EMF) [12].

To generate test input data to improve the structural coverage, I have used search algorithms (e.g., evolutionary algorithm). To reduce the cost of applying search algorithms, I have optimized the test generation and execution time by analyzing the source code and other related system aspects. To demonstrate the efficiency and correctness of the tool I have modeled new features of the system under test in a state machine model and used that model to generate system-level test cases. The objective is to see if the tool can detect any real defects in the system.

1.3 Research Settings

The research presented in this thesis has been done in collaboration with MicroPilot Inc. [9]. The collaborative research work was in the context of a bigger project to help MicroPilot get software certification. This thesis, specifically focuses on applying Model-Based testing in avionic industry. The objective is to automate the system level test generation process and to increase code coverage in order to earn air-worthiness certification DO-178C [3]. This project was funded by Natural Sciences and Engineering Research Council (NSERC), Canada.

1.4 Contributions

In this thesis, I have developed a Model-Based Testing (MBT) tool for an avionic system that can generate system-level test cases from the specification model. The tool enables test automation while improving traceability and code coverage and allows comprehensive testing of the system under test. Some of the contributions of this thesis towards the software engineering research and the industry are as follows:

1. **Introducing MBT in A Avionics industry.** The model-based testing tool I have implemented is not just a prototype, rather it is a functional tool that can be used in Micropilot's development environment. Applying the state-of-the-art research in industry is also as important as bringing innovation. I have also introduced a new approach in MBT that allows comprehensive testing of the system under test. Industries need to know that the state-of-the-art research in software engineering is applicable and works on industry code-base as well.

2. **Certification.** The first advantage of introducing MBT in MicroPilot [9] is to provide support for airworthiness certification. The proposed model-based testing tool improves requirements-to-test traceability and improves code coverage which are supporting facts for the certification.

3. **Improving Documentations.** Having the specification model is the prerequisite of being able to use an MBT tool. Specification models improve the documentations and traceability and increase the understandability of the system. Consequently, they reduce the maintenance cost of the system, as well.

4. **Test Automation.** Having specification models means having test cases, if there is an MBT tool. Software testers do not need to write new tests for new features. Instead, adding the new functionalities to the specification models enables the tool to generate tests for the new features. Thus, it makes easy to maintain test cases and saves a great deal of time in the development environment.

5. **Improving Constraint and Code Coverage.** I have also introduced a new approach in the model-based testing process that focuses on the model constraint coverage. To be more specific, my approach maximizes the MC/DC coverage (described in section 7.2) of the model constraints which consequently improves the code coverage.

6. **Bug Detection.** Model-based software testing allows early validation of requirements by detecting bugs as soon as possible in the product life-cycle. Using this tool, I have detected some unknown defects in the system and legacy spec-

ifications that are no longer exist in the code.

7. **Communication.** There is always a communication gap between safety engineers, software developers and software testers. Moreover, safety engineers need to communicate with certification authorities. So safety engineers must get insight into the software and it's safety compliance aspects. But they are not likely to be experienced in software engineering, which makes their task challenging. Such communication gaps are reduced by introducing UML system modeling.

1.5 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2, I briefly describe the Model-Based Testing (MBT) process including model coverage criteria, test data generation techniques such as search algorithms.I also describe the well-established branch distance calculation techniques used in fitness function to guide search algorithms. In Chapter 3, I describe the de-facto standard software modeling language-Unified Modeling Language (UML), and it's Extended Finite State machine (EFSM) used to model system specifications. The next is Chapter 4, where I introduce the System Under Test (SUT) and some related specification models. In Chapter 6, I present the process of developing a Model-Based Testing tool and frameworks and libraries used. In Chapter 7, I describe our new approach that emphasize testing each condition in the model and generate more test cases than the traditional MBT approach that improves the code coverage and bug detection capability. We describe

the recommended code coverage criterion for avionics systems and the application of evolutionary algorithm to generate test data that covers the uncovered areas of the code. In section 8.1, I present the effectiveness of the two search algorithm- 1+1 evolutionary algorithm and random search in generating test input data. In section 8.2, I present the effectiveness of our tool using a quantitative study on detecting bugs and improving coverage; and the lesson I learned while applying MBT in the avionic domain. In Chapter 9, I summarize our contribution and discuss some potential future works.

Chapter 2

Model-Based Testing (MBT)

Model-based Testing (MBT) is an efficient automated test generation process [13]. This technique uses the specification model of the system that models the requirements and functionalities as a basis to generate test cases. It is an application of model-based system design.

2.1 Model-Based Testing (MBT) Process

Model-Based Testing (MBT) is a process of automating test generation from a specification model. The specification model does not necessarily need to be a formal specification. In fact it is normally based on the requirements. The generic process of Model-based testing is briefly described in this section. As shown in figure 2.1 the entire process can be divided into four major steps.

1. The very first step of the Model-Based Testing is the modeling phase. In this step, the test designer manually models the system under test. While modeling

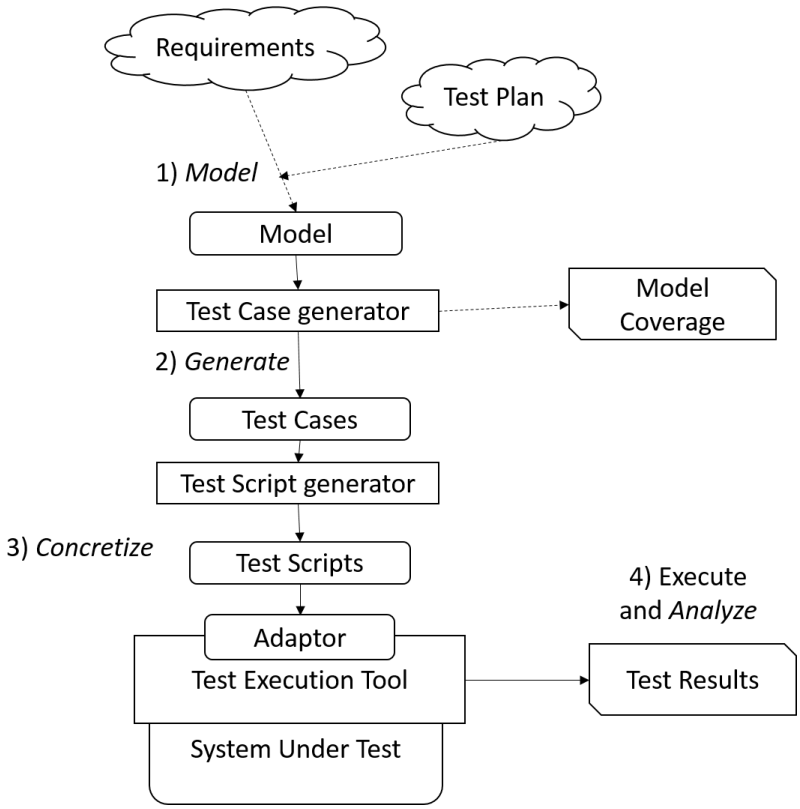

Figure 2.1: Steps of Model-Based Testing (MBT) Technique

the system, the designers may focus only on specific parts e.g., components, classes, features, etc. of the system or specific aspects e.g., functionality, security, performance, etc. of the system, they are interested to test. The narrow focus typically increases the scalability of the technique. The model is not only going to be used in the test generation process, but also serves as the requirement of the system. Since, the model is going to be used to generate tests, the modeling phase also involves "test planning" to make sure all the necessary requirements and specifications have been considered.

2. The second step involves generating abstract test cases from the model. In this phase, the model, typically, is converted to a directed graph. Then a model

coverage strategy, for example structural coverage is applied to that graph with the help of graph traversal algorithm (breadth first search or depth first search) to create a set of test paths from the initial state to a leaf state. The model coverage criteria are described in section 2.2. A test path is a sequence of nodes and edges where nodes are states and edges are transitions. Each of those paths represents a unique behavior of the system. The collection of trigger parameters is considered as the input for that scenario while the state invariants are the objectives of the test. Thus they create test oracles according to the expected state of the system to compare the actual behavior with the requirement of the system represented in the specification model.

Based on the coverage criteria, there can be several paths in an Extended Finite State Machine (EFSM) starting from the initial state. However, not all of them are valid in the context of the object behavior. The lack of appropriate values of input to satisfy the guard conditions will lead to an infeasible path. Such infeasible paths of state machine create difficulties to automatically generate test case [14].

3. The third step is known as the concretization. The abstract test cases are too generic and need language-specific format to be executable. For example, if the target language is C, then the abstract test cases need to be transformed into C language. In addition, the main challenge of this step is to generate test data. To generate executable test cases for a test path, the model-based testing (MBT) tool needs to find specific input data for each of the triggers that are invoked during the execution of that test path. The test data can be gener-

ated randomly or using a more sophisticated algorithm such as evolutionary search[15] or symbolic execution [16].

4. The final step deals with test execution and analysis. To execute the generated test cases, the tool uses a test adapter to be integrated with the test execution framework so that the test can be executed against the real system under test (SUT). The concrete test cases are stored in the test repository as test scripts so that they can be executed later without re-generating. Thus the generated test cases are executed within a test execution framework and the outputs are analyzed and reported. The analysis of generated test cases are similar to traditional test analysis. A failed test is analyzed to check if it is the actual failure of the system or it is due to a fault in the test code or test setup.

2.2 Model Coverage Criteria

An extended finite state machine (EFSM) represents a set of behaviors of the system. Each path of the state machine involves a unique behavior of the system. In the second step of model-based testing process test paths are selected based on a model coverage criterion. In the following sections, several mostly used model coverage criteria such as structural coverage, requirement coverage and historical fault coverage are briefly described.

2.2.1 Structural Coverage Criteria

In this criterion, test paths are selected based on the structural elements of the model. The most commonly used two structural coverage criteria are- a) All-state and b) All-Transition.

All-State: In this structural coverage criterion, a minimum set of test paths are selected covering all states in the state machine. This strategy ensures that all states are tested at least once in the generated test suite [17]. The main drawback of this strategy is that it does not necessarily covers all transitions, because it only focuses on covering all states in a state machine diagram. So there is a possibility that it may leave a faulty transition untested.

All-Transition: This structural coverage criterion overcomes the drawback of All-State coverage criterion. Here, a minimum set of test paths are selected covering all the transitions of a state machine. This strategy ensures that all the transitions are tested at least once in the generated suite [13].

2.2.2 Requirement Coverage

Unified Modeling Language (UML) also enables users to model the system requirements using UML profiles. The structural elements of a state machine such as transitions and states can be associated with the requirements through UML profile. The Model Requirement Coverage is based on such requirements associated with the state machine. It selects a minimum set of test paths that cover all the requirements and thus ensure that the generated test cases ensure that all the requirements of the

system under test (SUT) are tested at least once [18].

2.2.3 Historical Fault Coverage

The analysis of a failed test identifies the faulty section of the system. This failure information can be fed back to the model to annotate the structural elements of the state machine with the failure information. The Historical Fault Coverage criterion selects a minimum set of test cases that covers all the faults annotated in the state machine [19]. Thus, it ensures that all the previous faults are tested at least once.

2.3 Test Data Generation Techniques

The big advantage of model-based testing is its potential for test automation. Based on the given model and test specications, test cases are generated systematically by using several test generation techniques. Each transition in a path can have a trigger. Different components of a transition are described in section 3.1.2. A trigger is an event that may or may not have parameters. A transition occurs only when the guard constraint holds true. The guard constraint is typically controlled by the input parameters of the trigger that are given to the system when the system is in the source state. Therefore, to make a transition happen the model-based testing tool must generate a set of input data that holds the guard constraint true. For example, assume a transition and its target state is testing the activation of dead reckoning system. In aircraft navigation, dead reckoning is a safety feature that guides the aircraft when a GPS failure occurs by calculating the current position using previously determined position and advancing it based on the estimated speeds

over the elapsed time and courses. So, to test this behavior, the MBT tool must generate proper sensor input that deliberately fails the GPS system and fires the trigger. Upon firing of that trigger, it is expected that the system goes to the target state, that is, it must activate the dead reckoning system. Finding such input data is tricky. There are various techniques currently being used in search-based software testing to generate such input data. Some of the widely used test data generation techniques are briefly described in the following sections.

2.3.1 Random Generation

In this strategy, the test data are generated randomly. It is the simplest technique for data generation. For each transition, each set of test data are tested against the system to check if it is sufficient to trigger the transition. The process of random data generation and executing and monitoring the system keeps going until a suitable set of data is found. Each set of input data is evaluated by a fitness function that represents how fit the data set is to fire the associated transition. This process of test data generation is not guided and is very time consuming. For large input spaces it may not even be able to find the appropriate inputs.

2.3.2 Symbolic Execution

Symbolic Execution is only applicable in white-box testing when there is access to the source code. In static symbolic execution, the program under test is executed symbolically [20]. The idea is that, it uses a symbol (instead of actual value) for each input parameter to the subsystem and represents suitable values for the desired path

using symbolic expressions also known as constraints. Those symbolic expressions are then solved by constraint solver to get the actual data that covers the desired path. In dynamic symbolic execution, which is also known as Concolic execution, initially the system runs symbolically and solves symbolic constraints to select actual data. Then the system actually runs with those actual values to check if it covers the desired execution path. Dynamic symbolic execution is being widely used in Concolic testing [21].

2.3.3 Search-Based Algorithms

In Search-Based Software Testing (SBST), search-based algorithms includes meta-heuristic search, simulated annealing and evolutionary algorithms. In recent years, search-based algorithms are being used extensively, particularly for automated test data generation [22]. Search-based software test data generation is briefly described in the section 2.4.

2.4 Search-Based Test Data Generation

Test data generation using search algorithms is a field of search-based software engineering (SBST) [23]. Meta-heuristic search strategies are being widely used to automate test data generation to cover specific program structures by exploring maximum execution paths to improve structural coverage in white-box testing. There are several meta-heuristic search algorithms that have been used in search-based software testing (SBST), such as, hill climbing (HC), simulated annealing, evolutionary algorithms (e.g., GA), particle swarm optimization, ant colony optimization [22]. Hill

climbing is a local search algorithm and is much simpler than evolutionary algorithms. The search-based software testing often uses much more complex search algorithms to explore particular execution paths. However, a complex and expensive search algorithm such as genetic algorithm is only used if it performs significantly better then other low-cost algorithms. A search algorithm, for example hill climbing that returns a local optimal solution and does not find the global optimal solution is known as local search. On the other hand, a global search algorithm finds the global optimal solution and is not bound by the local optima. The search algorithm generates a huge amount of candidate solutions. A function called fitness function, also know as objective function, is defined to evaluate the quality of candidate solutions. A candidate solution is just one point in the search space. The fitness function returns a numerical value that represents how fit the individual is to achieve it's objective. Usually, the fitness function returns a normalized branch distance and hence the target is to minimize the objective function. An individual that achieves the objective value zero is the solution. This function guides the search toward a better solution. The fitness function is explained in section 2.5. Several search algorithms that are widely used in software test data generation, are briefly described in the following sections.

2.4.1 Genetic Algorithm (GA)

Genetic Algorithms, first introduced by Holland [24], are based on natural selection process that mimics the biological evolution. The algorithm iteratively modifies a population of individual solutions until a termination criteria is reached. The algorithm terminates when it finds the best solution or it reaches the maximum number

of iteration. Such algorithms are well-known to solve both constrained and unconstrained optimization problems in Search-Based Software Engineering (SBSE). A population of individuals of a specific size are initialized at the beginning of the algorithm and evolve through the crossover and mutation processes. During the evolution process of individual solutions in the algorithm, a fitness function is used to compare candidate solutions and to select the best one out of many. Therefore, the fitness function should be well-defined for a specific optimization problem. More about the fitness functions will be described in section 2.5.

2.4.2 (1+1) Evolutionary algorithm (EA)

Evolutionary algorithms are first introduced by Holland [25]. (1+1) Evolutionary Algorithm (EA) is very basic and simpler compared to the genetic algorithms. In this algorithm the size of the population is one and after mutation a new individual is created. This is why its called (1+1) Evolutionary Algorithm. The newly created individual competes with it's parent to become the parent of next generation. In addition, λ mutants can be generated from a parent and compete with the parent to become the parent of the next generation. This is called (1+λ) Evolutionary Strategy. Compared to the genetic algorithm, (1+1) Evolutionary Algorithm does not use crossover strategy, instead, it only uses the mutation process to evolve an individual towards the final solution and thus explores the search space.

2.4.3 Alternating Variable Method (AVM)

Alternating Variable Method (AVM), introduced by Korel [26], is a local-search algorithm. It tries to improve the fitness of an individual by manipulating one variable at a time. Let an individual contain a set of variables $V = v_1, v_2, ..., v_n$. The initial solution takes the lowest possible values for each of those variables. It aims to find a solution by linearly searching the search space. For example, to maximize the fitness of an initial solution V, it first increments the value of v_1 in each step and keeps the rest of the variables constant. The algorithm stops searching if it finds a solution that meets the objective. If it does not find the solution but finds a maximum fitness with respect to v_1, then it switches to the second variable v_2. Now, the value of v_2 gets incremented each time to maximize the fitness while keeping v_1 to it's previously found fit value and all other variables constant. The linear search continues as such until it finds a global optimal solution or it has explored the local search space.

2.4.4 Random Search (RS)

Random Search (RS) is an unguided search algorithm. It randomly generates a sample solution and evaluates it using the fitness function. The search stops if the best solution is found or after a specific number of tries. The best solution so far is given as output of the algorithm.

All the meta-heuristic search algorithms described above are to find a solution that achieve a specific objective. Objectives are domain specific. In search-based software engineering (SBST), the measure of fitness to achieve such objectives is done by a fitness function which is described in the next section.

2.5 Fitness Functions

Fitness function is one of the main parts of a search algorithm. It provides a measure to evaluate individuals. This measure guides the search algorithm to move towards a better individual in search of finding the optimum solution. In recent years, search algorithms have been extensively used in software engineering research in the context of coverage improvement. Previous studies in this area proposed two different categories of fitness functions:

1. Coverage-based Fitness Function: In this category, fitness functions are designed to maximize the code coverage. This approach provides guidance to the search by rewarding or penalizing the individuals. To achieve a certain coverage, this approach rewards an individual if it covers a specific part of the program and penalize if it does not [27, 28]. Hence, complex program structures such as complex exponents, deeply nested conditions, or predicates that depend on relatively few input values from a large input space are most likely to be uncovered. Consequently, most of the time, the search is driven towards long easily executed program paths [28].

2. Control-based Fitness Function: In this category, fitness functions aim to help the search to cover certain control constructs of the program. This approach helps the search algorithm to systematically try and cover all structures of the program required by the model coverage criterion.

   The search is often guided by the fitness function that depends on either branch distance [26], control structure [29] or both [1; 30]. [1] first introduced a com-

Table 2.1: Branch distance measure introduced by Tracey et al. [1] for logical conditions. C is a failure constant added whenever the term is false.

Relational Predicate	Branch Distance Formulae
Boolean	if *true* then 0 else C
$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + C$
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else C
$a > b$	if $b - a < 0$ then 0 else $(b-a) + C$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b-a) + C$
$a < b$	if $a - b < 0$ then 0 else $(a-b) + C$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a-b) + C$
$a \neg b$	Negation is moved inwards and propagated over a

Table 2.2: Objective functions introduced by Tracey et al. [1] for conditionals containing logical connectives. $dist()$ is the objective function from Table 2.1.

Logical Connectives	Value
$a \vee b$	$dist(a) + dist(b)$
$a \wedge b$	$min(dist(a), dist(b))$
$a \Rightarrow b$	$min(dist(\neg a), dist(b))$
$a \Leftrightarrow b$	$min((dist(a) + dist(b)), (dist(\neg a) + dist(\neg b)))$
$a\ xor\ b$	$min((dist(a) + dist(\neg b)), (dist(\neg a) + dist(b)))$

bination of both branch distance and control structure information to calculate the fitness of individuals. A distance is calculated whenever execution follows a path which cannot lead to the target branch. Such distance, called branch distance, measures how close it was to cover the target branch.

The formulae used for calculating the branch distance for single and multiple logical conditions joined by logical connectives are represented in Tables 2.1 and 2.2 respectively. Thus, the branch distance is computed using all the conditions of a decision statement at which the flow of control diverted away from the target branch.

The distance for each condition is then added together to get the branch distance for the entire predicate. However, the numerical value of branch distance for each condition can be any number based on the operands included in the condition. The maximum distance of a boolean operand is 1 (assuming true=1 and false=0) while for an integer the maximum distance can be ($Integer.MAX\_VALUE - Integer.MIN\_VALUE$). Therefore, the calculated branch distance cannot be used directly unless it has been normalized. Hence, each of those distances for each condition needs to be normalized between 0 (minimum distance) and 1 (maximum distance). Then the sum of those distances represents the branch distance for the entire predicate.

Chapter 3

The Unified Modeling Language (UML)

The Unified Modeling Language (UML), a general-purpose modeling language that enables software designers to visually construct and specify system components [31]. Those constructs serve as specifications and help to better understand the system. UML diagrams are partial graphical representation of a model of a system under design. There are two categories of UML diagrams:

Structural Diagrams. Structural Diagrams define the architecture of the system. For example, class diagrams, object diagrams, package diagrams, component diagrams, deployment diagrams and profile diagrams.

Behavioural Diagrams. Behavioral Diagrams represent the behavior of the system and its objects. For example, activity diagrams, sequence diagrams, state machine diagrams, use-case diagrams etc.

In Model-Based Testing(MBT), behavioural diagrams such as activity diagram [32], sequence diagram [33], state machine diagram([34] are used in software test automation.

A state machine is a mathematical model of computation. In a state machine a system can be in exactly one of a finite number of states at any given time which is why it is also known as Finite State Machine (FSM) or Finite State Automaton (FSA). One of the extensions of FSM is Extended Finite State Machine (EFSM) that retains the benefit of traditional FSM while overcoming FSM's limitations. A brief description of EFSM is given in the following section.

3.1 Extended Finite State Machine (EFSM)

An Extended Finite State Machine Model (EFSM) is an enhanced mathematical model of computation based on the traditional Finite State Machine (FSM). It models a behavior of the system using states, transitions, triggers and guards. An EFSM is defined as, $M = (S, T, E, V, A)$, where S is a set of states, T is a set of transitions, E is a set of triggers, A is a set of actions and V is a set of variables. A transition t has a source state $source(t) \in S$, a target state $target(t) \in S$ and a label $label(t)$. The meta-model of an EFSM is shown in figure 3.1. Several components of an EFSM are described below.

3.1.1 State

In an EFSM, a state s represents a formal situation of a system or object which is defined by system variables V. In an EFSM there is only one initial state (also

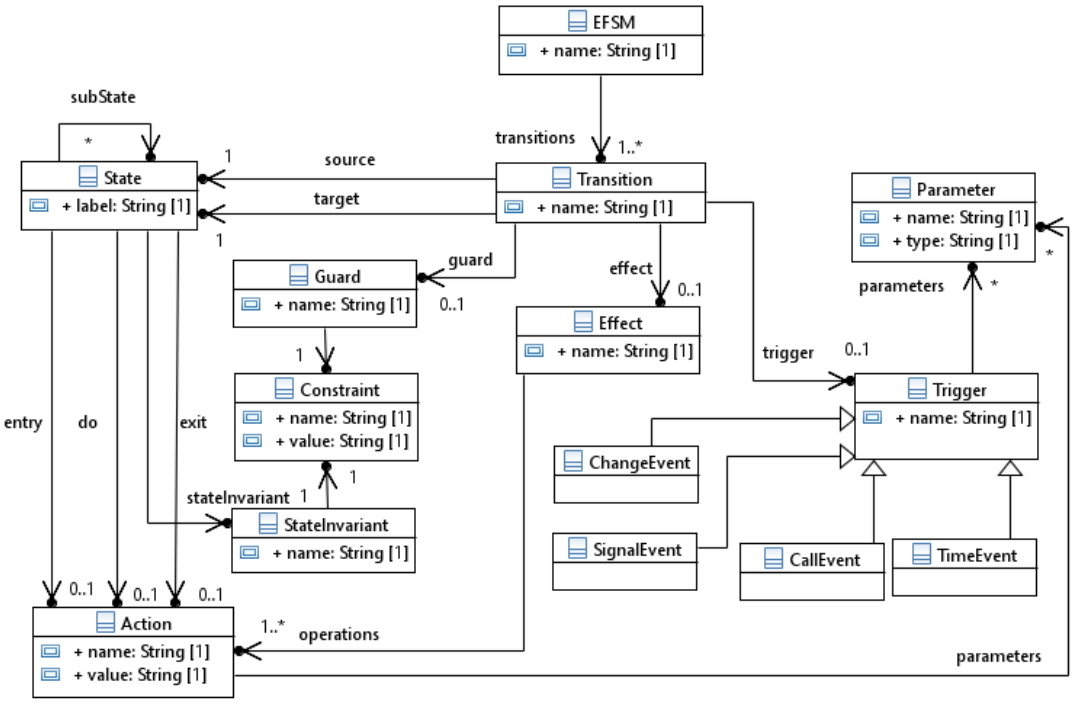

Figure 3.1: Extended Finite State Machine Meta-model

know as pseudostate) that represents the entry point of the behavior. There can be any number of other states required to represents the system's behavior. Each of the state is defined by a constraint c which is called the state invariant. A state invariant is formed by system variables V. The system stays in a state as long as the state invariant is true. An EFSM can optionally has an exit state. In order to comprehensively define a state behavior there are three types of actions (a) that are being used.

Entry: A set of actions occur in the beginning of the state.

Do: A set of actions occur after the Entry actions.

Figure 3.2: State and Transition

Exit: A set of actions occur when the system or object leaves the state.

A state can also represent a composite behavior of the system in that case it is called a composite state or nested state. Composite state is useful to define sub-system's behaviour.

3.1.2 Transition

A transition in an EFSM represents a transition between two states. A transition occurs in response to one of many external events as input. Those events are also known as triggers. The external events are categorized as $CallEvent$, $SignalEvent$, $ChangeEvent$ and $TimeEvent$. A transition T is defined as $T = e, g, a$ and is denoted as $e(p)[g]/a$, where $e(p) \in E$, p are the parameters of e, and g is a constraint that acts as the guard in the transition. The guard g is a constraint that prevents the transition to occur automatically. A transition from one state to another happens if the trigger e associated with that transition fires and the guard constraint satisfies. A trigger may cause a sequence of actions $a \in A$ (also known as effects) upon firing the transition that plays a role in moving the object to the next state. In a transition, if there is no trigger or guard then the transition occurs automatically.

In a traditional Finite State Machine (FSM), the transition is associated with a set

of input Boolean conditions and a set of output Boolean functions while in an EFSM model, the transition is expressed by $if-else$ statements of constraints (g). That is, upon satisfying the constraint the transition occurs and performs the specified data operations and brings the machine to a new state.

3.2 UML State Machine

Unified Modeling Language (UML) is a de-facto modeling language which has notations for describing all the components of an Extended Finite State Machine (EFSM). An EFSM written in UML is known as UML state machine or UML statechart. UML has notations to describe both the Mealy and Moore machines. In a mealy machine, an action depends on the current state of the system and the trigger event while in Moore machine, it depends on the entry and exit actions of the current state [35]. UML state machine also includes hierarchical nested structure to represent a composite state. In addition, it incorporates a declarative language, called Object Constraint Language (OCL), that can be used to describe rules or constraints in any UML diagram. The guard conditions, state invariants and even entry and exit behaviors of an EFSM are usually described using the OCL.

Chapter 4

Autopilot System at MicroPilot

MicroPilot is a Winnipeg based well-known company which has great reputation for their professional Unmanned Aerial Vehicle (UAV) products. In this following section, I will provide a high-level description of the MicroPilot autopilot system without getting into too deep and revealing confidential information of the system.

4.1 System Description

MicroPilot 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 the process of controlling and guiding such as taking off, keeping a safe altitude, climbing or descending to predefined altitude, heading to an assigned waypoint location, landing on an assigned location, etc. An autopilot system is also programmed to handle critical situations like 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 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. Each of the system features undergoes a rigorous unit and system-level testing. In this thesis, I focus on only system-level test generation. Running a system-level test with 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 that simulates the sensors and other environmental elements to see how the autopilot behaves in certain circumstances. This is why the system-level tests are also called behavioural tests. Traditionally, test engineers at MicroPilot manually write the tests and execute them under the simulator with the help of a test execution framework. Upon a test being successfully executed and passed in the simulator, then it goes to the next phase of testing to run it with the actual hardware.

Micropilot autopilot software can perform several flight commands. Some flight commands contain parameters (e.g., climb(x) and flyto(x,y)) and some of them do not (e.g., takeoff(), circuit() and flare()). One type of inputs of the autopilot system is the flight command parameters. MicroPilot autopilot is highly configurable by a set of input system variables. These variables 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 also modify those settings to customize and configure autopilot features they need. There are nearly 29,000 of such input system variables that does

not only define autopilot features but also communication links to the ground control systems.

All input system variables have corresponding code-level variables and any change in those variables by the simulator immediately changes the code-level variables as well. Thus, such variables act like an interface to the system that enables the test engineers to monitor and manipulate the system during run-time. There are two other types of system variables as well: sensor fields and state fields. 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. On the other hand, state fields reveal information about the system's state and health. There are in total 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. Therefore, the system variable is an abstraction of code-level variables. The simulator provides ways to manipulate the system inputs (e.g., flight commands, input fields and Sensor fields) and monitor the system's state.

Chapter 5

Literature Review

Model-based Testing (MBT) has been studied widely in the software engineering literature [36]. This test automation technique is even adopted by many industries such as telecommunication [37], automotive [38], etc. UML state machine has been one of the main notations to represent software specifications due to it's completeness and ease of use [13]. In this thesis, I also use UML state machines as the representational language of software specification. In Section 5.1 and 5.2, I review two main components of an MBT tool (items 2 and 3 from the Figure 2.1)- applying a coverage criterion to create abstract test cases and generating input data to concretize the abstract test cases. I have also proposed an approach to improve the code coverage of the generated test cases. Some previous studies regarding improving code coverage through MBT are presented in section 5.3. Finally, I mention some literature regarding Model-Driven Engineering(MDE) in safety critical system in section 5.4.

5.1 Model Coverage Criteria in MBT

To generate test cases from specification models, [39] proposed several coverage criteria based on data flow analysis. The authors proposed a set of coverage criteria, such as, all-nodes, all-edges, all-defs, all-p-uses, all-c-uses/some-p-uses, all-puses/some-c-uses, all-uses, all-du-paths, all-paths. All-nodes, all-edges and all-paths are known as state, transition and path coverage, respectively. All the other proposed coverage criteria- all-defs, all-p-uses, all-c-uses/some-p-uses and all-p-uses/some-c-uses lead to a set of complete paths that do not necessarily cover all associations between definitions and use of all variables. Additionally, all-uses and all-du-paths criteria cover all associations between definitions and uses of all variables. These data flow criteria ensures that every branch is traversed and every "important" path is also traversed. These approaches focus more on the data flow rather than control flow. Hence, these criteria enable branch testing and path testing and provide a way to intelligently select paths for testing.

Offutt et al.[13; 40] reported general model coverage criteria for automatically generating system-level tests from the UML state machine diagrams. They described techniques for generating test cases using different coverage criteria, such as full predicate coverage, transition coverage, transition pair coverage, and complete sequences coverage. The authors also provided a couple of practical solutions to define test cases from specifications of different costs. In their study, test specifications are described using simple algebraic predicate and specification graphs, a data structure that represents states and transitions in a graph, are represented using text. Those algebraic predicates consist of system variables appeared in specifications and relational oper-

ators such as equality, inequality and assignment.

Hong et al. [41] proposed a test sequence selection method from the UML state machine model using conventional data flow analysis techniques. The authors transformed the traditional state machine written in UML to an Extended Finite State Machine (EFSM). In the pre-processing stage, they flattened all the hierarchical and concurrent structures of states. In addition, the authors assumed that at a time exactly one external event can occur. Consequently, the number of transitions is further reduced and formed a flow graph. A set of test sequence is generated from the resulting flow graph covering all-transition criteria along with constraints that appear in the original finite state machine. They assumed a transition is a single execution unit and the data was traced between single execution unit and the flow of control.

Briand et al.[42] investigated the cost-effectiveness of adequate test suite for four different coverage criteria regarding UML state machine diagram: a) transition coverage, b) full transition-pair coverage, c) all paths coverage in label-transition tree and d) full predicate coverage. They measured the cost of test suite using the size of the test suite. The effectiveness of the generated test suite is measured in terms of fault detection rate. The size of the test suite depends on the number of method calls in the source state machine which is called the cumulative length of a test suite. The system under test varies in size and complexity, hence, the relative cost of coverage criteria is compared with the fault detection capability to measure the effectiveness. Each test sequence- a sequence of transitions starting from the initial state to the leaf state forms a test case. The authors also explicitly defined a maximum length of test paths considering two aspects: 1) the set of test paths must be adequate to cover all

transitions and states of the label transition tree from root to leaf and 2) generated sequences must be able to provide the incremental effect of increasing coverage. A new test case is added to the test suite if it is not already taken, and if it maximizes the coverage. Thus it creates a test suite of a minimum size.

5.2 Input Data Generation in MBT

Doungsa-ard et al.[15] introduced a novel approach to generate input data from the constraints using Genetic Algorithm (GA). They used all-transition coverage criterion to select a set of test paths for test generation. Based on that, the authors developed a model-based testing tool that includes genetic algorithm for test data generation; and execution of state machine model written in UML. The state machine execution module extracts triggers from a test sequence to test with the state machine. For each trigger it records the value of system variables included in the guard predicate of that transition. Using those recorded values, it measures the fitness value of the provided input data. To generate test data using the genetic algorithm, the author used a sequence of triggers as their chromosome format, a random algorithm for the mutation, and a two-point algorithm for crossover.

Chevalley et al.[43] introduced a statistical testing technique to automate test generation from state-machine diagrams. They also used all-transition coverage criterion for test adequacy. They introduced a functional distribution algorithm that dramatically increases the rate of changes in the selected input data capable of triggering one of the outgoing transitions. In order to automate test generation in real-time environment, they proposed to implement two modules: generators and collectors.

To identify and select the last trigger-enabled transition, Generator module generates the test input data while the collector module collects the measure of transition coverage and feed them back to the functional distribution technique of the generator module. Since, the guard predicates of some transitions depend on each other, i.e., the transitions are interdependent, therefore, the functional distribution fails to provide balanced coverage of transitions. Also, the stopping rule in their approach which is used to tune the size of the test has insufficient experimental validation. In addition, the study lacks a comparison, in terms of cost and effectiveness, with other well-established techniques.

Samuel et al.[44] introduced an automatic test data generation approach, that uses the control and data flow logics available in the finite state machine model. Their approach includes traversing the finite state machine to extract conditional predicates associated to each transition and applying function minimization techniques to generate test cases. The auto-generated test cases can be used to test composite state based behaviors. This approach only takes change events and time events as triggers along with guarded transitions. The authors reduced the size of the generated test suite that achieves transition coverage by early-testing the boundaries determined by predicates. Therefore, the maximum size of the generated test suite depends on the number of predicates in that state machine. However, their method is only applicable to boolean and integer data types. The authors used alternating variable method to generate test input data from local optimal solution. Another drawback of their modeling approach is that they assumed the constraints are represented as notes in the model instead of proper notation using the class diagrams, data types, attributes,

and constraints. Consequently, the constraints in the class diagrams are not even extracted. Another drawback of their approach is that, a test path is a sequence of adjacent transitions that starts from the initial node to a terminal node in the state machine diagram, consequently, most of the time this condition fails to satisfy the path condition since not every transition path ends with a terminal node.

Lefticaru et al.[45] used a genetic algorithm (GA) to find feasible input data for the method parameters extracted from the UML state machine diagram. The entire test generation process includes two potential steps: a) Finding a set of feasible sequence of transitions using all-transition coverage criteria. b) Finding the input parameter values for each method in the sequence that trigger the transitions in the test path. The authors represented each individual as $C = (c_1, c_2, ..., c_n) \in d_1 * d_2 * ... * d_n$, where $d_1, d_2, ..., d_n$ are the domains of the input variables and n represents the number of events in that sequence. The measure of fitness of a parameter includes two components: a) Approach distance- representing the number of transitions that have been covered in-ordered. b) Normalized branch distance- which represents the proximity of the guard conditions to be true for the state where the desired path diverges. They did not consider hierarchical or concurrent state machine diagrams.

Briand et al. [46] proposed a technique to automate the generation of test constraints based on states, event parameters, and actions for a sequence of transition. Here a test path is termed as the Transition Test Sequence (TTS), meaning, a feasible sequence of transitions that starts from the start state and ends in any of the leaf states. The authors defined an appropriate model representation technique and the Invocation Sequence Tree (IST) to represent every possible sequence of actions from

a TTS. The IST contains the extracted constraints from the TTS. IST also includes extracted models and information of the state-machine diagram. The model constraints are written in Object Constraint Language (OCL). The authors represented four different types of normalization for the OCL expression to support the analysis of model constraints and finally, to generate test constraints. But, while modeling operation contracts, the normalization of post-conditions should follow certain patterns. In addition, it requires many transformations before the generation of test data. Complex redundant constraints have to be identified manually and removed since complex redundant constraints slow down the constraint solving algorithms. Inconsistencies in the constraints can cause failure to converge towards a result.

Swain et al. [47] presented an approach to automate test generation from UML state machine diagram. Their approach consists of the following steps: a) Constructing the state machine diagram. b) Transforming the state machine diagram to state chart graph. c) Traversing the state-chart using depth-first search (DFS), a graph traversal algorithm to find sequences of states and associated transitions with predicates using various coverage criteria e.g., all transition, all state. It also supports composite states. d) Transforming the relational expressions of predicates to a predicate function. To find an appropriate value for the input parameter they evaluate the predicate function against that input value. After each iteration, input values are modified to evaluate the predicate function using the alternating variable method. When an appropriate parameter value is found the predicates are transformed to source code. e) Generate extended finite state machine from the source code. Finally, f) Generate the test case for each conditional predicate.

Kansomkeat et al.[48] introduced an approach to automatically generate test cases from UML state machine model that includes the following steps: a) Creating the state machine diagram using the IBM Rational Rose [49], a UML modeling tool. b) Transforming the state machine diagram into Test Flow Graph (TFG) using the proposed transformation method. TFG is a flattened hierarchical structure of states. c) Parsing transition sequence from the root state to each leaf state to achieve either all-state or all-transition coverage. d) Generating test cases from the transition sequence using GA. Finally, the effectiveness of test suite is measured by it's fault detection abilities using mutation analysis.

Shirole et al.[50] introduced a technique to automatically generate feasible test paths and data using the Genetic Algorithm from the traditional state machine diagram. Their proposed technique includes the following steps: a) Transforming the traditional state machine diagram into Extended Finite State Machine (EFSM). b) Transforming the extended finite state machine model into Extended Control Flow Graph (ECFG). c) Finding a set of feasible test paths satisfying the all-definition coverage criterion d) Generating test input data using Genetic Algorithm (GA). The fitness score used to evaluate each candidate solution of the genetic algorithm is measured as

$$fs = (K + 1) * (N - 1) * C_1 * C_2 \tag{5.1}$$

where K is the number of cycles allowed in the sequence, N represents the number of vertices, C_1 and C_2 are constants. The followings are the key points used in their approach to optimize the fitness score fs to achieve complete feasible path: a) When the start node of the test path is equal to the start node of the extended control flow

graph, then fitness value is decremented by constant C_2. b) When the end node of the path is equal to the last node of the extended control flow graph, then fitness value is decremented by constant C_2. c) Upon satisfying the guard predicate the fitness value is decremented by the constant C_1, otherwise incremented by $2 * C_1$. The feasibility of a test path depends on the sequence of transitions and the values of data members associated with that transition at that time. The state machine diagrams that this approach accepts are quite simple and have limited behavior.

Ali et al. [51] proposed several meta-heuristic search techniques, with optimized fitness function, to generate test data from Object Constraint Language (OCL) constraints. To guide search algorithms, the fitness function the authors used is based on branch distance. The main outcome of their study is to solve independent conditions of an OCL constraint separately and combined them at the end instead of solving the complex OCL constraints in the beginning. The authors compared their approach with some widely established algorithms, such as, Genetic Algorithm (GA), (1+1) Evolutionary Algorithm (EA) and Alternating Variable Method (AVM) and showed their approach's superiority over the others.

To provide a summary of literature reviewed in this section, table 5.1 shows a list of MBT approaches with their model coverage criteria (i.e., test adequacy criteria) and data generation techniques. All of them generate unit-level test cases and use the constraints as it is in the state machines. In this study I propose to evaluate each of the individual conditions in a constraint with the hope of increasing code-level C/D coverage. In chapter 7, I present the approach.

Micskei et al. [52] and Shafique et al. [53] have also reviewed several up-to-date

Table 5.1: A list of MBT approaches that generate code-level test cases.

Literature	Coverage Criteria	Data Generation	Conds. Cov.
[15]	Transition	Genetic Algorithm	NO
[43]	Transition	statistical	NO
[44]	Transition	AVM	NO
[45]	Transition	Genetic Algorithm	NO
[46]	Transition	Symbolic Execution	NO
[47]	State,Transition	AVM	NO
[48]	Transition	Genetic Algorithm	NO
[50]	Definition	Genetic Algorithm	NO
[51]	State,Transition	GA,AVM,EA	NO

Table 5.2: List of existing open-source MBT tools.

Tool	Reference	Last Modified	Input
fMBT	[54]	2017	Custom (AAL)
GraphWalker	[55]	2017	FSM
JSXM	[56]	2016	EFSM(Stream X machines)
MISTA	[57]	2015	PetriNet
JTroX	[58]	2014	LTS
Modbat	[59]	2015	EFSM(Scala)
ModelJUnit	[60]	2014	EFSM(Java)
MoMuT::UML	[61]	2014	FSM(UML)
OSMO	[62]	2016	Model (Java)

existing commercial, academic and open-source MBT tools that allow generating unit-level test cases. Commercial tools always include user manuals and case studies but do not include technical information since the vendors do not publish research papers due to the proprietary reasons. So, I investigated those open-source tools to see if I can take any of them and modify components I need. Table 5.2 shows a list of widely used open-source and academic MBT tools. These tools use traditional Finite State Machines (FSM), non-UML Extended Finite State machines (EFSM), Label Transition Systems(LTS) or Petri-Nets as the input but none of them takes UML based EFSM as input to generate tests.

Note that in this thesis, I am not only focusing on improving a test generation technique, but rather I aim to implement a fully functional MBT tool that can be used in MicroPilot [9] production environment. Therefore, the tool needs to be accurate, robust, easy to use and efficient.

5.3 Improving Code Coverage through Model-Based Testing (MBT)

In this section, I provide previous studies regarding existing model-based testing tools that aim to improve code coverage while generating test cases from the specification model.

Hartman et al.[63] created a prototype called Automated Generation and Execution of Test Suites for Distributed Component-based Software (AGEDIS). It is a model-based testing tool that includes components such as UML modeling tool, code-level test generation engine, test suite browser, defect analyzer and reporter. The UML model including class diagrams, block diagrams, and behavioral diagrams are then provided as inputs to generate test cases. In addition, AGEDIS is integrated with [64], a functional coverage tool, which allows test engineer to define function coverage model with methods, parameters, and variables of an object. The coverage report provided by [64] is feedback to the AGEDIS, which in turn creates test cases to improve functional coverage.

Artho et al.[65] provided a case-study combining model-based testing of Java programs with systematic exploration of input domain with run-time verification. The input domain of the system under test is explored using a model checker composed of symbolic execution tool Java PathFinder [66]. The run-time verification includes code instrumentation and event observation that enables evaluating constraints at run-time using EAGLE [67], a run-time program monitor. However, this approach does not aim to improve code coverage, instead, it aims to detect concurrency errors

such as dead-lock and race conditions.

Kicillof et al.[68] at Microsoft introduced a novel approach to automate low-level test generation by combining black-box model-based testing with white-box parameterized unit testing. The authors uses Pex[69], a symbolic execution tool, also developed at Microsoft, to extract concrete parameters values to improve branch coverage at the code-level. Then the parameter values are fed back to the black-box model based testing tool [70] which generates multiple test cases for each methods with a high branch coverage.

Vishal et al. [71] also studied the similar gray-box Constraint-Based Testing (CBT) approach to improve code coverage using SpecExplorer [70]. Constraint based testing is a promising technique in white box testing that uses constraint solvers to generate test data. SpecExplorer [70] is a white-box model-based testing tool for C-Sharp, described earlier in the literature, that uses Z3 SMT Solver [72] to get decision table. 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 [69]. 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.

Table 5.3 summarizes all the previous studies provided in this section. All of these approaches are applicable for generating unit tests from the UML behavioural models. All the approaches need access to the source code for static analysis to get program constraints in order to generate input test data. However, in this thesis I

Table 5.3: List of MBT approaches that increase code coverage.

Literature	Tool	Testing Criteria	Testing Level
[63]	AGEDIS	white-box	Unit test (C++)
[65]	PathFinder, EAGLE	white-box	Unit test (Java)
[68]	SpecExplorer, Pex, Z3	gray-box	Unit test
[71]	SpecExplorer, Pex, Z3	gray-box	Unit test C#

am looking for a black-box testing solution to improve the code coverage through MBT while generating system-level test cases without having any sort of access to the source code.

5.4 Model-Based Testing (MBT) in Safety-Critical Avionics Systems

There are several applications of Model-driven Engineering in safety-critical avionics systems; Model-based testing is only one of them. I have studied several UML-based solutions [73; 74; 75; 76; 77] that can be attributed to capture safety-related information in UML models, in case they can be useful add-ons to a typical UML model. For example, a study conducted by [78] introduced a new UML profile that can be used to extend standard UML test models for model-based testing with information that is relevant to airworthiness certification. Those test models can serve as supporting artifacts of certification in case MBT is applied. Apparently, none of the solutions proposed in this category of studies used an MBT technique to automate system level test generation. Instead, they only focused on modeling the system along with the safety-related information. In this study, I only use basic concepts such as timing from real-time domain which can still be modeled in UML standard diagrams. So I do not use any extra profile.

Chapter 6

Implementation of the Model-Based Testing (MBT) Tool

In this chapter, I present the implementation detail of the Model-Based Testing (MBT) tool. Developing such tool that automates system-level test generation for an autopilot is more complex than automating unit-level test generation since it requires seamless integration with the simulator, test execution framework, and adapting with a system specific test script. In the next section, I describe the modeling tool and in section 6.2 I briefly describe the process of MBT tool development.

6.1 Modeling Tools

Model-Based Testing (MBT) technique requires two types of development tools: a) The UML Modeling tool: it allows to write specification models with standard UML notations and b) MBT Tool: it automates the test generation process from the

Table 6.1: UML-2 Modeling tools

Name	Owner	License Type	OS Platform	Language
Papyrus	Eclipse Foundation	Open-source	Cross-platform (Java)	Java
Rational Software Arch.	IBM	Commercial	Windows, Linux	Java/C++
Enterprise Architect	Sparx Systems	Commercial	Windows	C++
MagicDraw	No Magic Inc.	Commercial	Cross-platform (Java)	Java

model. To choose a UML-2 modeling tool I have investigated several open-source and commercial modeling tools. Table 6.1 shows major tools that are being widely used in industry and research [79].

As the modeling tool, I chose to use [11], an industrial-grade open source model-based software engineering tool, over the others for several reasons. Such as,

1. It enables to use Eclipse Modeling Framework (EMF), which provides framework and code generation facility to build tools and other standalone applications based on a structured data model. EMF provides extensive library and run-time support to parse models and generate java/C/C++ code based on the model.

2. It brings an opportunity to use additional modeling environments such as Papyrus-RT (Real-time Systems modeling), RobotML(a Papyrus-based modeling environment dedicated to robotics), Eclipse UML Profiles Repository and Eclipse Safety Framework (tools for integrating safety techniques within a model driven engineering process). It also enables using a collection of papyrus-based tool chain that complements it.

3. It allows writing executable models.

4. It is open-source, academia friendly, customizable, highly documented and has a big industrial community.

5. It allows a domain specific modeling language by using UML profiles.

Papyrus supports creating Object Management Group (OMG) standardized UML diagrams [80], SysML diagrams and UML profiles. It stores the model in a serialized UML file which is easily parsable by EMF run-time library.

6.2 Developing the MBT Tool

Developing a Model-Based Testing (MBT) tool requires several steps from parsing the model to executing the test in the target environment. This section describes those steps briefly.

1. **Parsing the model.** Papyrus stores the model in the OMG specified XML format. I parse the XML files with the help of Eclipse Modeling Framework (EMF) that provides a number of complete APIs to work with the UML model generated by eclipse. The parsed models may require necessary cleanups. For example, the state machine can contain composite states. If a composite state has one or more incoming or outgoing transitions, then all the internal states are considered to have those incoming and outgoing transitions as well. Those transitions will also have the same events and guards.

2. **Extracting test path based on the All-Transition coverage criterion recommended by DO-331 using graph traversal algorithm.** As I explained in the section 2.2, All-transition coverage criterion ensures that all the

transitions in the state machine are traversed and to be tested at least once. I used a graph traversal algorithm, Depth First Search(DFS), to traverse the state machine from the initial state to all the leaf states based on the coverage criterion and retrieve a set of test paths.

3. **Writing test code.** This step is usually called model-to-text transformation in model-driven engineering. In most related work, the generated tests are in the format of a common programming language (C, C++, Java, Python, etc.). However, in our case, the goal is not to generate unit test but rather system-level tests that run on the Autopilot's simulator. So the abstract test cases need to be converted into MicroPilot's own test format. The tests are written in a language independent format and stored in a relational database. Hence, the generated test cases need a transformation to a specific data structure.

4. **Generating appropriate test input data for each event in the test path that triggers the transition.** The generation of input data is a challenging task in both system and unit level testing. In MBT, usually, an event (e.g., *CallEvent*), causing an initialization of a trigger, can have one or more parameters. Based on the System Under Test (SUT), only a proper combination of parameter values will cause the transition to happen (assuming there are guards on the transition). Finding an appropriate value for a parameter is more challenging when there is no access to the source code. To generate such test data I used two different techniques which are described in the section 7.4. There are two ways to test the generated input data against the system:

(a) **Symbolic execution**: In white-box testing, where there is access to the source code, the System Under Test (SUT) can be executed symbolically based on the actual implementation of the system. The symbolic execution returns a set of constraints that need to be solved by constraint solvers to get a range of values for each parameter. Finally, it selects an absolute value for the input parameters based on the range.

(b) **Concrete Execution**: In black-box testing, where there is no access to the source code, unlike the symbolic execution, the system actually runs in the test environment with the generated test data. In this case, the test data is generated using a test data generation technique explained in section 2.3. To evaluate each set of test data a fitness function needs to be defined. Based on the fitness returned by the fitness function a final best solution is selected.

The goal of this MBT tool is to generate system-level test cases which is black-box testing. The purpose of the system-level testing is to test the system as a whole instead of code-level unit. Hence, I cannot apply symbolic execution rather concrete execution with a search algorithm. To execute the system under test to evaluate a set of input data, I have integrated the simulator and the test execution framework. Applying an expensive search algorithm such as genetic algorithm would be justified if the solution is not achievable by inexpensive algorithms or it performs considerably well over the other. So, I begin by applying a less expensive search algorithm- (1+1) evolutionary algorithm. I have also implemented the random search to compare its result with the evolutionary

algorithm.

Chapter 7

Improving Model Constraint Coverage Using Model-Based Testing

One of the requirement of DO-178C certification is to provide adequate test suites. The most critical Level-A certification requires 100% code coverage [3] in terms of Modified Condition/Decision coverage (MC/DC). In the next couple of sections, I briefly describe C/D and MC/DC criteria and in section 7.3, I describe how I generate combination of predicates for a test path covering MC/DC criterion. Section 7.4 shows how model-based testing improves such coverage.

7.1 Condition/Decision (C/D) Coverage

The basic code coverage criterion is the statement coverage which represents what percentage of statements in the code has been covered by a test suite. Having 100% statement coverage does not necessarily mean the code has been tested comprehensively and there is no bug in it. Due to the short-circuit behavior of compilers during evaluating conditional statements (i.e., decision) a great deal of conditions can be left untested. One of the improvements to statement coverage is Condition coverage which represents what percentage of conditions has been covered by a test suite. However, 100% condition coverage is also not sufficient to test a decision thoroughly since it does not guarantee both true and false evaluation of each condition. Condition/Decision (C/D) coverage is a code coverage criterion that does not have this drawback and emphasizes on testing each of the conditions of each decision. It is a hybrid of condition coverage and decision coverage and is measured as the union of condition coverage and decision coverage. This coverage criterion ensures the possible outcome of each condition of the decision are tested at least once. However, it does not guarantee testing of independent effect each condition in a decision.

7.2 Modified Condition/Decision Coverage (MC/DC)

Modified Condition/Decision Coverage (MC/DC) criterion is a modification of C/D coverage proposed by the safety certification [3]. 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 following conditions:

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.

There are two ways to show a condition's independent effect on a decision's outcome: a) varying just that condition while keeping all the other possible conditions fixed or b) varying just that condition while keeping fixed all the other possible conditions that could affect the decision's outcome.

To measure the coverage of system-level test cases, MicroPilot uses the BullsEye-Coverage tool [81]. This tool does not give a measure of MC/DC coverage but function coverage and C/D coverage of the code. When the program compiles, BullsEyeCoverage instruments the source-code such that it can keep track of which lines and conditions has been invoked during the execution of a test case.

7.3 Generating Predicate Combinations Covering MC/DC Criterion of Model Constraints

To improve the code coverage using the system level test cases, I target model constraints (i.e., guards of each transition) in an MC/DC manner. The idea is that unlike a typical MBT coverage criterion (e.g. all state coverage and all transitions

Table 7.1: Modified Condition/Decision Coverage of a Sample Predicate.

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

Table 7.2: An example of MC/DC Combinations of a predicate having three conditions.

Original predicates (n=3)	A *and* $(B$ *or* $C)$
Splitted by OR	A *and* B A *and* C
New predicates **by negating** **each condition** (Total $n + 1$: 4)	$\neg A$ *and* B A *and* $\neg B$ $\neg A$ *and* C A *and* $\neg C$

coverage) I will focus on a more demanding criterion (MC/DC on the model-level). To be more specific, if a guard predicate of a transition contains n conditions then it takes $n + 1$ test cases to cover all MC/DC conditions of the guards constraints.

For example, for a guard condition ($A > B$ *and* $X > Y$ *and* $P > Q$), there need to be at least four test cases shown in the table 7.1 to meet the MC/DC criterion for that predicate.

However, a test path typically includes several transitions where each of them may have a guard predicate. To generate a full test suite out of all these predicates I follow these steps:

First, each predicate is parsed as an expression tree. Each non-leaf node represents operators (i.e., *and* and *or*) and leaf node represents conditions (e.g., $A > B$, $X > Y$). One of the major advantages of MC/DC criterion is that it prevents the short-circuit in predicate evaluation. For example, for a predicate $M > NorR > S$,

Table 7.3: An example of MC/DC Combinations of two guard predicates.

Original predicates	A *and* $(B$ *or* $C)$	$(X$ *or* $Y)$ *and* Z
Splitted by OR	A *and* B A *and* C	X *and* Z Y *and* Z
Combinations	A *and* B, X *and* Y A *and* B, Y *and* Z A *and* C, X *and* Y A *and* C, Y *and* Z	
New predicates by negating each condition (Total: 16)	$\neg A$ *and* B,X *and* Y A *and* $\neg B$, X *and* Y A *and* B, $\neg X$ *and* Y A *and* B, X *and* $\neg Y$ $\neg A$ *and* B, Y *and* Z A *and* $\neg B$, Y *and* Z A *and* B, $\neg Y$ *and* Z A *and* B, Y *and* $\neg Z$ $\neg A$ *and* C, X *and* Y A *and* $\neg C$, X *and* Y A *and* C, $\neg X$ *and* Y A *and* C, X *and* $\neg Y$ $\neg A$ *and* C, Y *and* Z A *and* $\neg C$, Y *and* Z A *and* C, $\neg Y$ *and* Z A *and* C, Y *and* $\neg Z$	

if the first condition $M > N$ is true then the evaluation of the second condition does not affect the evaluation of the whole predicate. Hence, the system skips evaluating the second condition.

To prevent such short-circuit, I split each predicate containing operator *or* into multiple predicates, so that all predicates contain only *and*. Now I take combinations of predicates of all transitions, so that all the combinations are individually tested. After that, from each of those predicates in a combination, several more predicates are generated by negating each condition, so that each of those conditions is accessed individually to make sure they have an effect on the overall predicate's evaluation.

Thus, if a predicate contains n conditions, then there are at least $n+1$ combinations of predicates required to cover MC/DC criterion for that predicate shown in table 7.2.

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 of this procedure for a test path having two guard predicates, the table 7.3 shows combinations covering the MC/DC criterion. This example shows how 16 more predicate sets are generated from a test path having only two guard predicates. That means, the tool will generate 16 more test cases for that test path.

7.4 Test Data Generation

Each of those combinations described in the previous section, is a different execution path of the autopilot system. To explore such path, I need a set of test data that guides the autopilot to follow that path. Since the idea is to test independent effect of each condition in a predicate, all the internal and external input variables that control the variable in each condition need to be identified and considered in the input data set. There are two types of internal variable in the system under test: flight command parameters and autopilot settings. The external variables of the system are the sensor inputs.

In traditional model-based testing, the test input data for a test path is generated for each transition at a time. However, in the system under test, flight command parameters and autopilot settings are set once before the system starts assuming the

UAV is fully controlled by the autopilot instead of a ground control system. In the avionic domain, autopilot settings are configured before the start of a flight. Changing the configuration settings in the middle of the flight is unrealistic and a bad practice. However, it is only the external variables i.e., the sensor input that changes over time during a flight. Therefore, I chose to generate input for the entire test path at once in the beginning instead of doing it transition by transition. I provide the flight command parameters and autopilot settings before a flight starts and simulate the sensor inputs in the beginning of the flight.

To generate the test input data that is appropriate for the test path, I use an efficient search algorithm. The entire procedure of generating the test input data for each path of a state machine is summarized in the algorithm 1 where test data is generated using a search algorithm.

Algorithm 1 GenerateTestDataUsingSearch(startstate,statemachine)

1: $testPaths \leftarrow DepthFirstSearch(startstate, statemachine)$
2: $input[\,]$
3: $inputFields \leftarrow input\ system\ variables$
4: $SensorFields \leftarrow Sensor\ fields\ in\ environment$
5: **for all** $path\ p \in testPaths$ **do**
6: $\quad guard \leftarrow guard\ in\ t$
7: $\quad flightCommand \leftarrow flight\ command\ in\ tr$
8: $\quad parameters \leftarrow parameters\ in\ flightCommand$
9: $\quad testData \leftarrow Search.GetSolution(parameters, inputFields, SensorFields)$
10: $\quad input[p] \leftarrow testData;$
11: **end for**
12: **return** *input*

The use of an expensive search algorithm is only justified when the solution is not achievable by an inexpensive one or it performs significantly well over the other. Applying a genetic algorithm in our case would be very expensive since it takes

running the simulator each time to evaluate an individual which takes several minutes per individual. Therefore, I opt to use rather less expensive search algorithms- (1+1) evolutionary algorithm and random search. The following sections provides brief description of those algorithms.

7.4.1 (1+1) Evolutionary Algorithm (1+1 EA)

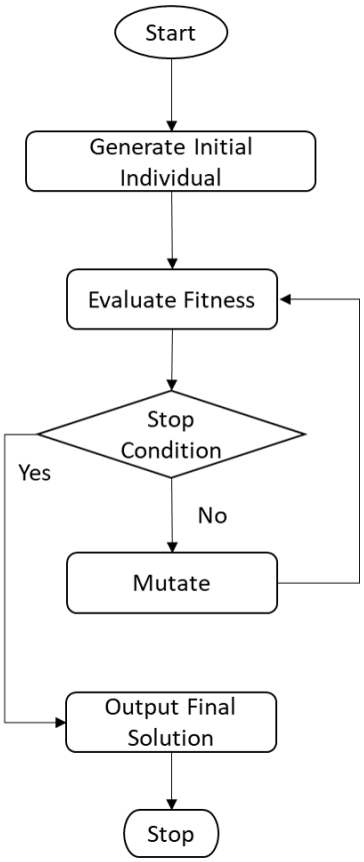

Figure 7.1: A Standard (1+1) Evolutionary Algorithm

I have applied a basic (1+1) Evolutionary Algorithm with 50% mutation rate and with maximum of 200 iterations. The cost of applying a search algorithm is briefly mentioned in section 7.7. There are more than 29,000 input variables by default if

there is no dependency analysis. The number of input variables in the search varies and it depends on the model constraints and the dependency analysis. The type of input variables are boolean, integer, signed integer and geographical coordinate. Therefore, the search space is considerably large. Due to the huge search space and expensive execution, I opt to use a high mutation rate that might help exploring the search space in a fast fashion. The entire process of a standard evolutionary algorithm is summarized in Figure 7.1. Usually, in an evolutionary algorithm, the initial individual is generated randomly. (1+1) Evolutionary Algorithm uses an evolutionary strategy called mutation rate. The mutation rate controls the internal change an individual goes through. For each gene in an individual, a random number is generated. If the number is less than the mutation rate then the gene encounters a change otherwise left unchanged. In the mutation, the value a particular gene gets depends on its data type. The value can be anything between the $DataType.MIN\_VALUE$ and $DataType.MAX\_VALUE$. For example, an gene of type integer gets a value between -2147483648 and 2147483647 while another gene of type boolean receives value 0 or 1.

Figure 7.1 represents how a standard (1+1) Evolutionary Algorithm algorithm works. Initially, a random individual is generated. If the individual is fit, that is, the fitness function returns zero then the individual is taken as the solution otherwise it undergoes a mutation. The mutated individual is evaluated again to see its fitness. If it is fit then its taken, otherwise, it undergoes a mutation again. This process keeps continuing until the a termination criterion is reached. The termination occurs if a solution is found or the iteration reaches a maximum number of generations.

7.4.2 Random Search

To compare the performance of (1+1) Evolutionary Algorithm, I 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. It returns a solution if found otherwise keeps iteration until the maximum iteration is reached.

7.5 Optimization Problem Formulation

The idea of applying search algorithms is to find a set of input values that satisfy certain model constraints. The objective function of this optimization problem is as follow.

$$F = \sum_{t=1}^{n} B_t + AL$$

Here, F is the fitness of an individual. AL is the approximation level, which is the index of first uncovered transition from the end. B_t= distance of the constraint at transition t. The components of the objective function are described in details in the next section. This function measures a distance, representing, how far it is to satisfy a certain model constraint. The objective is to minimize that distance, hence, it is a minimization problem. The number of input variables in the search varies and it depends on the model constraints and the dependency analysis. By default there are 29000 input system variables and 160 sensor variables. The type of input variables are boolean, enum, integer, signed integer and geographical coordinate.

7.6 Evaluating Fitness of An Individual

I have integrated 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 value of specified state fields over time. After the execution, the simulator stores the log in a file, which means that I can calculate the fitness only after the execution.

The objective of each search is to cover a specific branch 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 branch distance described in section 7.6 and approximation level. Approximation level, first introduced by [30], guides the search to satisfy transition guard constraints in-order from start to end of a test path. An individual satisfying all the guard constraints except the last is expected to be fitter than the individual satisfying all the guard constraints except the first. The final fitness of an individual is the sum of branch distance and approximation level.

$$F = \sum_{t=1}^{n} B_t + AL$$

Here, F is the fitness of an individual. AL is the approximation level, which is the index of first uncovered transition from the end. B_t= distance of the constraint at transition t.

A guard constraint may have several OR logical operators. So, to evaluate each individual accordingly, each constraint splits into several constraints, based on the number of OR operator it has. For example, a constraint (x *or* y) *and* z splits into two constraints (x *and* z) and (y *and* z). So, if any of the subsequent constraint

evaluates to true then the main constraint is also true. Similarly, the distance of the main constraint is equal to the minimum distance of any of the subsequent constraint.

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

b_{tj} are the distances of the constraints found by splitting constraint b_t with OR. If any individual covers any b_{tj} constraint then it covers b_t as well.

The distance of a guard constraint is the sum of normalized distance of each individual condition in that constraint.

$$b_{tj} = \sum_{c=1}^{K} N(D_c)$$

Here, D_c is the distance of condition c and $N(D_c)$ is the normalized distance of condition c.

$$N(D_c) = \frac{dist(c)}{maximum\ possible\ dist(c)}$$

7.7 Optimization

As mentioned in section 4.1, there are 29000 input fields, 160 sensor fields in the system under test. The type of each of those fields varies from boolean to signed integer or even geographical coordinates. The set of input variables of a test path includes several such fields depending on the state fields included in guard constraints. Moreover, the execution of the simulator is very expensive in terms of time and computational resources. Therefore, the search needs to be optimized so that it can generate test cases with a low budget.

7.7.1 Reducing Search Space

The ideal way to reduce the search space is by knowing the set of internal and external variables that affects the transitions in the test path. The predicates in a transition contains state fields and input fields. The sub-set of such fields that affects particular test path would be much smaller than the entire set. As I explained in the system description, the value of a state field depends on the input to the program i.e., input fields, command parameters and sensor inputs. So, finding those internal and external variables that affects each transition would be promising to reduce the search space.

To identify those influential variables in a systematic way, one would do a one-time static analysis on the source code. A static analysis strategy that applies in such scenario is called program slicing [82]. 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. A slice of the program is computed based on a set of variables or statements in the program. A slicing criterion is defined as $S = (l, V)$, where l is a statement and V is a set of variables. The slice includes all the statements that might influence the values of the variables in V in the statement l. A slice of a program is independently executable and shows the same behaviour as the original program. This technique is usually used in debugging to locate source of errors in a program. It is also applied in software maintenance, optimization, program analysis, and information flow control. An example of program slicing is shown in table 7.4. In this example, the left side of the table shows a small program where we want to do a dependency analysis of the variable $totalItem$. A backward program

Table 7.4: An example of program slicing.

Sample Code Block	Slice at *write*(*totalItem*)

`int i;`
`int cost = 0;`
`int totalItem = 0;`
`int discount = 0.15;`
`for(int item = 1;`
`item < cart.size();`
`item++) {`
`quantity = cart[item][0];`
`unitPrice = cart[item][1];`
`cost += quantity*unitPrice;`
`totalItem += quantity;`
`}`
`cost = cost * (1 - discount);`
`write(totalItem);`
`write(quantity);`
`write(discount);`
`write(cost);`
	`int totalItem = 0;`
	`for(int item = 1;`
	`item < cart.size();`
	`item++) {`
	`quantity = cart[item][0];`
	`totalItem += quantity;`
	`}`
	`write(totalItem);`

slicing tool would statically analyze the program in a bottom-up fashion and return a slice of that program that might affect the value of the variable *totalItem*. The output slice of this example is shown in the right side of the table. Based on the slice, it is easy to extract all the variables that *totalItem* depends on.

Since, the autopilot is developed using C language, I looked for existing program slicing tool for C shown in the table 7.5.

The Wisconsin Program Slicing Tool [83] 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.

Table 7.5: Program Slicing Tools for C.

SL.	Tool	License Type	Platform	Last Updated
1	Wisconsin Program-Slicing Tool	Commercial	SunOS, Unix	Not Available
2	The Unravel Project	Open source	SunOS, Unix	July 1996
3	Frama-C	Open source	Windows, Linux	Up to Date

The Unravel [84] is an open-source program slicing tool for C that runs under UNIX environment. As pointed by [85], this tool does not support several recent extension 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 [86]. I investigated this tool comprehensively and tried to apply it on our System Under Test (SUT) to analyze whole program effect. 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) [87] 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.

So, to apply such program slicing tool, it requires lot of modifications to make it compatible with the system under test which could be a separate master's thesis.

Being unable to apply program slicing tools to reduce the search space, I have manually analyzed and extracted the source code to find all the potential internal and external variables that can affect each test path and left the automation of this task for future work.

7.7.2 Reducing Execution Time by Preemption.

In the simulator one hour of real flight takes several minutes based on the computer configuration and the hardware type of the UAV. The simulation that runs on a highly configured development machine (Intel Corei7 CPU 3.40 GHz 16 GM RAM 64bit OS x64-based processor) takes about 500 seconds with a minimum amount of flight plan (i.e., $takeoff$ and $circuit$). The system under test is a safety-critical system. 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 applying search algorithms to automate the test data generation process. To reduce the execution time without affecting the usual behavior of the autopilot and the simulator, I have investigated the collection of state fields (described in the section 4.1).

Since, the simulator can monitor what flight command has been executed in the autopilot at a certain time, I use this opportunity and modify the test code to stop the

simulation whenever it finishes executing the flight command of interest. Therefore, if I am testing the behavior of the $takeoff$ procedure I do not need to wait for the execution of landing procedure. To give it a more perspective, the same simulation on a development machine (Intel Corei7 CPU 3.40 GHz 16 GM RAM 64bit OS x64-based processor) takes about 23 seconds with a minimum amount of flight plan (i.e., $takeoff$ and $circuit$) if I am testing only the $takeoff$ flight command. Thus, it reduces the execution time to a great extent depending on what flight command is under test.

Chapter 8

Results

In this chapter, in section 8.1, I present the results of two search algorithms- 1+1 evolutionary algorithm and random when generating test input data. To evaluate the effectiveness of our model-based testing tool, I have conducted an empirical study which is presented in section 8.2.

8.1 Search Results

I have applied (1+1) Evolutionary Algorithm to generate test data described in section 7.4 for a sample anonymized state machine of the system under test shown in figure 8.1. I run (1+1) Evolutionary Algorithm and random search in a highly configured development machine (Intel Corei7 CPU 3.40 GHz 16 GM RAM 64bit OS x64-based processor).

As described in section 7.3, MBT generates one new test case for each MC/DC combination. Our sample state machine contains 7 test paths shown in table 8.1

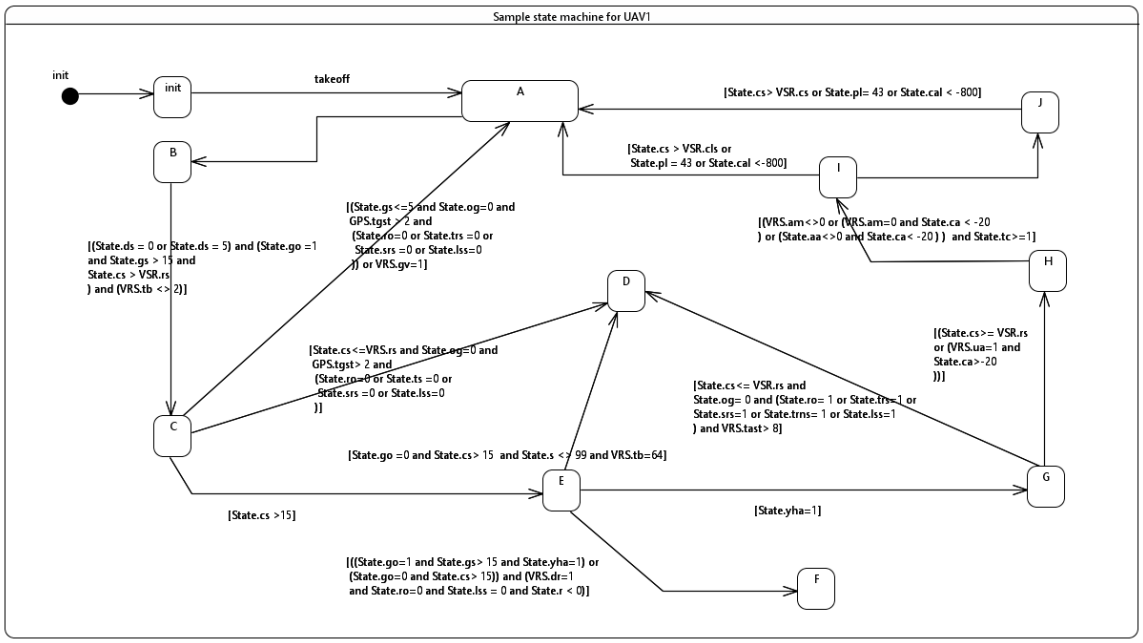

Figure 8.1: A sample anonymized state machine of the system under test.

Table 8.1: Test paths of the sample state machine 8.1.

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**

including 12 states, 16 transitions and total 384 combinations. Each of those combinations is a target that needs to be satisfied by a set of input values using the search algorithm. While evaluating an individual for a new target, I not only evaluate that particular target but also all the other targets. This approach is known as the whole test suite generation approach [88]. This approach is efficient since an individual

might be fit for more than one target. I have also applied the Random search algorithm to compare the performance of Evolutionary Algorithm. Based on the search result, there are many targets that are satisfied by the search algorithm. There are also several targets that are not satisfied by the search algorithm. I have analyzed all the conditions that are not covered and categorized all targets based on the reason for not being covered. The categories are described as follows,

1. Satisfied: The search algorithm successfully managed to find solution regarding such targets.

2. Not Satisfied: There are conditions that cannot be satisfied due to the existence of bug or limitations. I have further categorized these types of conditions as follows.

   I Buggy. Conditions that can not be satisfied due to bug in the code. The condition is covered and 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 where 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 understandably 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 is achievable but the improper simulation prevents it to justify as a solution.

IV Limitation of our MBT Tool. There are some conditions that cannot be satisfied based on the current approach MBT, where we, like most other MBT works, 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 changed anytime during the execution. Therefore, if there exist opposite conditions on a path that depends on 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 as the AutoPilot and have the two models runs concurrently, which was outside of the scope of this thesis.

V Limitation of the search algorithm and budget. There is a 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 generate test cases from the entire state machine, I have given the same amount of budget to each algorithm. Evaluating an individual is time consuming since it involves integrating the test execution framework and running the simulator. Due to the lack of budget, I have provided 200 maximum simulator executions to satisfy a target. With this budget, the evolutionary algorithm took 26,210 executions to finish generating test cases from the sample state machine which took 145 hours. I have given the random search algorithm the same number of executions (i.e., 26,210)

to finish generating test cases from the sample state machine. The random search took 149 hours to finish the same number of executions. In the search space, there are variables that might cause execution error if some particular values are assigned to them. Any individual that causes such error cannot be evaluated and a high value, typically 100 is assigned to them as the fitness value. Since, the random search is not guided, it might produce more such erroneous individuals then the evolutionary algorithm. Moreover, such erroneous individuals take more time to finish the execution. Table 8.2 and 8.3 show the results of 1+1 evolutionary algorithm and random search respectively. The performance of these algorithms are the same except the path 5,6, and 7. With the same amount of budget, evolutionary algorithm has finished trying all the targets but the random search has three unattempted (uncovered) targets in path 7, although those three targets are not satisfiable due to the infeasibility and the limitation of the simulator. In path 6, there is one less target categorized as infeasible in random search than evolutionary algorithm. That particular target contains an infeasible condition along with other feasible conditions. However, the random search failed to satisfy the feasible conditions as well, so I have categorized that target as "Uncovered by Random Search". Similarly, in path 5, 2 targets are categorized as "Uncovered by Random Search". However, in path 5, there is one target satisfied by evolutionary algorithm but not by random search. In summary, evolutionary algorithm has satisfied one more target (250 targets out of 384) and attempted covering seven more unsatisfied (due to the infeasibility and limitation of the simulator or our MBT tool) targets than the random search.

Table 8.2: Result of 1+1 Evolutionary Algorithm with budget 26,210.

Path#	Targets	Bug	Inf.	Limitation of		Uncovered 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 8.3: Result of Random search with budget 26,210.

Path#	Targets	Bug	Inf.	Limitation of		Uncovered 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

8.2 Analysis of the Detected Bugs

The effectiveness of the generated tests is usually measured by its capability for detecting bugs. The bugs are detected when the behaviour in the state machine does not match with the actual behaviour of the code. Our MBT approach emphasizes on testing each of the conditions in a constraint individually so it is more demanding than a typical MBT. Using this approach, from the sample state machine of the system under test shown in figure 8.1, the proposed approach has detected new bugs and wrong specification. In total, the tool has found one fault regarding autopilot configuration, and two faults regarding incorrect code implementation. Note that the legacy codes that are found, are not categories as bugs but they are in the infeasible category. Below is a high-level analysis of the detected bugs (the details are omitted due to confidentiality reasons):

1. 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 VRS fields. These configuration fields are also exposed to the users. A wrong configuration will result in a bug. To be exact, in the above state machine the condition $VRS.gv = 1$ is wrong.

2. Bug 2: Incorrect transition constraints. For instance in our case the conditions from State C to A and C to D are wrong since the code implements the condition $State.tgst > 0$ instead of $State.tgst > 2$.

3. 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 $State.tast > 8$.

Although, I have used only one sample state machine to test 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.

8.3 Improvement In Code Coverage

One of the objectives of this research was to improve code coverage to help for certification. In section 7.1 and 7.2, I have described Condition/Decision (C/D) coverage and Modified Condition and Decision Coverage (MC/DC) respectively. Our approach tries to improve code coverage by targeting high-level model constraints in an MC/DC manner. As presented in the previous section, the MBT tool has generated test cases for 250 targets out of 384 targets. Therefore, it has 65.1% MC/DC coverage of model constraints.

In the sample state machine, there are 7 test paths. In tradition model-based testing approach, there would be one test for each of those test paths. i.e., 7 test cases in total. But our approach generates 384 test cases for those 7 paths including those main 7 test cases. I measure the code coverage of those 7 test cases and the entire test suite separately using the Bullseye Coverage [81] tool. The Bullseye Coverage [81] measures the coverage in terms of function coverage and C/D coverage. The table 8.4 shows the measurement of code coverage before and after running

the entire test suite. It also shows coverage of all the dependencies of the system under test. As the results show, the function coverage improves from 26% to 28% and the Condition/Decision (C/D) improves from 19% to 28%. In other words, our approach improves the function and C/D coverage by 7.6% and 47.3%. The sample state machine of the system under test deals with only one flight command (i.e., $takeoff()$) of the system and does not go beyond that command, which reduces the scope of improvement in overall function coverage.

To do a comprehensive analysis on code coverage of that particular flight command, I have measured the code coverage of the corresponding source code functions of the fight command. Each high-level flight command is mapped to a code-level function. For example, the flight command $takeoff$ is mapped to a code-level function $takeoff()$. Table 8.5 shows the internal C/D coverage of that function before and after running the new test suite. As the table shows, the test suite generated by our approach covers 14 more conditions and decisions (C/D) than the test suite generated by the traditional approach. However, there are still 114 C/Ds left to cover. With the help of domain expert, I have investigated the reason behind this low C/D coverage of the function $takeoff()$. Some of the reasons are given below.

- The same $takeoff()$ function is used in different types of UAV such as fixed-wing, multi-rotor, vtol and blimp and each UAV has different configuration. In our experiment, I have generated test cases only for only fixed-wing UAV.

- If the state machine is too abstract and represents very high-level specification then the test cases generated from it may miss some low-level conditions.

- There are several conditional statements in the code that include internal vari-

ables that are not exposed to the simulator. Therefore those branches of the code cannot be covered with the system-level test cases.

- The code implements some infeasible conditions that come from legacy code that are no longer in action.

- There are several constraints in the model that contain arithmetic operators. However, the company's test execution framework does not allow any arithmetic operator in any side of a condition. Due to this limitation I skipped generating test cases to test those constraints. The company is planning to add this feature to their framework.

- As described in section 8.1, due to the limitation of the simulator and MBT tool, there are still 131 model-level targets left to cover by the search algorithm. Therefore, the code-level conditions corresponding to them are missing as well.

Therefore, exposing all the important system variables and reducing the limitation of the simulator, MBT tool and test execution framework, and removing infeasible code implementations would certainly improve the code coverage of the test suite.

Table 8.4: Code coverage of the original vs. improved MBT (with model-level MC/DC coverage).

Directory	Function Coverage					Condition/Decision Coverage				
	Total	Original		Improved		Total	Original		Improved	
		Covered	%	Covered	%		Covered	%	Covered	%
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%**

Table 8.5: C/D coverage of code-level $takeoff()$ function.

Function	Before	Uncovered C/D	After	Uncovered C/D
takeoff()	50%	128	55%	114

8.4 Lessons Learned

While implementing our proposed Model-Based Testing tool and applying it to generate system-level test cases for an avionic system, I have learned some important lessons regarding system modeling and test generation. In this section, I describe the lessons I have learned.

1. **System Modeling.** Modeling the system is always a crucial part of the model-based testing. It is a prerequisite of model-based testing. Modeling the system behaviour precisely using UML-2 standard is the first step to apply our tool for test automation. However, modeling a system requires understanding of both UML specifications and system domain. Therefore, the required skills for test engineers to model the system are much higher than for test engineers writing sequential test procedures. Since the specification models need to be prepared before the test automation starts, this induces a delay for the proper test execution. Our System Under Test (SUT), an avionic system, is very complex. For such a complex system, test models need to be abstracted from a large amount of details; otherwise, the test models and the tests would become unmanageable. Moreover, the lack of formal specification in the industry makes the system modeling a labor-intensive task.

   I have also learned that modeling the system can reveal details about the code that the domain experts are not aware of. For example, legacy code that is still sticking around.

2. **Applying Search Algorithms.** In a complex avionics systems, applying a

search algorithm can be extremely expensive if all the relevant input system variables of the subsystem are not identified. A simple simulation of such a system must include proper takeoff and landing procedures which drastically increase the execution time as well. Consequently, a search algorithms that needs to evaluate so many test cases to find an optimum solution will be infeasible due to time constraints. In this study, I manually did the dependency analysis and excluded irrelevant input variables from the search space but ideally one needs to automate this task as well.

I also realized that, in an industry setting, limitations of a test generator is only one challenge among many more (such as limitation of tools and frameworks and infeasible paths due to legacy code, etc.) for low coverage or missing faults.

Chapter 9

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. To obtain safety certification, the testing must demonstrate that the software satisfies all of its requirements with a high level of confidence that no error will lead to unacceptable failure condition as determined by the safety assessment processes [3]. Model-based testing can dramatically increase the testing effectiveness. In addition, it provides a way to map requirements to the test cases, which is crucial for locating system defects. Moreover, the test suite must provide evidence of high code coverage that shows the verification of code-base against the requirement.

To improve the size and effectiveness of the test suite in terms of code coverage, traceability and bug detection capability, I have developed a Model-Based Testing (MBT) tool for an industrial autopilot system and introduced it to that industry that has never used MBT before.

In our MBT tool, I have introduced a new approach that maximizes model-constraint coverage in terms of MC/DC and generate system-level test cases which consequently increase low-level code coverage. Our experiment on a sample state machine model shows, the new approach improves the code coverage (C/D) significantly. The new technique enables comprehensive testing of the system under test and detects real faults in the system as well as legacy specifications that are no longer in use. Our experiment with the sample state machine is an example of how the tool can be used effectively to detect bugs and improve code coverage.

To further improve the tool I have also thought of following future research directions.

1. Systematic and Automatic Dependency Analysis. Reducing the search space requires a dependency analysis (e.g., program slicing) of the system variables that are in the model constraint. The reason of such dependency analysis is to find a small set of input system variables that affects the behavior of a particular transition in a test path.

2. Minimizing Limitations. The model-based testing works on top of the simulator of the system under test and the test execution framework. Any limitation of those dependencies limits the capability of the MBT tool as well. Moreover, changing the strategy of simulating the external inputs by the MBT tool might overcome it's limitation. To reduce such limitations, those dependencies need to undergo some modifications, for example, improving system-level test representation to incorporate complex logical expressions in the test code. Therefore, it requires efforts from both the industry and the researchers.

Bibliography

[1] Nigel James Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, Citeseer, 2000.

[2] Automated Flight Control. In *Advanced Avionics Handbook*, chapter 4. Federal Aviation Administration, 2009.

[3] DO-178c Software Considerations in Airborne Systems and Equipment Certification. *RTCA Inc.*, 2011.

[4] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[5] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85–103, May 2007.

[6] LDRA, LDRA Software Technology. `http://www.ldra.com/en/ Last accessed 22 May 2017`.

[7] Tatiana Mangels and Jan Peleska. Ctgen-a unit test generator for c. *arXiv preprint arXiv:1211.6191*, 2012.

[8] A.E. Berner, J.R. Bucher, L.E. Holliday, and H.C. Murray. Aircraft flight emulation test system, November 9 1993. US Patent 5,260,874.

[9] Micropilot Inc. `http://micropilot.com`, Last accessed 1 January 2017.

[10] Eclipse Model To Text (M2T). The Eclipse Foundation. `https://eclipse.org/modeling/m2t` Last accessed 1 January 2017.

[11] Papyrus Modeling Environment. The Eclipse Foundation. `https://eclipse.org/papyrus` Last accessed 2017-01-01.

[12] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[13] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.

[14] Jan Peleska. Industrial-strength model-based testing-state of the art and current challenges. *arXiv pre-print arXiv:1303.1006*, 2013.

[15] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart. Test data generation from uml state machine diagrams using gas. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 47–47, Aug 2007.

[16] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated*

Software Engineering, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[17] Kenneth Koster and David C. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 541–544, New York, NY, USA, 2007. ACM.

[18] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 25–36, New York, NY, USA, 2006. ACM.

[19] A. Andrews, S. Elakeili, and S. Boukhris. Fail-safe test generation in safety critical systems. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 49–56, Jan 2014.

[20] James C King. Symbolic execution and program testing. *Communications of The ACM*, 19(7):385–394, 1976.

[21] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

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

[23] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[24] John H Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992.

[25] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

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

[27] Harmen-Hinrich Sthamer. *The automatic generation of software test data using genetic algorithms*. PhD thesis, University of Glamorgan, 1995.

[28] Marc Roper. Computer aided software testing using genetic algorithms. *10th International Quality Week*, 1997.

[29] Bryan F Jones, H-H Sthamer, and David E Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[30] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[31] Grady Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[32] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box

method. In *11th Asia-Pacific Software Engineering Conference*, pages 284–291, Nov 2004.

[33] M. Sarma, D. Kundu, and R. Mall. Automatic test case generation from uml sequence diagram. In *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pages 60–67, Dec 2007.

[34] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 86–91, New York, NY, USA, 1993. ACM.

[35] Taylor L Booth. *Sequential Machines and Automata Theory*, volume 3. Wiley New York, 1967.

[36] M. Aggarwal and S. Sabharwal. Test case generation from uml state machine diagram: A survey. In *2012 Third International Conference on Computer and Communication Technology*, pages 133–140, Nov 2012.

[37] Fredrik Abbors, Veli-Matti Aho, Jani Koivulainen, Risto Teittinen, and Dragos Truscan. *Applying Model-Based Testing in the Telecommunication Domain*, page 487524. CRC Press, 2011.

[38] E. Bringmann and A. Krmer. Model-based testing of automotive systems. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 485–493, April 2008.

[39] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, (4):367–375, 1985.

[40] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[41] Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo-Hwan Bae, and Hasan Ural. A test sequence selection method for statecharts. *Software Testing, Verification and Reliability*, 10(4):203–227, 2000.

[42] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings. 26th International Conference on Software Engineering*, pages 86–95, May 2004.

[43] P. Chevalley and P. Thevenod-Fosse. Automated generation of statistical test cases from uml state diagrams. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 205–214, 2001.

[44] Philip Samuel, R Mall, and Ajay Kumar Bothra. Automatic test case generation using unified modeling language (uml) state diagrams. *IET software*, 2(2):79–93, 2008.

[45] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, pages 188–195, Sept 2007.

[46] Lionel C Briand, Yvan Labiche, and Jim Cui. Automated support for deriving

test requirements from uml statecharts. *Software & Systems Modeling*, 4(4):399–423, 2005.

[47] Ranjita Swain, Vikas Panthi, Prafulla Kumar Behera, and Durga Prasad Mohapatra. Automatic test case generation from uml state chart diagram. *International Journal of Computer Applications*, 42(7):26–36, 2012.

[48] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using uml statechart diagrams. In *Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*, SAICSIT '03, pages 296–300, Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[49] IBM Rational Rose. IBM Inc. `http://www-03.ibm.com/software/products/en/enterprise`, Last accessed 1 January 2017.

[50] Mahesh Shirole, Amit Suthar, and Rajeev Kumar. Generation of improved test cases from uml state diagram using genetic algorithm. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 125–134, New York, NY, USA, 2011. ACM.

[51] Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C Briand. Generating test data from ocl constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402, 2013.

[52] Zoltn Micskei. Model-based testing (MBT). 2016.

[53] Muhammad Shafique and Yvan Labiche. A systematic review of model based testing tool support. *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04*, pages 01–21, 2010.

[54] Intel. fMBT. Intel Inc. `https://01.org/fmbt` Last accessed 1 January 2017, 2012.

[55] Kristian Karl. Graphwalker. GraphWalker. `www.graphwalker.org` Last accessed 1 January 2017, 2013.

[56] Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipate. JSXM: A Tool for Automated Test Generation. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 352–366, Berlin, Heidelberg, 2012. Springer-Verlag.

[57] Dianxiang Xu, Weifeng Xu, Michael Kent, Lijo Thomas, and Linzhang Wang. An automated test generation technique for software quality assurance. *IEEE Transactions on Reliability*, 64(1):247–268, 2015.

[58] Axel Belinfante. JTorX: A Tool for On-line Model-driven Test Derivation and Execution. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 266–270, Berlin, Heidelberg, 2010. Springer-Verlag.

[59] C. Artho, M. Seidl, Q. Gros, E. H. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata. Model-Based Testing of Stateful APIs with Modbat. In *2015*

30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 858–863, Nov 2015.

[60] Mark Utting. How to design extended finite state machine test models in java. In *Model-Based Testing for Embedded Systems*, pages 147–169. CRC Press, 2011.

[61] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, and H. Brandl. MoMut::UML Model-Based Mutation Testing for UML. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, April 2015.

[62] Teemu Kanstrén and Olli-Pekka Puolitaival. Using built-in domain-specific modeling support to guide model-based test generation. In *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*, pages 295–319. Apple Academic Press, 2014.

[63] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. *ACM SIGSOFT Software Engineering Notes*, 29(4):129–132, 2004.

[64] FOCUS. IBM. `http://researcher.watson.ibm.com` Last accessed 1 January 2017.

[65] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, et al. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.

[66] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio

Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[67] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 264–, April 2004.

[68] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Victor Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, A-MOST '07, pages 1–11, New York, NY, USA, 2007. ACM.

[69] Nikolai Tillmann and Jonathan De Halleux. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[70] SpecExplorer. Microsoft Inc. `https://msdn.microsoft.com/en-us/library/ee620411.aspx` Last accessed 1 January 2017.

[71] V. Vishal, M. Kovacioglu, R. Kherazi, and M. R. Mousavi. Integrating Model-Based and Constraint-Based Testing Using SpecExplorer. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pages 219–224, Nov 2012.

[72] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Con-*

ference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[73] Gregory Zoughbi, Lionel Briand, and Yvan Labiche. Modeling safety and airworthiness (rtca do-178b) information: conceptual model and uml profile. *Software & Systems Modeling*, 10(3):337–367, 2011.

[74] Gregory Zoughbi, Lionel Briand, and Yvan Labiche. A uml profile for developing airworthiness-compliant (rtca do-178b), safety-critical software. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, MODELS'07, pages 574–588, Berlin, Heidelberg, 2007. Springer-Verlag.

[75] Miguel A De Miguel, Javier Fernández Briones, Juan Pedro Silva, and Alejandro Alonso. Integration of safety analysis in model-driven software development. *IET software*, 2(3):260–280, 2008.

[76] K Hansen and Ingolf Gullesen. Utilizing uml and patterns for safety critical systems. *Jürjens et al.[JCF+02]*, pages 147–154, 2002.

[77] Jan Jürjens. Developing safety-critical systems with uml. In *International Conference on the Unified Modeling Language*, pages 360–372. Springer, 2003.

[78] Heiko Stallbaum and Mark Rzepka. Toward do-178b-compliant test models. In *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on*, pages 25–30. IEEE, Nov 2010.

[79] Tim Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.

[80] First Needham Place. Object management group. *mars*, 2005:06–12, 2000.

[81] Bullseye Coverage. Bullseye Coverage Technology. `http://www.bullseye.com` `Last accessed` 1 January 2017.

[82] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[83] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[84] James R Lyle, Dolorres R Wallace, James R Graham, Keith B Gallagher, Joseph P Poole, and David W Binkley. 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.

[85] James R. Lyle and Dolores R. Wallace. Using the unravel program slicing tool to evaluate high integrity software. In *In Proceedings of 10th International Software Quality Week*, 1997.

[86] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Form. Asp. Comput.*, 27(3):573–609, May 2015.

[87] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer.

CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[88] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.