Generating Effective Test Suites for Reactive Systems using Specification Mining

Presented By

Prasad Ramesh Bokil

Submitted in total fulfilment of the requirements of the degree of

Masters By Research

Faculty of Business

Bond University

The Australia

January 6, 2014

Abstract

Failures in reactive embedded systems are often unacceptable. Effective testing of embedded systems to detect such unacceptable failures is a difficult task.

We present an automated black box test suite generation technique for embedded systems. The technique is based on dynamic mining of specifications, in the form of a finite state machine (FSM), from initial runs. The set of test cases thus produced may contain several redundant test cases. Many of the redundant test cases are then eliminated by an aggressive greedy test suite reduction algorithm to yield the final test suite. The tests generated by our technique were evaluated for their effectiveness on five case studies from the embedded domain. The evaluation of the results indicate that a test suite generated by our technique is promising in terms of effectiveness and scales easily. Further, the test suite reduction algorithm may sometimes remove non-redundant test cases too. Therefore, in our experimentation, we have also evaluated the change in the effectiveness of test suites due to this reduction.

In this thesis, we describe the test suite generation and reduction technique in detail and present the results of the case studies.

This thesis is submitted to Bond University in fulfilment of the requirements of the degree of Masters by Research. This thesis represents my own original work towards this research degree and contains no material which has been previously submitted for a degree or diploma at this University or any other institution, except where due acknowledgement is made.

Prasad Ramesh Bokil
Faculty of Business
Bond University
Robina 4229
Australia

Acknowledgment

I would like to express my gratitude to my supervisors Prof. Padmanabhan Krishnan, Prof. Marcus Randall and Mr. R. Venkatesh for the useful comments, remarks and engagement through the learning process of this master thesis. I would also like to thank my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

Table of Contents

List of Figures

List of Tables

1. Introduction

Certain software in embedded systems, like avionic and automotive systems, is *safety-critical*. Examples of such software are flight navigation, autopilot in avionics and braking system and airbags deployment in automotive systems. Failures in these systems risk loss of life and property. No *safety-critical* software should have critical errors, else they may manifest into unacceptable failures. Preventable losses have been observed due to software failures [59], [37], [20]. However, preventing or detecting critical errors in embedded systems is a difficult task [14].

Attempts have been made to detect errors in software, both by software analysis and by software testing. We discuss some of these attempts in detail in the following sections.

1.1 Analysis for error detection

Errors may get detected at various stages of a software development life cycle - requirement specification, design and implementation. We list the predominant analysis methods and mention their limitations for industrial embedded systems.

Requirement analysis includes the task to detect errors and inconsisten-

cies in the requirements. This prevents percolation of errors into the design phase and beyond. Requirement analysis may include validating the requirements against expected behaviour and detecting ambiguous requirements. There exist tools capable of detecting errors in requirements like QUARCC [9], QUARS [24] and KaOS [83]. However, these tools are not adequate for industrial software [48]. In most projects, requirements are present in non-executable informal notations [48]. So requirement analysis is manual, which makes the error detection task time-consuming and error-prone. Also, the requirements keep changing, which adds to the complexity of the analysis [21].

Design analysis involves detecting errors in the design artifacts such as state-charts and UML models. Example of errors include infeasible designs, performance issues deviation from specifications, deadlocks and unreachable states in state-charts. Though advantageous, the basic limitation of design analysis is availability of a low level design that represents the implementation. Many a times, for legacy as well as other software, the designs are absent. Thus design analysis requires an effort to develop designs, which is not always put in [36]. With no designs, design analysis cannot be done. Even when designs are present, there are many cases where the design does not depict the actual implementation [62]. Last minute changes to the software is the primary cause of this, where changes are made to an implementation without updating the design artifiacts due to time constraints. In such cases, it is hard to perform any design analysis.

Program analysis aims at automatic analysis of program behaviour. The two main approaches in program analysis are: static analysis and dynamic analysis. Static analysis techniques inspect the code for errors without execution of the code [23]. They can help uncover a number of errors in the

code, like zero division, deadlocks and array index out of bounds. Tools and techniques implementing static analysis include PolySpace [61], ASTREE [13] and TECA [54]. These tools do scale up for large codes, but are imprecise [74]. The tools display a number of warnings which can be potential errors in the code. These warnings need to be manually verified for actual errors, which takes a lot of time and effort and the process is error prone. The problem of manual verification of warnings is compounded with increase in complexity of the software, as the number of warnings is directly proportional to size of code. As a result, static analysis has its limitation for use on industrial embedded software.

Dynamic analysis, on the other hand, relies on program execution to study the code behaviour, often using instrumentation. The effectiveness of such an analysis depends greatly on the sufficiency of test inputs, *i.e.*, it must be ensured that an adequate slice of program's set of possible execution behaviours have been observed. A practicable measure of this comes from software testing techniques such as *code coverage*. Dynamic analysis has the additional ability to find security issues caused by the code's interaction with other system components like SQL databases, application servers or Web services. Among its advantages over static analysis, the following are noteworthy:

- identification of vulnerabilities in a runtime environment.
- ability to analyse applications in which one does not have access to the actual code.
- identification of vulnerabilities that might have been false negatives in the static code analysis.

However, such an analysis is often too complex to work with. For instance,

it's quite difficult to trace a vulnerability back to the exact location in the code. Besides, one cannot guarantee the full coverage of the source code in dynamic analysis, as it is performed based on user interactions or automatic tests. Thus, no analysis at any level is sufficient for industrial embedded software.

1.2 Testing for error detection

There exists basic testing methods which are used to detect software errors that could lead to failures [63]. We mention the predominant testing techniques and their limitations for industrial embedded systems.

One of the best testing methods is exhaustive testing, which tests the software for all combination of inputs. This ensures tests which can detect all errors from the system. However, even for trivial programs, exhaustive testing is infeasible since it is time and effort intensive. Also, in most cases, it is not possible to generate an exhaustive test suite, like programs with unbounded loops. Thus other forms of testing are used. Requirement based testing is one of the primary forms of testing, where a test suite is prepared for checking the system based on the requirements of the system. Model based testing [5] is used when a model of the system is available. This model drives generation of test suite. Other types of test suite generation techniques include those that try to achieve metrics such as structural coverage [88] [31] [77], mutation killing [19], and so on.

We argue that none of the above techniques of testing are sufficient for embedded systems [55]. Requirement based tests may not exercise the complete code [53], because of which bugs may remain undiscovered. Model based testing has problems similar to design analysis. The models of software are

either absent or do not faithfully represent the implementation. Thus, model based testing may not give useful tests. Test suite generation techniques to achieve structural coverage suffer either from accuracy or scalability issues when run on large systems [57]. It is also noteworthy that coverage may sometimes be unsuitable to generate test suites [43], but this is often domain specific.

With none of these sufficing, there is a need for an effective testing technique for industrial embedded systems. We aim at developing one such technique that will help testing of these systems. For this, we choose to detect errors from program implementations. The rest of this paragraph justifies this choice of ours. Many a times, software programs are the only executable artifacts available with the team. Automated testing using non-executable artifacts is difficult. Non-executable artifacts will have to be made executable, which may require time and effort and the process is error-prone. Most requirements and design documents are either non-executable or they do not match with the implementation for reasons mentioned earlier. Software programs, on the other hand, are executable and thus their testing can be easily automated. Also, there are cases where the requirement and design is free from critical errors, but the implementation may contain errors. These errors may be introduced while converting design into implementation. Example of such errors are null pointer dereferencing and array index out of bounds. These errors are usually introduced by a programmer while coding. Thus, to improve the usability of the technique, we focus on error detection from software programs.

Static and dynamic program analysis are often used to automatically detect errors from embedded programs. Error conditions are encoded as properties and the analysis tools try to determine whether these properties

hold in the system. A different use of these analysis tools is to automatically generate test suites for the program which in turn would detect errors. In practice, these tools are useful on small programs. However, it is well known that for most industrial applications, due to the size and complexity of programs, static analysis tools are imprecise and dynamic analysis tools are un-scalable [57].

Software testing methods are traditionally divided into **white-** and **black-box** testing. The difference is in the point of view that a tester takes while designing the test cases. White-box testing focuses on testing the internal structures or workings of a program, as opposed to the functionality exposed to the end-user. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. Black-box testing, in contrast, treats the software as a "black box", examining functionality without any knowledge of internal implementation. The tester is only aware of what the software is supposed to do, not how it does it.

While white-box testing can be applied at different levels of the software testing process, it is usually done at the *unit* level. Though this method of test design can uncover many errors and problems, it might not detect unimplemented parts of the specification or missing requirements. Black-box testing, on the other hand, can be applied to most levels of software testing: *unit*, *integration*, *system* and *acceptance*.

A notable limitation of black box testing comes from the possibility that coincidental aggregation of several errors may produce the correct response for a test case, preventing error detection. This makes it challenging to generate effective test cases. Moreover, the black-box testing method makes it difficult to design test cases (with just functional specifications) and may lead to redundant test cases. However, we prefer black-box testing for this

work because of the following advantages of it (Tanja E. Vos, et al.[84] discuss the relevance of such testing approaches in similar settings):

- The test is unbiased because the designer and the tester are independent of each other.
- The tester does not need access to the code; knowledge of any specific programming languages isn't required.
- The test is done from the point of view of the user, not the designer.
- Test cases can be designed as soon as the specifications are complete.
- It averts the need for program analysis which is often costly. As a result, the techniques scale better.

There are existing techniques for error detection of programs using black box analysis. Random testing is a simple black box test generation technique, which may be effective in some cases [38]. We aim to refine random testing using information about the behaviours of the program. We identify the set of input and output variables used by the program and then capture the functional (input-output) behaviour of programs. Hence our approach is predominantly black box that relies on the identification of these variables. Our technique is based on specification mining [89]. We extract partial specification of the system by observing behaviour of the system over a test suite. The specification is in the form of a Finite State Machine (FSM). This specification guides further test generation. The advantages of our technique are that it is precise and scalable in principle and requires no program analysis. On the downside, owing to the use of a black box approach, our technique may generate a larger test suite as compared to white-box techniques. Error detection becomes increasingly difficult with the increasing size of test suites

(the response for each test case needs to be computed and tested against an oracle). As a result, we need to reduce the test suite. While reducing the test suite, we need to ensure that the size of the test suite reduces without compromising on the error detection effectiveness of the test suite. There have been several efforts on minimization of test suites [40], [80]. Measures, such as test case similarity [35], have been introduced to identify relevant tests and to understand test suites better. To address the issue of test suite reduction, we also present a straightforward test suite reduction algorithm. This algorithm reduces the size of the test suite in the black box paradigm while trying to maintain test suite effectiveness. As explained by Mary Jean Harrold, et al. [75], automated test suite reduction does help in reducing number of tests, but suffers from loss in error detection of test suite. Our technique exhibits similar characteristics. Thus, there is a trade-off between the number of tests against which error detection from program is to be measured and the loss of effectiveness.

In evaluating the effectiveness of our approach, we use Modified Condition Decision Coverage (MC/DC) [53] as a measure of test suite effectiveness. While coverage directed test generation isn't the most effective, especially for avionics systems [78], the reader must note that we only use coverage as an effectiveness measure and not to guide the test suite generation. For embedded systems, MC/DC is a commonly used coverage criterion, especially for safety critical applications. Importance of MC/DC over other practical coverage criteria has been established by Kalpesh Kapoor, et al. [52]. Standards like DO-178B [2] for avionics and ISO-26262 [3] for automotive, mandate MC/DC for all its systems.

1.3 The problem statement

Specification mining by observing system behaviour isn't unexplored, though not for embedded systems. But the key question we wish to answer is the following:

Can black box techniques be as *effective* as white box techniques?

We look for an answer in the context of industrial systems in the embedded domain and use MC/DC as an effectiveness measure for the technique. To find the answer, we evaluated the effectiveness of the algorithm sketched below on a program P viewed as a black box with just an input-output relation.

1. Generate an initial test suite TS.
2. Extend TS to TS_1 using specification mining techniques leading to a sink-free FSM.
3. Reduce TS_1 to TS_2 by eliminating test cases that are not necessary for state coverage but that satisfy a *connectedness* criterion (*i.e.*, FSM is a connected graph but not a multigraph).

The main contribution of our work is a black box test suite generation technique for reactive embedded systems, which generates a *satisfactory* test suite. By *satisfactory*, we mean a test suite which is comparable in effectiveness with respect to a white box test suite and is acceptable for practitioners. We show the effectiveness and scalability of the technique via case studies.

While improvements to the algorithms need to be investigated, initial results are promising.

In theory, our black box test suite generation technique can be applied to any reactive program. The only prerequisite is that inputs, outputs that form the state and program executable are available. However, for simplicity of experimentation and evaluation, we have our analysis to reactive programs written in C. So for explanation purposes, we demonstrate our technique on a sample C program. Inputs, process and outputs at each stage of the technique are detailed.

Rest of the thesis is organised as follows. Chapter 2 presents the literature review. Chapter 3 introduces terms used in the thesis. Chapter 4 explains our technique of test suite generation using specification mining and test suite reduction technique in the black box domain. Chapter 5 reports the test suite evaluation process, experimentation process and results of experimentation. Chapter 6 concludes with summary and future work.

2. Literature Review

Effective testing of embedded software in a black box environment is a difficult task [6]. There is a need for tests that verify the software. However, manually preparing such tests is time consuming and error prone. As discussed later, current testing methods may not be sufficient for effective detection of bugs from these systems. Thus, there is a need for an automatic test generation method. Such a technique may generate effective, but a lot of tests. This highlights the need for test reduction as well.

Our literature review is divided into three parts. The first part presents work related to test generation. We give motivation as to why test generation is important and list the usual types of test generation techniques. Explanation regarding limitations of these test generation techniques with respect to our domain is provided. We discuss specification mining in detail, which is our approach of test generation. The second part gives insight on the test suite minimization techniques and argues about our choice of test suite minimization technique.

The aim of generating tests is to find errors in software. The generated tests must, therefore, be effective on the software. A metric to measure the test suite effectiveness needs to be decided. In the third part of this chapter, we look at several test suite effectiveness measurement techniques and explain

the motivation behind choosing code coverage as a measure for the current work.

In the next couple of sections, we focus on black box techniques for generation and minimization of test suites. Evaluation of test suite effectiveness is independent of the technique used for generation of test suite. Hence, we do not restrict ourselves to black box for measuring test suite effectiveness.

2.1 Test generation

In this section, we focus on test generation methods on software. Some of the methods are as follows:

Exhaustive test generation [30] is one of the most simplest test generation techniques, which, in the ideal case, guarantees generation of data to detect all possible errors of the software. This is possible for programs with finite number of all possible inputs. However, in practice, it is extremely hard to generate exhaustive test data for the system. Number of inputs to the program, datatype of the inputs, unbounded loops and reactive behaviour of the program make use of exhaustive inputs infeasible. Also, there exists the oracle problem, where there needs to be a check of the output of the program over a test case, with the desired output of the system. Being too effort and time intensive, very few projects have an automated way of checking this. Thus, all checkings are done manually. It is extremely time consuming and error prone to manually check the output of a program with its desired output. Thus, for almost all programs, exhaustive test generation is not useful.

Requirement based test generation helps produce test data based on requirements of the program. This type of inputs test the program for

their functionality. Robert M. Poston [70] gives a detailed explanation of an automated test generation depending on requirements of the system and its usefulness. However, requirement based tests may not exercise complete code [53], because of which critical errors in the program may not be discovered. Thus, just requirement based testing is insufficient.

Model based test generation [5] is used when the model of the software is available. This model drives generation of test suite. But, many a times, for legacy as well as other codes, the models are absent. Thus model based testing requires an effort to develop models, which is not always put into [36]. In the absence of models, model based testing cannot be done. Even when models are present, there are many cases where the model does not depict the actual implementation [62]. This is mostly because of last minute changes required in the software, when changes are made directly on implementation without updating the model because of time constraints. In such cases, model based testing may not give useful tests.

At code level, there are various **white box test generation techniques**, which are summarized by Jon Edvardsson [22]. Program analysis is one of the techniques used to generate a test suite [28]. A test suite generated to improve code coverage [31] [77] is often used in the industry. Test suites are also prepared to detect memory related errors [92], concurrency errors [56] like read-write race and deadlocks. Richard A. DeMillo, et al. [19] explain a technique for test generation for mutations of the program. Performance of most white box test generation techniques on small examples is appreciable, but may be unsatisfactory for large systems as they run into accuracy and scalability issues [57] [7] [8].

There are a few **black box test generation techniques** as well. A. A. Omar, et al.[67] give a survey on the black box test generation techniques.

We provide, below, a brief overview of some of the techniques listed there.

Random test generation is a trivial black box technique to generate a test suite. Dick Hamlet [38] discusses situations where random testing would be sufficient and would be an alternative to systematic testing. However, random testing is mostly unsystematic and there is no guarantee that it would catch the errors [32].

In *equivalence partitioning (EP)* [67], the input domain of a program is partitioned into a finite number of equivalence classes. Assumption is that, for all classes, a test of a representative value of a class is equivalent to a test of any other value of that class. *Boundary value analysis (BVA)* is similar to EP, with the constraint that values are picked at the boundaries of the classes. This helps testing at boundaries, where errors may be present. Stuart C. Reid [72] explains how BVA is better than EP and random testing on an avionic code.

However, EP and BVA require (mostly manual) identification of equivalence classes and generating tests within that classes. The determining factor of success of EP and BV is the quality of creation of equivalence classes. With manual effort involved, cost of test generation increases and this process becomes error prone, reducing the effectiveness of error detection.

Cause effect graphing [63] is a systematic technique for representing test cases as a combination of inputs. The test cases can point out ambiguities and incompleteness in the specification. Yet, the process can be difficult to apply in practice, because the complexity of applying the technique increases for large number of causes (distinct input or equivalence class of input) [68].

The condition table method [30] is a method in which a condition table is prepared by looking at program specification. Tests are prepared from combination of conditions relevant to the correct operation of the program.

Another such method is *the category partition method* [68], where tests are prepared by systematically decomposing the program specifications. Both these methods depend heavily on availability of the specification, which may not always be available. Also, manual effort is involved making the process costly and error prone.

Except for random testing, the above black box testing methods are not suitable for reactive programs. In reactive programs, inputs may appear at varying intervals and response of the system is determined by the previous state of the program. Thus, sequences of inputs are needed to increase the effectiveness of the test suites, which is not exhibited by any of the above techniques. Also, most black box techniques are specification based or require manual intervention, which limit the applicability of the techniques.

In a paper by Michael D. Ernst, et al.[39], a technique to improve test suite using operational abstractions has been mentioned, where a better test suite is derived from an existing test suite by checking the behaviour of each test case. This technique requires domain knowledge of the system, which may not always be available.

Test generation using specification mining is a relatively newer area related to test generation. We explain in detail the literature review regarding specification mining.

Specification Mining

Recently, there has been research in generating specifications by observing program behaviour, which has been loosely summarized by Andreas Zeller [89]. Techniques have been developed for specification mining on Object Oriented (OO) systems. Andreas Zeller, et al. [16] explain the *ADABU*

technique for capturing object behaviour models from JAVA code. SPY [27] is a technique to recover specification of a software component from the observation of its run-time behaviour. Similarly, Mayur Naik, et al. [64] demonstrate use of static and dynamic analysis for preparing specifications from JAVA programs.

The technique of specification mining has also been applied to test generation. TAUTOKO [15] is a tool to generate specifications from dynamic analysis of programs and then generate test data from the specifications. Michael D. Ernst, et al. [90] explain a combination of static and dynamic analysis has been used for automated test generation and authors claim it to be superior than TAUTOKO. All these approaches of specification mining are strictly for OO systems, while most of embedded systems do not have OO programs, but are rather coded in C language [73].

Similarly, for determining Application Program Interface (API) behaviour, specifications can be mined either from the program source code using static program analysis [17] [76] or from execution traces [79]. However, these techniques cannot be directly applied to embedded systems because of the difference of nature of the programs. To mine API behaviour, a 'set of valid APIs' is considered as a 'program state'. Most specification mining algorithms for API programs are based on this 'state' concept. Clearly, this concept of state (and thus even the algorithms) cannot be used for embedded programs.

There are techniques where knowledge of the program internals or domain is used for specification mining. In a paper by Fides Aarts, et al. [4], regular inference has been used to build a Finite State Machine (FSM) from program behaviour. Antti Kervinen, et al. [47] discuss a technique to generate test models from test cases using domain specific language to prepare the models. Gerard J. Holzmann, et al. [44] discuss a technique and a tool Modex,

to extract specifications from code, where code is annotated with specific statements to help in specification extraction. These techniques require the user to have a certain degree of knowledge about the system and also to make changes to the program/technique. This may not be possible at all times due to the time constraints in the project.

A number of authors suggest ways to prepare formal specification from programs. Patrice Godefroid, et al. [29] explain the automatic preparation of symbolic equations for testing x86 processor instructions. Using an exhaustive test suite, the authors are able to determine the behaviour of Arithmetic and Logic Unit (ALU) type instructions, by considering the instruction as a black box. Claire Le Goues, et al. [58] explain a process to generate specifications from code using code quality as its guide. Although it reduces the false positives in the specifications, it extracts specific patterns in code and not the entire specification. Also, as shown by Mark Gabel, et al. [25], techniques that match a specific pattern of the specification to all possible program component combinations, are NP-complete in its general form.

Thus, our literature review suggests that specification mining technique has not been applied for test generation of embedded reactive programs. In the following section, we look at options for :

- representation of extracted specifications and our choice as FSM,
- representation of *state* of FSM and our choice of output variable values as state, and
- test generation techniques using specification mining and our choice of technique.

Choice of representation of extracted specifications

Specification mining techniques use different representations for the specification, like equations, models and FSMs. Patrice Godefroid, et al. [29] represent the extracted specification using equations while Antti Kervinen, et al. [47] use models to denote the extracted specifications. However, a majority of the representation of extracted specification is a FSM [15], [90], [64], [17], [76]. Clearly, FSM is the preferred form of specification representation. The primary reason for choosing FSM is that the representation closely depicts the implementation. Also, FSMs can be executable, which help in automating processes over the FSMs. The FSMs can be expanded or contracted, which depict refinement and abstraction of the specification. Our target programs are reactive programs, which are usually represented as state-charts. Since state-charts are built on top of FSMs, reactive programs can be naturally represented as FSMs. Thus, like majority of the specification mining techniques, we also choose FSMs as our choice of specification representation.

Choice of *state* of FSM

The choice of *state* is an important consideration for specifications represented as FSM. Most specification mining techniques are for object oriented type of systems for which, *state* in the specification is the list of all available methods in the class. We explain with example, two *state* capture concepts present in the literature. Illustration is provided by an example from paper by Carlo Ghezzi, et al. [27], as in Listing 2.1.

1. *State* based on availability of methods

```
public class Stack  {
   public Stack()  { .. }
   public void push(String element) { .. }
   public void pop() throws Error { .. }
   public String top() throws Error { .. }
   public boolean isEmpty() { .. }
   public int size() { .. }
}
```

Listing 2.1: Example code to explain program state

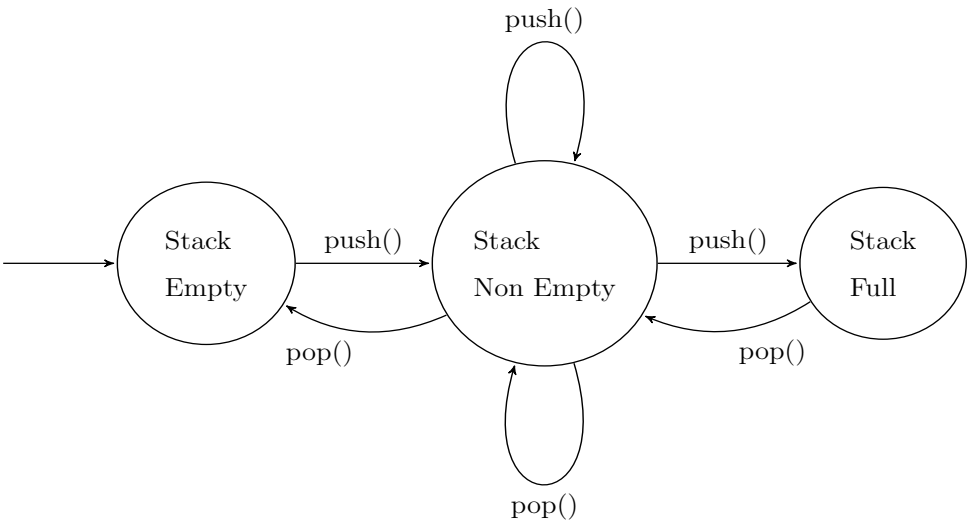

Figure 2.1: FSM example 1 explaining program state

The example code in Listing 2.1 implements a *stack* data structure. It has methods like push and pop. A sample FSM of the stack is represented in Figure 2.1. It shows four states depending on the working of the program. The main point to be noted is that each state represents the set of available methods present in that state. Thus, just assuming push and pop methods,

- the *Initial State* has the initialize method
- the *stack empty* state has just push () method available
- the *stack full* state has just pop () method available, while
- the *stack non empty* state has both pop() and push () methods available.

This type of *state* concept is predominantly used in object oriented systems. However, inferences from the example suggest that this concept of *state* is not possible for non-object oriented systems.

2. *State* based on values of method return

For the same example code, Listing 2.1, an alternate representation of *state* is as in Figure 2.2. This representation is used by Andreas Zeller, et al. [16]. In this representation, the return values of methods are used to form the state of the program. Again, this representation is not possible for non-object oriented systems, since class and methods do not exist. Thus, this representation cannot be applied directly for our technique. For our technique, we use a modified form of 'state based on values of method return'. We elaborate on our choice of *state* representation.

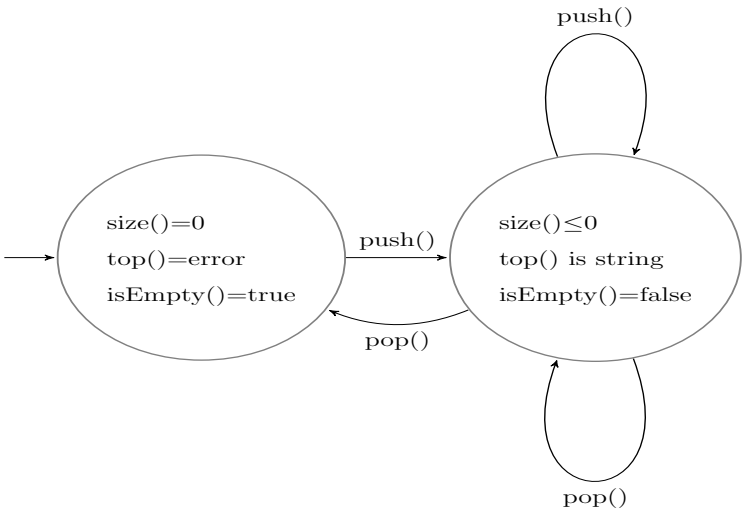

Figure 2.2: FSM example 2 explaining program state

In 'state based on values of method return' technique for OO systems, the methods are just an interface to retrieve values of members of the class. Thus, the values of members of the class determine the state of the program. These members are chosen to represent the state since the aggregation of values of members depict the information required to form the program state. Also, values of members are persistent, which means that they retain values over multiple method executions over the class.

For our imperative programs, we can use a similar concept of state. All variables in the program of imperative system are similar to members of the class of OO systems. Before execution of the program and class, variables and members will all be initialized to their default state respectively. During execution of the program, variables can indicate the current state of the program, similar to members of the class. With this correlation, we can modify the technique of 'state based on values

of method return' with values of variables as state for our technique.

The state of the program will be represented by variables which help to maintain the persistence of the program. For imperative systems like C, these can be the global and static variables of the program. Static variables can be converted to global using temporary variables. However, if global variables in the program never change their values (act like constants), then they may not represent the program state. These would be similar to members of a class whose values never change and thus do not contribute to state representation. As a result, we consider a subset of global variables, which are variables whose values change during execution of the program. As explained in 3.3, we call this subset of global variables as output variables of the program. Along with output variables, return value of program iteration is required to represent the state. This is the value returned by each iteration of the reactive loop in the program. This is explained in detail in 3.1 and 3.3. For simplicity, we call the return value of each iteration as a output variable as well. Thus, we use a representation in which the FSM state is represented by values of output variables of the system.

Choice of test generation technique

Studies suggest that, till now, TAUTOKO [15] is the most successful black box technique for test generation which uses specification mining techniques. Thus, we develop our technique in a similar way.

We discuss about TAUTOKO in detail. TAUTOKO [15] is a tool for OO programs to generate specifications from dynamic analysis of programs. The specifications are in the form of an automata and are used to generate test data to detect exceptions in the program. In this technique, the list of

all available methods is considered as state of the specification. TAUTOKO prepares a specification from a seed test suite, and enhances the specification by execution of all possible methods from every state of the program, thus generating a complete specification. In this process, it generates an effective test suite which detects exceptions in the object oriented programs. So, for a class in object oriented system, one can test for all sequence of methods in that class. A test case of this test suite consists of a sequential list of methods of the class, along with values.

Since TAUTOKO uses an exhaustive approach, the technique may not scale up for large programs. Also, we cannot borrow the concept of state of TAUTOKO for non object oriented embedded programs. The concept of *program state* is the list of all available methods at a particular execution state in the program. For TAUTOKO, some inputs cannot appear at certain times (like, a *pop* method cannot appear when the program state is *stackEmpty*). But, for reactive embedded systems, all inputs are equally likely to appear at all times. Our technique of test generation is derived from TAUTOKO, but is scalable in principle.

2.2 Test Suite Reduction

Developing a quick and efficient test suite reduction technique is hard. The optimal test suite reduction problem is an instance of set-cover problem which is NP-Complete [26].

Ideally, we want to generate a test suite which is minimal in size and is effective to detect all errors in the program. The test suite size should be small to tackle the oracle problem. At the same time, we do not want to loose any test suite effectiveness (*i.e.*, test coverage) due to test suite minimization.

As explained by W. Eric Wong, et al. [87], optimally minimizing test suite with respect to a criterion may lead to major reduction with negligible losses in test suite effectiveness. However, these results were contradicted by some studies [42] [51], who claimed that optimally reducing test suite for a criterion does decrease their effectiveness. However, on a space application, Eric Wong, et al. [86] demonstrated that significant test suite reduction can be achieved with little or no loss in test suite effectiveness. Thus, it is generally assumed in the research community that test suite minimization does not cause much loss in test suite effectiveness [49].

Work has been done on test suite reduction for regression testing [33], [46]. The test suite reduction for regression testing decreases the time required to re-test the software after changes are made to the software. When certain features of a software are modified, the entire test suite is executed over the software. This is to check if the output of the modified software is same as per the requirement. However, one can execute only a subset of the test suite and not execute those test cases which do not execute any modified part of code. This saves time and effort in executing the tests. Regression testing of high-assurance software is particularly expensive, such as software that is produced for airborne systems. One reason for this expense is the extensive verification required for the software. As quoted by Mary Jean Harrold, et al. [51], one of the company reports that for one of its products of about 20,000 lines of code, the MC/DC-adequate test suite requires seven weeks to run. Test suite reduction for regression can help solve this problem.

However, reduction methods for regression test suite cannot be used for our technique. This is because, these reduction methods reduce the effort of execution of unnecessary tests as opposed to discarding tests having no value. These techniques help choose test cases to temporarily ignore for execution

while we want techniques to completely delete the unwanted test case.

Most test suite reduction techniques are performed over some properties. Such properties include code coverage (like [45], [51]) and mutation analysis (like [66]). Test suite reduction for code coverage discards test cases which add no value to code coverage. Similarly, test suite reduction for mutation analysis discards test cases which do not increase mutation kill ratio. However, all these techniques are useful when program code is available. In the black box domain, code coverage based or mutation based test suite reduction algorithms cannot be used, program code is unavailable to perform any such analysis.

For black box, there are few test suite reduction techniques. A random technique is unusable as one can always delete the important test cases [34]. So random test suite reduction is rarely used. Most times, test reduction is based depending on some criteria. The HGS algorithm [41] is a test suite reduction technique, which reduces a test suite based on requirements. The HGS algorithm uses a greedy technique which selects the next test case which matches the most requirements. Mary Jean Harrold, et al. [41] showed that this technique reduces test suites without major loss of test suite effectiveness and the claim was supported [34]. Mats P. E. Heimdahl, et al. [42] explain another greedy technique to reduce test suites using models. This uses model checking techniques to reduce the test suites. Michael D. Ernst, et al. [39] propose a new technique for generating, augmenting, and minimizing test suites called the *operational difference technique*. This technique analyses program properties rather than program code. Jiang Zheng [91] explains a black box technique for selecting test cases for regression, based on documentation.

Of these techniques, we select a slightly customized version of the HGS

algorithm. The reason for selecting HGS algorithm is that it is easy to implement and has been proven to be effective. Most other techniques of test suite reduction require different prerequisites whereas our technique need the specifications, which have been generated by us.

2.3 Evaluation of test suite effectiveness

The effectiveness of a test suite is dictated by the number of errors detected by it. In order to get an absolute measure, one may look at the ratio of errors detected by a test suite to the total number of errors in a program. However, this ratio is not easy to arrive at, as the total number of possible errors in a program is rarely known. As a result, test suite effectiveness is measured in relative terms. If a test suite *TS1* finds more errors than another test suite *TS2*, then *TS1* is said to be more effective than *TS2*.

We would like to measure the effectiveness of the test suite generated by our technique. This would be measured on a relative basis with test suite generated using other techniques. For checking effectiveness, we need two versions of the program, one with errors and other with the errors fixed. Executing the test suites on the programs would show their effectiveness. However, coming up with this setup is hard because of the actual availability of such versions of a program. So we decide to use the other established methods to measure test suite effectiveness.

Jeff Offutt, et al. [60] explain a few alternate methods of test suite measurement. As per the paper, mutation testing and code coverage are effective forms of test suite effectiveness measurement. We discuss both methods of test suite effectiveness measurement and conclude on a metric.

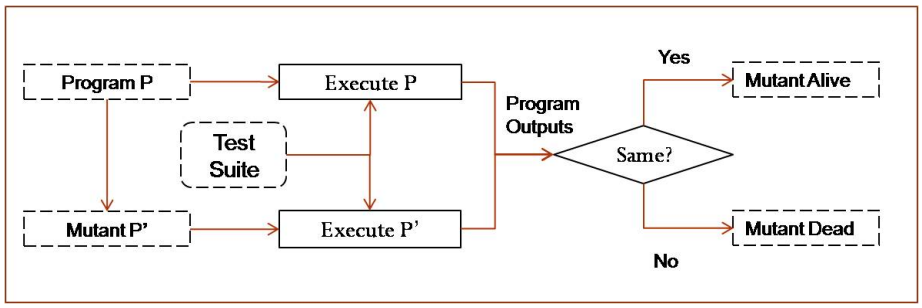

Figure 2.3: Mutation Testing

Mutation Testing

Mutation testing [65] can be used to measure quality of test suite for the given program. The flow diagram in Figure 2.3 attempts to illustrate the technique. In mutation testing, the source code of a program P is mutated (or modified) to create a program P'. The test suite of the program is executed over both versions of the program and their respective output is compared. If any test case in the test suite produces a different outputs for program P and P', the mutant is said to be killed (or detected). This means that the current test suite is good enough to catch a bug in the program where P' is the buggy version of the program. Similarly, mutant programs *P1*', *P2*', .. *Pn*' are automatically created based on some properties. Thus, Mutation Kill Ratio (MKR) is computed for a test suite which is the ratio of number of mutants killed by a test suite to the total number of mutants against which the test suite was run. A test suite *TS1* is considered to be more effective than *TS2* if MKR of *TS1* is greater than that of *TS2*.

We could have used MKR as a measure in evaluation of test suite effectiveness. With a C program and an input test suite, there are tools available for computing MKR, like Proteum [18] and MILU [50]. However, our experience with the tools indicates that these tools are inadequate when run for

industrial software. We tried to execute Proteum on our case studies, but the tool would throw an exception for industry code. Building an automated mutation analysis tool was beyond the scope of the thesis, since it was time and effort intensive. As a result, we could not use mutation testing for effectiveness evaluation of our test suites.

Code Coverage

A large number of coverage criteria have been defined for a variety of testing applications. Hong Zhu, et al. [93] illustrate some fundamental notions underlying these while presenting a comprehensive survey of various types of test adequacy criteria. For the sake of completeness, we introduce some of the most basic ones here: (For the ease of illustration of some of these criteria, we implicitly switch between the two equivalent notions - a program and its control-flow graph.)

- **Function coverage** - Has each function in the program been called?
- **Statement coverage** - Has each statement in the program been executed?
- **Decision coverage** - Has every edge in the program been executed, *i.e.*, have the requirements of each branch of each control structure been met as well as not met?
- **Condition coverage** (or *predicate* coverage) - Has each boolean subexpression evaluated both to true and false? This does not necessarily imply decision coverage.
- **Condition/decision coverage** - Have the decision and condition coverage been satisfied?

For safety-critical applications, we often look at a stricter criteria called **modified condition/decision coverage** (MC/DC). This criterion extends condition/decision criteria with the requirements that each condition should affect the decision outcome independently. For example, consider the expression 2.3.1 in a code:

```
if (a or b) and c then ..
```
(2.3.1)

The following tests satisfy the condition/decision criteria for the above statement: (`a=true, b=true, c=true`) and (`a=false, b=false, c=false`). However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of `b` and in the second test the value of `c` would not influence the output. Therefore, the tests needed to satisfy MC/DC are (`a=false, b=false, c=true`), (`a=true, b=false, c=true`), (`a=false, b=true, c=true`) and (`a=true, b=true, c=false`).

For embedded systems, MC/DC is a commonly used coverage criterion. Importance of MC/DC over other practical coverage criteria has been proven by Kalpesh Kapoor, et al. [52]. Safety critical standards like DO-178B [2] for avionics and ISO-26262 [3] for automotive mandate MC/DC for all its systems. Therefore, we use MC/DC as our code coverage criterion for checking effectiveness of test suites.

Similar to most structural coverage criteria, MC/DC of a test suite is measured as follows:

$$MC/DC = \frac{\begin{pmatrix}\text{Total number of conditions which}\\ \text{have showed independent effect}\end{pmatrix}}{\begin{pmatrix}\text{Total number of conditions}\\ \text{present in the code}\end{pmatrix}} * 100 \qquad (2.3.2)$$

We assume that, if test suite *TS1* attains more MC/DC than test suite *TS2*, then *TS1* is said to be more effective than *TS2*. The intuition behind the assumption is that more the MC/DC, more is the coverage of the code of the program.

In the next chapter, we introduce some of the basic terms and notions that will frequently appear in the rest of this thesis. We start with an informal definition of an embedded reactive system and introduce a few commonly used terms in software testing. Later, we get into the details of MCDC, a code coverage criteria that we employ to test the effectiveness of our technique, and briefly explain finite state machines.

3. Preliminaries

This section introduces some terms and notions which are used in this thesis.

3.1 Reactive embedded software

Most software in embedded systems is reactive in nature. The software accepts a set of inputs, processes them and produces a set of outputs before processing the next set of inputs. Figure 3.1 shows the interaction between a reactive system and its environment. The environment can be any entity which provides continuous inputs to the system. A practical approach to model-based testing of reactive embedded systems is to allow modeling of the environment to enable test automation [6]. Examples of environment are humans, sensor values and other similar systems. The environment provides input to the reactive system. The reactive system processes these inputs and generates outputs, which are sent back to the environment. The system will mostly maintain an internal state. This internal state will facilitate the system to decide on the output values. This state may note the number of times a particular input has occurred, last instance of an input, last instance of an output and the internal timer values. Once the outputs are presented to the environment, the reactive system will read the next set of inputs and

the process continues.

A typical example of a reactive embedded software is a wiper control of a car. The system is supplied with inputs to operate the wiper. This input can be provided either by a human (by adjusting wiper setting), sensor values (rain sensor) or the car itself (if ignition is on). An example input can be to operate the wiper at medium speed. The wiper control system processes the inputs and performs calculations depending on its internal state. For example, the input state may be that the wipers are off. In this case, it needs to start the wiper operation and increase its speed. It may also happen that the wiper is already running at high speed, where in the wiper speed needs to be reduced. Thus, the wiper control module computes the wiper operation and produces an output. As soon as the output is produced, the wiper control system waits for the next set of inputs to work on.

There are two assumptions about reactive systems. Firstly, it is assumed that a reactive system operates instantaneously, that is without any time delay. In practice, any reactive system takes a non-zero time for its operation, which is acceptable in practice. Secondly, it is assumed that reactive systems will run for a relatively long time, depending on its deployment. For example, it is expected that the automatic door operation system of a car is in operation for entire lifetime of the car. This makes it important to test reactive systems for long sequences of input, so that some defects can be detected. A peculiar feature of reactive embedded systems is that any of the available inputs can occur at anytime. As an example, wiper on or off, speed of the wiper, rain sensor and other sensors are inputs that can occur at anytime while the system is functioning. This is in contrast to other systems like banking software, where a series of different inputs in a well defined sequence is needed for proper functioning of the software.

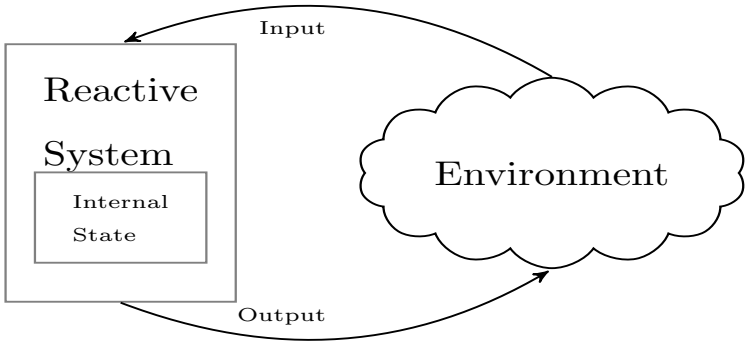

Figure 3.1: Reactive System

Another feature of most embedded systems is that the inputs are predominantly boolean or enumeration types. For example, whether the wiper is on or off, wiper speeds and so on are all types of such inputs. The above information regarding embedded systems can be used for test generation of these systems.

A sample outline of a sequential program which depicts a reactive system is shown in Listing 3.1. In the program, the driver function contains a loop which executes forever. This loop is called the reactive loop. Each iteration of this loop is called as an iteration of the reactive loop or program iteration. This loop reads inputs, executes the system and produces outputs, all in a continuous manner.

3.2 Test vector, test case and test suite

Almost all software systems need to be tested for error detection. We define terms used during this testing and explain it with the help of the wiper control example.

A *test vector* for a system is a single assignment of values to inputs of that

```
driverFunction( ) {
   repeat the loop forever {
   read inputs from environment for the reactive system
   execute the reactive system
   report outputs of reactive system to the environment
   }
}
```

Listing 3.1: Sequential program depicting a reactive system

system. A single test vector is input at each iteration of reactive loop of the program. A *test case* (or *test data*) for a system is a sequence of one or more test vectors. The sequence of test vectors in a test case should be constant. The output of a test case may change if the order of test vector execution is altered. Ideally, a test case should contain expected values of the system for each test vector. Our test cases need not have expected values since we ignore all expected values for the purpose of test generation. A *test suite* is a set of test cases. The order of test cases in the test suite can be altered without changing the output of the system.

For the wiper control system of the car, assume that we would like to test if the wiper works for medium wiper speed for 10 seconds followed by high wiper speed for 5 seconds. Thus, a *test vector* would be the value of medium speed of the wiper (say *tv1*). Another *test vector* would be the value of high speed of the wiper (say *tv2*). These test vectors would be input at different iterations of the system. A *test case* would contain the test vector *tv1* to be executed for 10 seconds followed by test vector *tv2* for 5 seconds. Note that changing the ordering of test vectors may change the output of the system for the test case. Many such individual test cases form the *test suite* for the wiper software.

3.3 Inputs and outputs of an embedded software system

Inputs to an embedded software system are values which are provided by the environment to the reactive program. At a program level, we define input variables as variables which are read in the program. Similarly, outputs are values generated by the reactive system for the environment. We define output variables of a program as global variables to which values are assigned in the program and the return value of program iteration. We assume the set of input and output variables to be disjoint.

3.4 Modified Condition Decision Coverage

Modified Condition Decision Coverage (MC/DC) is one of the most widely used code coverage criterion for embedded systems. MC/DC shows an independent effect of each of its conditions on the decision. To show an independent effect of a condition (called condition under consideration or CUC) on a decision, two sets of test values are needed. In both sets, the values of all conditions, except CUC, are masked so that they do not play any role in the decision making process. Thus, toggling the value of the condition under consideration, the decision should toggle and this becomes a test case which satisfies MC/DC for that condition.

Consider that a particular decision in the code is *cond1 AND (cond2 OR cond3)*. Such code types are common in embedded systems. A usual error in such systems is the logical operator error (AND replaced by OR and vice versa).

For the above case, let us consider *cond2* to be CUC. The two test cases

for MC/DC of *cond2* are *(cond1=T, cond2=T and cond3=F)* and *(cond1=T, cond2=F and cond3=F)*. The two test cases toggle the value of just CUC and the decision toggles. Now assume that first AND operator in the decision should have been an OR operator as per the specification. In this case, the output of the decision in one of the test cases would be computed differently than in the specification. This would help uncover the logical operator error.

There are three types of MC/DC: Unique-Cause MC/DC, Unique-Cause + Masking MC/DC, and Masking MC/DC [11]. We explain in brief the three types of MC/DC and explain our choice of Masking MC/DC in the experiments.

1. **Unique-Cause MC/DC** requires a unique cause for all possible (uncoupled) conditions. Unique cause means that toggling a single condition should change the expression result, with all other conditions constant. In the case of strongly coupled conditions, no coverage set is possible. For example, consider the following expression.

   ```
   if (a or b) and (a or c) then ..
   ```
   (3.4.1)

   For the above expression, conditions a are strongly coupled in the expression. Unique-cause MC/DC cannot be achieved for this expression, since changing the value of one condition changes the other condition too. For such cases, no guidance is provided by DO-178B standards on how to cover these conditions. Fortunately, expressions with strongly coupled conditions are quite rare in airborne software (one study puts it at 72 conditions of 20,256 expressions).

2. **Unique-Cause + Masking MC/DC** requires a unique cause for all possible (uncoupled) conditions. For strongly coupled conditions,

masking will be allowed for that condition only (i.e., all other (uncoupled) conditions will remain fixed). For expression 3.4.1, unique-cause + masking MC/DC is allowed to have values for tuple *(a,b,c)* as *(0,0,1)* and *(1,0,1)* to show independent effect of first instance of the condition *a*.

3. **Masking MC/DC**, as its name implies, allows masking in all cases. This is an extension beyond 2 that masking be allowed for strongly coupled conditions only. For expression 3.4.1, masking MC/DC can have values for tuple *(a,b,c)* as *(0,0,1)* and *(1,0,0)* to show independent effect of first instance of the condition *a*.

Out of the above three types of MC/DC, masking MC/DC is the preferred choice of MC/DC to satisfy maximum MC/DC of a system. There are multiple reasons for choosing masking MC/DC [11]:

1. Masking MC/DC requires a number of tests equivalent or lesser than that of the other forms of MC/DC,

2. The performance of masking MC/DC is nearly identical from the probability of error detection viewpoint,

3. More independence pairs at all levels can be prepared for masking MC/DC than for either of the unique-cause forms. It is assumed that the larger the number of independence pairs, the easier the coverage would be to attain.

MC/DC is the only practically useful code coverage criteria, which guarantees that detection of logical errors. Other code coverage criteria either give no guarantees (like decision coverage) or require too many test cases

(like multiple condition coverage). As a result, we use masking MC/DC in the evaluation of test suite effectiveness.

3.5 Finite State Machine (FSM)

A finite state machine is a mathematical model of computation consisting of a *set of states*, a *start state*, an *input alphabet* and a *transition function* to go from one state to another depending on the input symbol. It can be conceived as an abstract machine that can be in one of a finite number of states. At any given time, the machine can be in only one of its states (the *current* state).

The behaviour of state machines can be observed in a number of systems performing a predetermined sequence of actions depending on a sequence of events with which they are presented. Some common examples include vending machines which dispense products when the proper combination of coins are deposited, elevators, spell-checkers, traffic lights switching between red, yellow and green, and combination locks which require the input of combination numbers in the proper order.

Finite-state machines have been used to model a large number of problems, among which are electronic design automation, communication protocol design, language parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines have been used to describe neurological systems and in linguistics to describe the grammars of natural languages.

The example in Figure 3.2 encodes as an FSM, the design of an overly simplified elevator controller (figure and explanation refered from 'FSM tutorial' [82]). The elevator can be at one of the two floors (states): Ground or First.

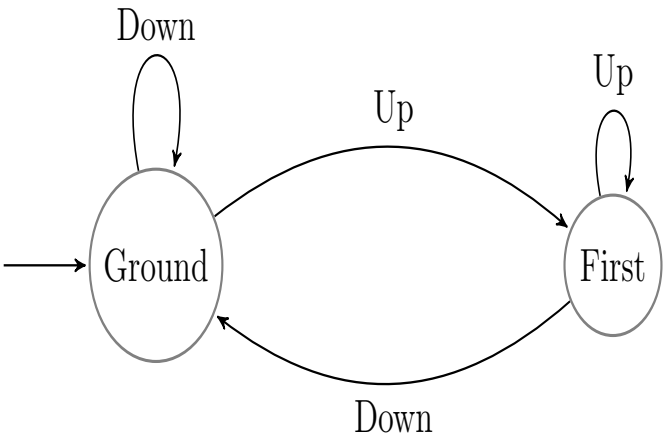

Figure 3.2: An Example FSM

This is controlled through a switch which can take two values (inputs): Up and Down. The circles represent the states and the arrows represent transitions between them. The arrow labels indicate the input value corresponding to the transition. For instance, when the elevator is in the Ground state and the input value is Up, the state of the elevator changes to First.

We will now start looking at the test generation technique, in the chapter to come. To start with, we will spend some time to get a high level overview of the procedure. As a next step, we would zoom in to illustrate all the important steps of the process. The implementation details and the pseudocode of each of them would also form a part of this illustration. We would be ending the chapter with a demonstration of our technique on a sample program.

4. Test Generation Technique

This chapter describes our test generation technique. We split this chapter into four sections. Initially, we give a high level explanation of our technique. Next, we provide a detailed explanation of the test generation process. This includes inputs to our technique and the algorithms underlying test suite generation and test suite reduction. The third section contains the implementation level details of the technique. The final section demonstrates our test generation technique on an example program. We explain each part of the technique, with their inputs and outputs, when the example program is input to test generation.

4.1 Test Generation Technique - High Level

A high level test generation technique is as explained in Figure 4.1.

Our aim is to generate a test suite from the program. This is done by exploiting the dynamic behaviour of the program. The test generation comprises of the following steps.

1. Inputting a reactive program and an initial test suite.

2. Running the test suite on the reactive program

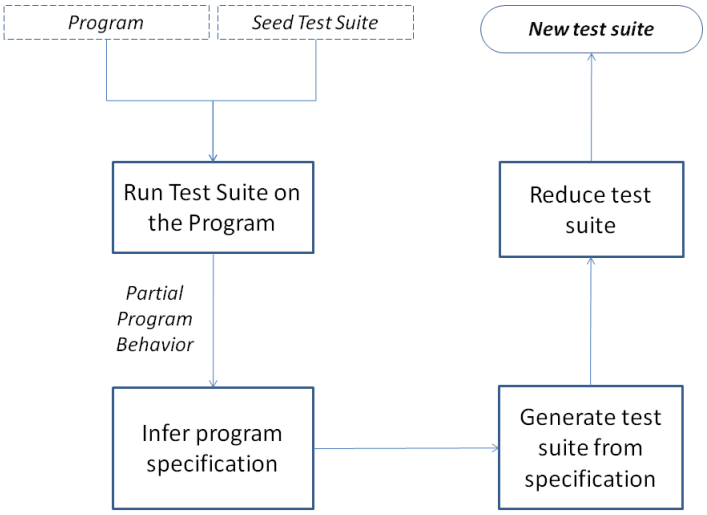

Figure 4.1: Test Generation Technique

The initial test suite, obtained as an input, is executed over the input program to produce execution sequences.

3. Inferring program specifications

   Execution sequences are converted to a specification, depicting the behaviour of the program for the initial test suite.

4. Generating test suite from specification

   Properties of specification are used to generate additional tests, which would explore new behaviours of the program.

5. Reducing the generated test suite

   Test cases not contributing to test suite effectiveness are removed.

6. Outputting the test suite thus generated

   It is the initial test suite with the additional tests appended to it.

The next section details this process of test generation.

4.2 Test Generation Technique - Low level

Before explaining the test suite generation process, we explain the inputs to the test generation process.

4.2.1 Inputs to test generation

The inputs to test generation process are: a reactive program, the input and output variables of the program, an initial test suite and a timeout value. The characteristics of these are explained below.

Reactive program

Our technique of test generation through specification mining is directed toward reactive embedded systems. Programs showing the characteristics of reactive embedded systems (see Section 3.1) are selected. Thus, we assume our programs accept a set of inputs, process them and produce a set of outputs before processing the next set of inputs.

Input and output variables

Inputs and outputs of embedded programs are explained in Section 3.3. For our technique of test generation, we assume that input and output variables of the program have been provided. In Section 5.4, we show how these variables can be extracted automatically from the program.

Initial test suite

As explained in Section 3.2, a test suite consists of test cases. We call a test suite an initial test suite if the test suite is already available to

us. We require this initial test suite for our test generation.

We assume the availability of such an initial test suite (TS), to guide the specification mining process. In case such a test suite is not available, Section 5.4 explains possible steps to prepare an initial test suite.

Timeout value

For some programs, our technique of test generation may take time that is not acceptable in practice. This is true for reactive programs, which are designed to run for long durations and thus will have long tests. Preparing such tests may take time. For these programs, we need some external control to stop the test suite generation process if it exceeds a pre-specified time limit. If such an external control is not available, the test suite generation process may take a very long time before stopping. Hence, we accept a timeout value for our test generation technique. When the time taken by the test generation algorithm exceeds the timeout value, the test suite generation process is halted. The value of timeout can be configured to be within the time constraints. This enables us to experiment with large systems with predecided time bounds on the test generation process.

4.2.2 Test Generation Method

For each stage of test generation and reduction, we classify the stage into three components. The first component informally explains the process of that stage. The next component gives input and output formats of each stage. The final component gives implementation level details of that stage with algorithms and data structures used in implementation.

We have implemented our test generation technique in the *PERL* pro-

gramming language. The main function of the *PERL* script has steps as in Algorithm 1. Throughout this thesis, we refer to this algorithm as the *main algorithm*.

Inputs to the algorithm are: a C file (F), name of the selected function under test ($funcName$), the list of input variables (I_{list}), the list of output variables (O_{list}), the initial test suite (TS) and the timeout value (TM_{val}). For simplicity of explanation, the algorithm shown here accepts only a single C file. This, however, is not a limitation of the procedure. The algorithm can trivially be modified to accept multiple files as input. The algorithm first prepares a wrapper function to test the selected function. Next, an executable file is prepared to execute the test suite over the selected function. A loop executes the set of statements of recording program behaviours (recProgBehaviour), preparing FSMs from program traces (prepareFSMsfromTraces), merging the FSMs to generate a single specification FSM (mergeFSMs) and generating test data from the specification FSM (genTestDataFromSpec). The loop terminates when no new test data can be generated from the specification with our technique (errorFlag) or when timeout has occurred ($lTimeVar >= TM_{val}$). Once the loop has executed, we have TS with initial and new test cases. The test suite (TS) is optimized using a reduction criteria (reduceTestSuite) to generate a reduced test suite $TS_{reduced}$. The algorithm outputs the test suite $TS_{reduced}$.

In practice, prepareFSMsfromTraces and mergeFSMs are part of a single stage. For simplicity, they are explained as separate stages.

Each of these are explained below in detail.

1. **Prepare program environment**

   Preparing a program environment includes preparing a driver function and using the driver function to prepare a program executable.

Algorithm 1 PseudoPERL script for test generation

Input: $F, I_{list}, O_{list}, TS, TM_{val}$

$driverFunc$ = prepareProgramDriver ($F, funcName, I_{list}, O_{list}$)

$execFile$ = prepareExecutable ($driverFunc, F$)

$testGenFlag, lTimeVar, FSMspec$ = (TRUE, START, ())

$TS_{new} = TS$

while $testGenFlag$ = TRUE **do**

$traceFiles$ = recProgBehaviour ($execFile, TS_{new}$)

$FSMs$ = prepareFSMsfromTraces ($traceFiles$)

$FSMspec$ = mergeFSMs ($FSMs, FSMspec$)

$newTestData, errorFlag$ = genTestDataFromSpec ($FSMspec$)

if $errorFlag$ = FALSE **then**

TS = appendNewTestData($TS, newTestData$)

$TS_{new} = newTestData$

if $lTimeVar >= TM_{val}$ **then**

$testGenFlag$ = FALSE

end if

else

$testGenFlag$ = FALSE

end if

end while

$TS_{reduced}$ = reduceTestSuite (TS)

Output: $TS_{reduced}$

Initially, we prepare a driver for the program. This driver simulates an environment to the actual program function. The driver is similar to the program shown in Listing 3.1. Driver contains a reactive loop where input values are accepted, the program function is called and the output is displayed. The driver can be generated with the availability of input and output variables.

Once the driver function is generated, we prepare an executable file with the program and driver function.

Implementation detail

To generate the driver for the program, we need

- input and output variables with their datatypes, and
- signature of the program function.

With this information, the driver can be prepared as shown in Figure 4.1. Function *prepareProgramDriver* from *main algorithm* produces this driver function. The driver function is written to a separate file. The driver in the figure is for programs in pseudo C language. A similar driver can be prepared for most such languages.

In the initial part of the driver, file pointers and temporary variables are declared. This declaration is with respect to the datatypes. Next, a *driverFunction* is written which simulates the actual reactive environment. The *driverFunction* opens the input test file for reading and executes the reactive loop till all tests in a test file are read. It also calls functions which read inputs and print outputs in a desirable format.

Once the driver is generated, an executable file is prepared. We use the *GCC* compiler to generate the executable file. The function *prepare-*

Executable calls the *GCC* compiler with the C files and driver function as input and outputs an executable file.

2. **Run initial test suite on the program**

   The initial test suite TS is executed on the program to produce a set of program runs.

   The driver in step 1 is prepared such that each test case is input to the program. The function is executed for that test case. Result, namely the values of output variables, are printed to a file. This process of reading test vector, executing the function and capturing outputs is sequentially performed for each test vector in the test case. This process is repeated for each test case in the test suite.

   Each test case produces a program trace or a program run, which contains information regarding the program's run for a set of inputs, executed in sequence. We prepare this trace as shown in Figure 4.2

   The program run has information of the values of input variables and program states in the sequence of execution. First, the initial values of output variables are recorded. This is followed by the values of input variables. Next, the values of output variables after execution of inputs on the function are noted. This pair of related input-output values is recorded for all test vectors in a test case. Execution of all test cases in the test suite result in a set of such program runs.

   Implementation detail

   The executable generated in the previous stage is used to prepare program runs. The function *recProgBehaviour* produces these program runs by running the test suite (TS) over the executable. The PseudoPERL algorithm for *recProgBehaviour* is shown in Algorithm 2.

```
// This driver is prepared for function <programFunction>.

FILE *fpInpFile;
char *inpFile;

// Temporary variables prepared as per their datatype.

<dt1> retVar;         // return value of program function
<dt2> glbForParam1;   // parameters of program function
<dt3> glbForParam2;

void driverFunction( ) {
   unsigned int iterationVar = 0;
   fpInpFile = openFile(``inpFile");

   printf(``INITIAL_OUTPUTS\n");
   reportOutputsFromFunction ( );

   while( fpInpFile != NULL ) {
      getInputsForFunction ( );
      retVar = <programFunction>(glbForParam1, glbForParam2,
          ...);
      reportOutputsFromFunction ( );
   }
}

void getInputsForFunction ( ) {
   fscanf(``%d %f %d %f ... \n",&glbInp1, &glbInp2, &
       glbForParam1, &glbForParam2, ...);
   printf(``INPUTS\n");
   printf(``%d_%f_%d_%f ... \n",glbInp1, glbInp2, glbForParam1,
       glbForParam2, ...);
}

void reportOutputsFromFunction ( ) {
   printf(``OUTPUTS\n");
   printf(``%d_%f ... \n",glbOut1, glbOut2, ...);
}
```

Listing 4.1: Sample Program Driver

```
INITIAL_OUTPUTS
<O1>_<O2>_<O3>_ ...
INPUTS
<I1>_<I2>_<I3>_ ...
OUTPUTS
<O4>_<O5>_<O6>_ ...
INPUTS
<I4>_<I5>_<I6>_ ...
OUTPUTS
<O7>_<O8>_<O9>_ ...
...
```

Figure 4.2: Program Trace Format

Algorithm 2 PseudoPERL function for *recProgBehaviour*

Input: $execFile, TS$

$traceFiles = ()$

for $testCase$ in TS **do**

$opFile =$ prepareTraceFile ( )

$opFile =$ execute ($execFile, testCase$)

$traceFiles =$ addToSetOfFiles ($traceFiles, opFile$)

end for

Output: $traceFiles$

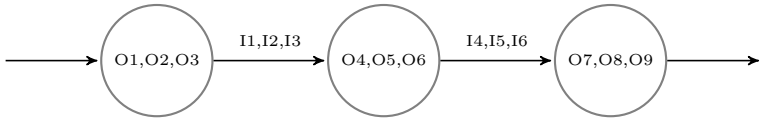

Figure 4.3: FSM of sample program trace

The function *recProgBehaviour* takes the executable file (*execFile*) and the test suite (TS) as input to produce a set of trace files $traceFiles$. Each test case in the test suite is executed over the executable file. The driver function present in the executable file prepares an output trace file recording behavioral information of the test case. This information is captured in the $opFile$ file. These files are collected in a set which we call $traceFiles$.

3. **For each program run, prepare a specification in the form of a FSM**

   An individual program run is converted into a FSM. Each output becomes a state in the FSM and the input becomes the guard on some transition. Each run produces a FSM. To prepare this, we sequentially traverse the program run. The values of initial output form the initial state of the FSM. The value of next occurring inputs form the guard on the transition to the state formed from the following values of output variables. This process continues until all the states have been exhausted.

   For the program run in Figure 4.2, Figure 4.3 gives the FSM so produced.

   Implementation detail

   The function *prepareFSMsfromTraces* in the *main algorithm* prepares

FSMs from program runs. The PseudoPERL algorithm for *prepareFSMsfromTraces* is shown in Algorithm 3.

We iterate over the set of program traces to get each program trace. A program trace thus selected is converted into a FSM. For preparing a FSM from a program trace, we read pairs of lines from the program trace and classify every pair as either an input or an output. The first line of the pair denotes whether the pair is an input (INPUTS) or an output (INITIAL_OUTPUTS or OUTPUTS). The second line of the pair contains the respective values. In case of input, the values are added to the input array (addToInputArray). In case of output, they are added to the state array (addToStateArray). After every output pair denoted by (OUTPUTS), the sequence of inputs and states are stored in transition array (addToTransitionArray). Thus, after reading the complete trace file, we obtain three array of arrays: inputArray, stateArray and transitionArray. These arrays store the entire information of the trace file.

Arrays inputArray and stateArray are of the same format. For example in Figure 4.3, stateArray will be ((O1,O2,O3...), (O4,O5,O6...), (O7,O8,O9...)) and inputArray will be ((I1,I2,I3...), (I4,I5,I6...)). Each element of stateArray will contain an array of output values. Similarly, each element of inputArray will contain an array of input values. For both arrays, the values are added to the respective arrays after performing a duplicate check. It may be possible that values to be added as element are previously present. In that case, the new element is not added, but the index of previously present same array element is returned.

Array transitionArray contains the linking of the arrays in the form

Algorithm 3 PseudoPERL function for *prepareFSMsfromTraces*

Input: $traceFiles$

$FSMs = ()$

for each $runFile$ in $traceFiles$ **do**

inputArray, stateArray, transitionArray = (), (), ()

while ($line = readNextLine(runFile)$) not end of file **do**

if $line = INITIAL\_OUTPUTS$ **then**

opLine = readNextLine(runFile)

prevStateIndex = addToStateArray (stateArray, opLine)

else if $line = INPUTS$ **then**

opLine = readNextLine(runFile)

prevInputIndex = addToInputArray (inputArray, opLine)

else if $line = OUTPUTS$ **then**

opLine = readNextLine(runFile)

stateIndex = addToStateArray (stateArray, opLine)

transitionArray = addToTransitionArray ((prevStateIndex, prevInputIndex, stateIndex))

prevStateIndex = stateIndex

end if

end while

FSMs = addToFSMs ($FSMS, (inputArray, stateArray, transitionArray)$)

end for

Output: $FSMs$

of $((s_0, i_0, t_0)$, (s_1, i_1, t_1) ...), where s_x and t_x represents index of state array elements while i_x represents index of input array. For example in Figure 4.3, transitionArray will contain ((0,0,1), (1,1,2), ...).

Thus, these three arrays form a FSM represented in a specific format. Using a combination of these three arrays, one can easily generate a representation as in Figure 4.3. For each trace file, the three arrays are produced and collected in *FSMs*. The *FSMs* are output from this stage of the algorithm.

4. **Merge individual FSMs to form a single FSM**

   Each FSM depicts a behaviour of the program for a specific input. The program may show either total identical behaviour or partial identical behaviour on two or more inputs. This identical behaviour can be detected and the corresponding FSMs merged, to reduce redundancy of behaviours in the FSMs. Thus, all FSMs are merged into a single FSM to make it easier to work with a single FSM representing complete behaviour of program for a test suite than many individual FSMs.

   The individual FSMs are combined into a single FSM by merging states with same values. If two FSMs have a state with same values, then the two states form a single state. All incoming transitions to the two states are now sink into the merged state. Also, all outgoing transitions from the two states have a single source state which is the merged state.

   Figure 4.4 depicts this state merging process. For simplicity of explanation, we represent the state values and input values with q and i respectively. Consider *FSM 1* as a FSM produced from a program run and *FSM 2* as another FSM from an other program run and state qB and qY to be the same states. The merged FSM is shown in the

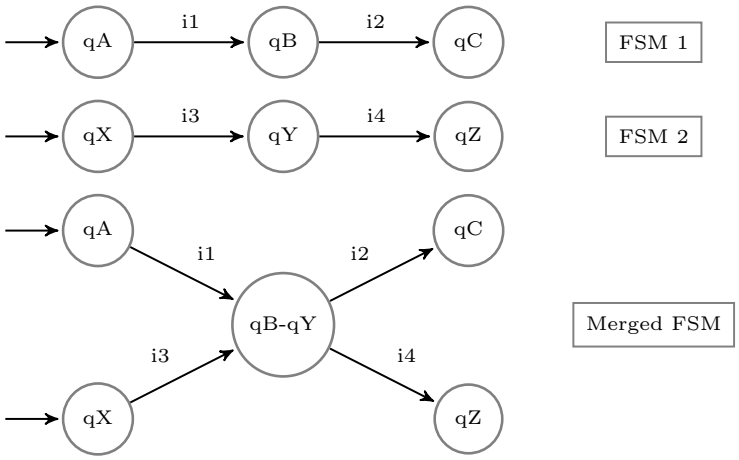

Figure 4.4: State merging process

diagram where the states are merged and the transitions are adjusted. All other non-identical states retain their existence in the merged FSM along with their transitions.

Implementation detail

The function *mergeFSMs* in the *main algorithm* merges individual FSMs into a single FSM specification. The PseudoPERL algorithm for *mergeFSMs* is shown in Algorithm 4.

The $FSMspec$ contains the specification in the form of a FSM. Format of $FSMspec$ is same as the format of $FSMs$, i.e. collection of three arrays of inputs, states and transitions. To prepare $FSMspec$, we merge the $FSMs$. Thus, we want to merge all respective arrays of inputs, states and transitions into three arrays of inputs, states and transitions which will represent the complete specification.

We pick each $singleFSM$ from set of $FSMs$ and merge it with $FSMspec$. Initially, $FSMspec$ will be empty and thus will be directly assigned the

Algorithm 4 PseudoPERL function for *mergeFSMs*

Input: $FSMs, FSMspec$

for each $singleFSM$ in $FSMs$ **do**

$(inputArray, stateArray, transitionArray) = singleFSM$

if $FSMspec = ()$ **then**

$FSMspec = singleFSM$

else

$setOfSimilarStates$ = compareFSMsForState($FSMspec, stateArray$)

$FSMspec$ = addAllUniqueStates ($FSMspec, setOfSimilarStates$)

$setOfSimilarInputs$ = compareFSMsForInput($FSMspec, inputArray$)

$FSMspec$ = addAllUniqueInputs ($FSMspec, setOfSimilarInputs$)

for each $transition$ in $transitionArray$ **do**

($source, input, target$) = $transition$

$source$ = getStateIndexFromFSMspec ($FSMspec, stateArray[source]$)

$input$ = getInputIndexFromFSMspec ($FSMspec, inputArray[input]$)

$source$ = getStateIndexFromFSMspec ($FSMspec, stateArray[target]$)

$FSMspec$ = addTransToSpec (($source, input, target$))

end for

end if

end for

Output: $FSMspec$

values of $singleFSM$. For all further cases, we first determine all pairs of similar states (compareFSMsForState) and inputs (compareFSMsForInput) in $singleFSM$ and $FSMspec$. All unique states and inputs are added to the $FSMspec$. For each transition in transitionArray of $singleFSM$, the tuple (source, input, target) is updated with new index values from similar arrays of $FSMspec$. This updated transition is added to $FSMspec$ using function (addTransToSpec). If a particular transition is found to be duplicate, it will not be added by (addTransToSpec).

Iteratively performing this action for all FSMs, we will get a merged FSM in $FSMspec$. This specification will be used for test generation process in the next step.

5. **Generate additional test cases using the merged specification** Additional test cases are generated such that each newly generated test case explores unexecuted runs of the program. This is achieved by taking runs that end in a state that has no outgoing transition and extending it by an input test vector that was generated earlier. This will lead to a new transition to either a new state or an existing state.

   The motivation to choose a terminal state is with the assurity that any input executed from a terminal state will generate a new state or a new transition or both. The specification in the form of a FSM represents the behaviour of the program for a test suite. If we are able to prepare test cases such that new states and transitions in the specification are generated, then we may have new test cases which probe previously unexplored behaviours of the program. If we execute a new test vector from a non terminal state, we might generate another transition to the

same state of the FSM. Thus, we might have two inputs which have same source and target states. This would mostly mean taking the same program path with different inputs, which is unlikely to increase code coverage. Intuitively, it is easier to achieve more program coverage by generating new states in the program. Thus, we target test generation from terminal states than other intermediate states.

For example, referring to the merged FSM in Figure 4.4, states qC and qZ are two states from which there are no outgoing transitions. One such terminal state is randomly selected, say state qC. Also, an input vector from the test suite is randomly selected, say ix. Next, existing test case to reach state qC from initial state is selected and the newly selected test vector of ix is appended to it. So the new test case formed is $\langle ...,i1,i2,ix \rangle$.

Generating a test case by selecting qB-qY as the selected state, may result in an input such that new transition is from state qB-qY to state qC or qZ. Thus we select a terminal state for test generation.

Implementation detail

The function *genTestDataFromSpec* generates new test data using *FSM-spec*. The PseudoPERL algorithm for *genTestDataFromSpec* is shown in Algorithm 5.

The steps in the algorithm are as per the technique explained above. First a random terminal state (*state*) is selected (getRandomTerminal-State) from the specification. We used *PERL* provided *rand* function to generate random values. Next, a list of input vectors (*inputList*) to reach *state* from the initial state is deduced (getListOfInputToReachState). A random test vector from the available set of test vec-

Algorithm 5 PseudoPERL function for *genTestDataFromSpec*
Input: $FSMspec$

$state$ = getRandomTerminalState ($FSMspec$)

$inputList$ = getListOfInputToReachState ($FSMspec, state$)

$inputSelected$ = getRandomInputVector ($FSMspec$)

$newInput$ = prepareInput ($inputList, inputSelected$)

Output: $newInput$

tors is selected (getRandomInputVector) and is appended to $inputList$ (prepareInput) to produce a new input test case as $newInput$. This $newInput$ is the test case generated by our technique.

The new test case is executed on the actual code of the program. This program execution produces a new program run. The process of generating a FSM from program run is repeated. This FSM is merged with the existing specification. The new program run produces a new program transition and a new program state in the specification. This step of test generation is repeated until there are no terminal states or the process times out. At the end of this step, a new test suite is generated which is the collection of all test cases generated in this step.

6. **Test suite reduction**

   The test suite generated from step 5 is reduced as follows:

   (a) Sort the test suite in descending order with respect to number of test vectors in a test case. This helps us ensure that a test case completely contained within another is removed from the final test suite.

   Thus, a sorted test suite will have test cases with more number of

test vectors at the beginning of the test suite and the number of test cases with lesser number of test vectors at the end of the test suite.

(b) Using the sorted test suite regenerate the merged specification as a FSM as in the test generation technique. Each test case is individually executed and FSMs are prepared from their program runs. Then, the individual FSMs are merged as per the merging process explained in the test generation technique. The order of merging the individual FSMs is determined by the order of test cases in the sorted test suite. Thus, the FSMs prepared by the first two test cases are merged first. This new FSM is next merged with the third test case to produce a new merged FSM. This process continues until all test cases in the test suite are exhausted.

While merging the test cases, we want to discard all test cases which do not generate a new state or which generate only duplicate transitions between source and target states with different inputs. A test case is not discarded when,

- the test case generates a new state in the FSM, or
- the test case generates a transition to an existing state in the FSM where there is no direct transition between the source and target state.

Considering Figure 4.5, assume that we have an initial merged FSM formed from a test suite and three new test cases *TC1*,*TC2* and *TC3*. Successive merging of FSMs produced by *TC1*, *TC2* and *TC3* with the initial merged FSM produces a new merged FSM. Looking at the initial merged FSM and the new merged FSM, one can conclude the following:

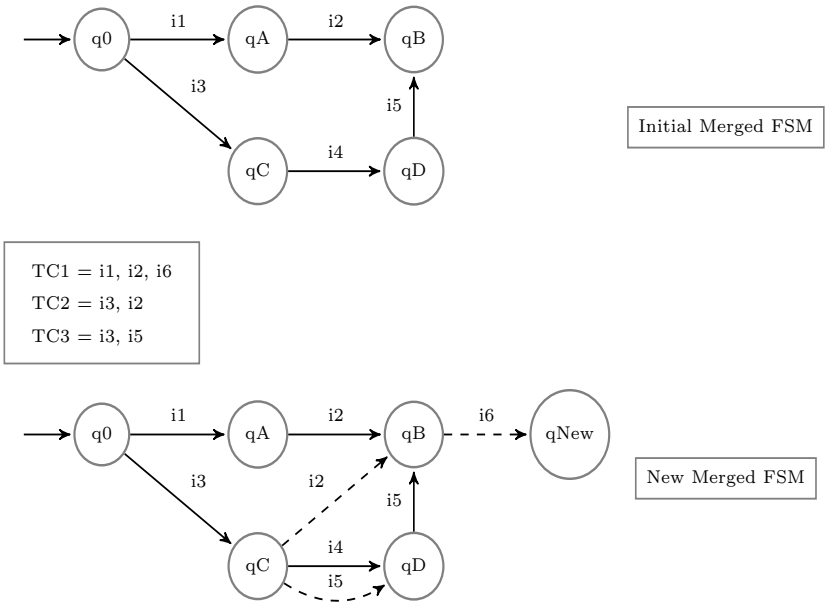

Figure 4.5: Test suite reduction

- *TC1* generates a new state (*qNew*). Since a new state is generated, this test case is not discarded.
- *TC2* does not generate a new state, but generates a transition between states *qC* and *qB*. Since there is no previous direct transition between state *qC* and *qB*, this test case is not discarded.
- *TC3* neither generates a new state, nor a new transition. It does generate a transition between states *qC* and *qD*, but there already exists a transition between these two states. As a result, this test case is discarded, since it is assumed that this test case does not add any value to the test suite.

(c) Output is a reduced test suite with all unrequired test cases removed from final test suite. All unrequired test cases, if added to the test suite, would not produce any new state in the FSM and

would also not produce any unique transition between a source and target state.

Implementation detail

The function *reduceTestSuite* in the *main algorithm* removes redundant test cases from the generated test suite based on a criterion. The PseudoPERL algorithm for *reduceTestSuite* is shown in Algorithm 6.

The reduction algorithm is a modified version of Algorithm 2. The test suite file TS and an executable file $execFile$ are input to the algorithm. Initially, we sort the test suite file TS based on the number of test vectors (sortTestSuite). Priority is given to test cases with larger number of test vectors. For each test case in the test suite, following actions are performed in sequence. First, a test case is executed over the program to produce a program run (recProgBehaviour). The program run file has same format as that of Figure 4.2. Program run file is read and classified as either input or output pair as in Algorithm 2. While classifying and preparing arrays from the file, additional information regarding states and transitions is prepared. Every new state and every transition between two unconnected states is reported by functions (addToStateArray) and (addToTransitionArray) respectively. Once such a state or transition is found, the test case is not discarded. All other test cases are discarded. Thus, we get a reduced test suite in the form of $TS_{reduced}$.

7. The reduced test suite is now used in the evaluation process.

Algorithm 6 PseudoPERL function for *reduceTestSuite*

Input: $TS, execFile$

```
TS_reduced = ()
inputArray, stateArray, transitionArray = (), (), ()
TS = sortTestSuite ( TS )
for each testCase in TS do
  traceFile = recProgBehaviour ( execFile, testCase )
  newStateFlag = FALSE;
  while (line = readNextLine(traceFile)) not end of file do
    opLine = readNextLine(traceFile)
    if line = INITIAL_OUTPUTS then
      (prevStateIndex, newStateFlag) = addToStateArray (stateArray,
      opLine)
      if newStateFlag = TRUE then
        last # "last" is PERL is same as "break" in C
      end if
    else if line = INPUTS then
      prevInputIndex = addToInputArray (inputArray, opLine)
    else if line = OUTPUTS then
      (prevStateIndex, newStateFlag) = addToStateArray (stateArray,
      opLine)
      (transitionArray, newTransFlag) = addToTransitionArray ((prevStateIn-
      dex, prevInputIndex, stateIndex))
      if newStateFlag = TRUE or newTransFlag = TRUE then
        last
      end if
      prevStateIndex = stateIndex
    end if
  end while
  if newStateFlag = TRUE or newTransFlag = TRUE then
    TS_reduced = appendToTestSuite ( TS_reduced, testCase )
  end if
end for
```

Output: $TS_{reduced}$

4.3 Demonstration on an example program

In this section, we run our technique of test generation on an example C program and explain the inputs and outputs at each stage of the technique.

4.3.1 Example program

We use a simple wiper code program as an example to illustrate our technique. Relevant explanations have been made in this section, with respect to the example. The code of wiper program is in Listing 4.2.

The wiper code consists of a single function, which determines the functioning of the wiper. The function accepts five inputs from the environment, out of which two are boolean inputs, two are enumerations while one is an unsigned integer. The function generates four output variables, where two are boolean, one is enumeration while one is an unsigned integer. The *main* function acts like the environment of the wiper function. As seen from the code, the *main* function repetitively and sequentially performs the following three steps:

- Get input values
- Call wiperControl function
- Report output values

The *while* loop in *main* function is the reactive loop of the program. Thus, the *main* function simulates a reactive environment for the *wiperControl* code.

```
// Inputs from the environment as below
bool wiperOperate;          // Operate ON or OFF
bool engineStatus;          // Engine ON or OFF
enum rainSensorEnum {VERY_LOW,LOW,MEDIUM,HIGH,VERY_HIGH}
    rainSensor;
enum wiperSpeedEnum {VERY_SLOW,SLOW,INTERMEDIATE,FAST,VERY_FAST}
     wiperSpeed;
unsigned int vehicleSpeed; // Value between 0 to 200

// Outputs to the environment as below
bool wiperOutput;           // Wiper should operate or not
bool wiperDirection;        // Go left or right
enum wiperSpeedEnum outputWiperSpeed;
unsigned int timerVar;      // Timer variable from 0 to MAX_INT

// Actual Function
void wiperControl( ) {
   // Function which computes the outputs as per the
   // inputs and previous output values
}

// Environment to wiperControl function
void driverFunction ( ) {
   while(1) {
      getInputsForFunction ( );
      wiperControl( );
      reportOutputsFromFunction ( );
   }
}
```

Listing 4.2: Example Program: Code

Table 4.1: Example program: Initial Test Suite

Variable	wiper Operate	engine Status	rain Sensor	wiper Speed	vehicle Speed
TestCase 1	0	0	0	0	0
	1	1	0	1	0
	1	1	0	1	20
TestCase 2	1	0	0	0	0
	1	1	2	0	0
	1	1	0	1	20
	1	1	0	3	40
TestCase 3	1	1	0	0	0
	1	1	0	4	100

4.3.2 Execution on Example Program

1. Inputs to the technique

   Input program The program explained in Listing 4.2 is an example reactive program. We assume this program to be representative of most embedded programs.

   Input and output variables The list of input and output variables is available. There are five input and four output variables as mentioned in 4.3.1.

   Initial test suite Let us assume that we have the initial test suite as in Table 4.1. This test suite has three test cases, each with a set of test vectors. Each test vector is input to the wiperControl code at each iteration of the reactive loop.

Timeout value For this example, we ignore the timeout value since this example is for explanation purpose.

2. Test Generation Technique on example program

   We explain each step of the technique on execution on the example program.

   (a) **Prepare program environment**

   We prepare a driver for the program.

   The *driverFunction* is this driver function for the example program. The *driverFunction* can be automatically prepared once the input and output variables are known. The *getInputsForFunction* function sets input values to input variables while the *reportOutputsFromFunction* function displays the values of output variables. The *wiperControl* is the actual function call to perform the functionality of wiper control. All this code is put in a while loop, which acts as a reactive loop.

   (b) **Run initial test suite on the program**

   We execute the initial test suite TS (Table 4.1) on the example program to get a set of program runs. On execution of first test case in the test suite of Table 4.1, the execution run prepared is as in Figure 4.6.

   Execution of all test cases of Table 4.1 gives three similar program runs, one each for a test case.

   (c) **For each program run, prepare specification in the form of a FSM**

   From the three program runs, three individual FSMs are obtained. Each FSM depicts the behaviour of the program for a test case.

```
INITIAL_OUTPUTS
0_0_0_0
INPUTS
0_0_0_0_0
OUTPUTS
0_0_0_10
INPUTS
1_1_0_1_0
OUTPUTS
1_0_0_25
INPUTS
1_1_0_1_20
OUTPUTS
1_1_0_35
```

Figure 4.6: Example Program: Trace Format

For the program run in Figure 4.6, FSM labelled *FSM1* is produced as in Figure 4.7.

Assume the three FSMs to be as in Figure 4.7.

(d) **Merge individual FSMs to form a single FSM**

Figure 4.9 depicts the merged FSM from individual FSMs.

To prepare the merged FSM, we pick the first two FSMs *FSM1* and *FSM2* of Figure 4.7. Observing the two FSMs, one can conclude that:

- states *Sa* and *Se* are same,
- states *Sb* and *Sf* are same,
- none of the other states are identical.

Thus, one can merge state *Sa* with state Se. None of these states have any incoming transition. Both have a single outgoing transition. Thus, one can generate a single state *Sae*, which has same value as *Sa* and *Se*. The outgoing transition from *Sa*, now has *Sae*

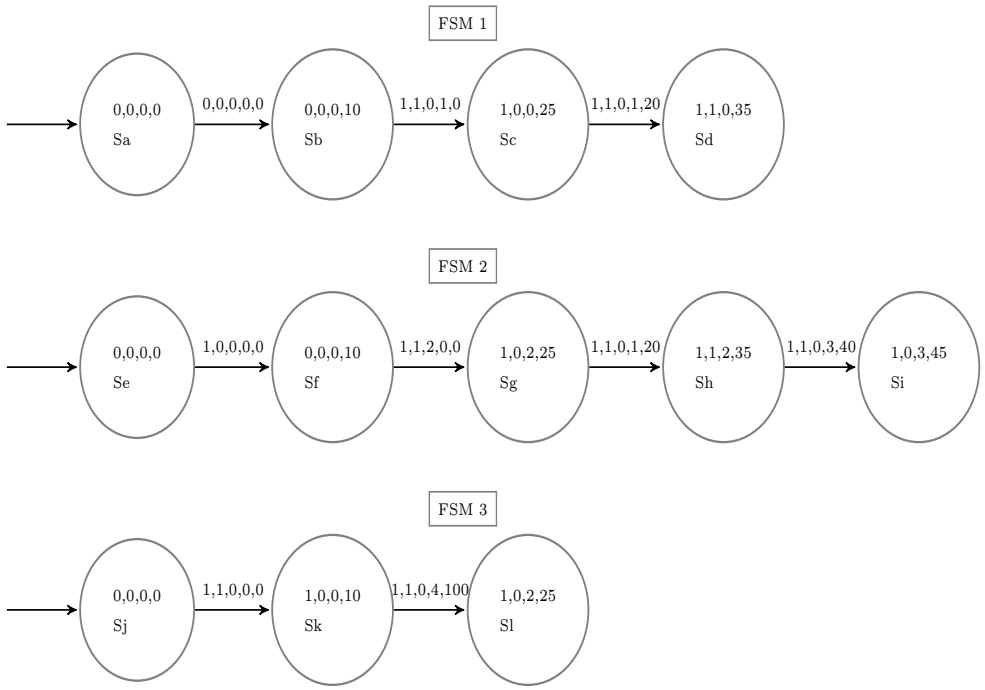

Figure 4.7: Example program: Three FSMs of program run

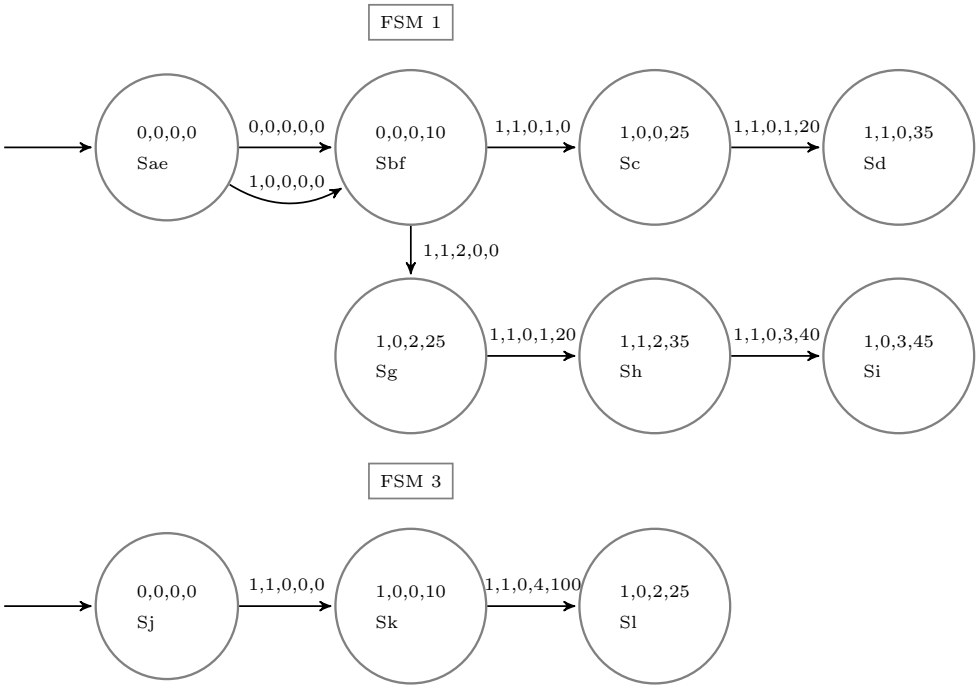

Figure 4.8: Example program: Semi-merged FSM

as the source and *Sb* as the target. Similarly, outgoing transition from *Se*, now has *Sae* as the source and *Sf* as the target. In an identical manner, states *Sb* and *Sf* can be merged to give state *Sbf*. Thus, we have a semi-merged FSM as in Figure 4.8. *FSM3* has been replicated in Figure 4.8 from Figure 4.7 for simplicity.

Next, *FSM3* is to be merged with the semi-merged FSM of Figure 4.8. Two states of *FSM3* have an identical state in Figure 4.8. Thus, we create a merged FSM as in Figure 4.9.

(e) **Generate additional test cases using the merged specification**

We target terminal states of FSM and execute inputs from those states. Referring to the merged FSM in Figure 4.9, the two high-

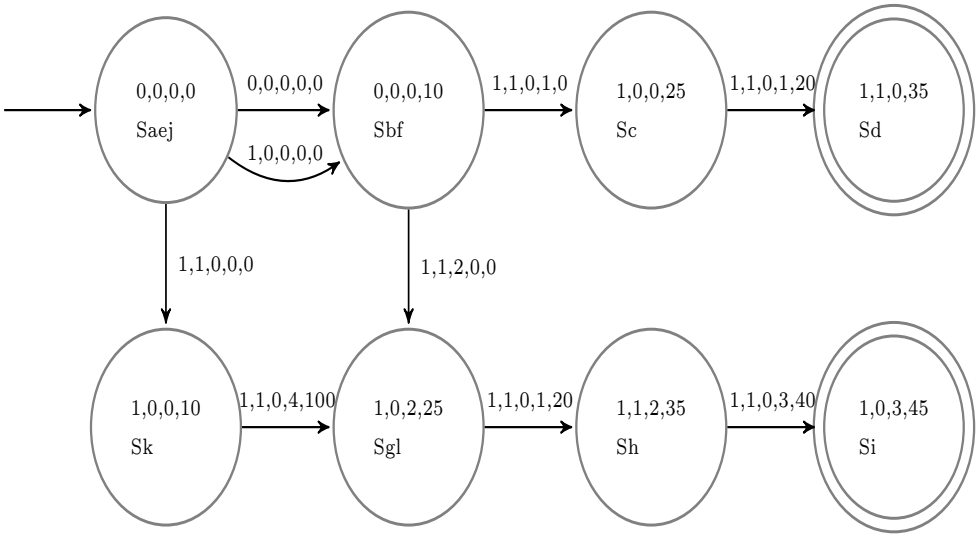

Figure 4.9: Example program: Merged FSM specification

lighted states *Sd* and *Si* are the terminal states from which there are no outgoing transitions. One such terminal state is randomly selected, say state *Sd*. Also, an input vector from the test suite is randomly selected, say *1,1,0,1,0*. Next, existing test case to reach state *Si* from initial state is selected and the randomly selected test vector is appended to it. So the new test case formed is $\langle$*0,0,0,0,0 - 1,1,0,1,0 - 1,1,0,1,20 - 1,1,0,1,0*$\rangle$.

This new test case is executed on the actual code of the program. This program execution produces a new program run. The process of generating a FSM from program run is repeated. This FSM is merged with the existing specification. The new program run produces a new program transition and a new program state in the specification. This step of test generation is repeated until there are no terminal states or the process times out.

At the end of this step, a new test suite is generated, which is the

Table 4.2: Example program: New Test Suite

Variable	wiper Operate	engine Status	rain Sensor	wiper Speed	vehicle Speed
TestCase 1	0	0	0	0	0
	1	1	0	1	0
	1	1	0	1	20
TestCase 2	1	0	0	0	0
	1	1	2	0	0
	1	1	0	1	20
	1	1	0	3	40
TestCase 3	1	1	0	0	0
	1	1	0	4	100
TestCase 4	0	0	0	0	0
	1	1	0	1	0
	1	1	0	1	20
	1	1	0	1	0

collection of all test cases generated in this step.

(f) **Test suite reduction**

Assume the test suite in Table 4.2 has been generated by the test generation technique. This test suite is reduced as follows:

i. Sort the test suite in descending order with respect to number of test vectors in a test case. As seen in the table, test cases 2 and 4 have four test vectors each, test case 1 has three test vectors while test case 3 has two test vectors. Thus, the order of test cases in the test suite is now 2, 4, 1, 3.

ii. Using the sorted test suite regenerate the merged specification

as a FSM in the test generation technique.

Thus, test cases from the test suite are executed in the sorted order. Clearly, test case 1 is completely contained within test case 4. Thus the individual FSM created by test case 1 will be a subset of the FSM of test case 4. Also, test case 4 is executed first, followed by test case 1. As a result, during merging, test case 1 will not contribute to the merged FSM, since the merged FSM will already have test case 4. So test case 1 will not generate any new state or a new transition. So, test case 1 is ignored.

iii. Output is a reduced test suite with all unrequired test cases removed from final test suite. Thus, the final test suite is as shown in Table 4.2 without the first test case.

(g) The final test suite is output from the technique.

In the following chapter, we evaulate our technique based on the experiments performed on some selected case studies. To begin, we describe the process of test suite evaluation in details. We explain the different case studies on which we worked and list the resources used for the experimentation. Then, we move on to illustrate the experimental set-up and the steps involved. We provide the results and explain our observation towards the end of the chapter. We end the chapter drawing inferences from our experiments and listing the limitations of our technique.

5. Experimental Evaluation

In this chapter, we explain the actual experimentation performed on case studies to determine the effectiveness of our test suite with test suites generated from other techniques. We first explain the process of test suite evaluation in detail. Next we describe the example programs chosen for experimentation while arguing the choice of these case studies. This is followed by the experimentation details. All steps present in the test generation process are explained on the example programs. Next section shows the results and observations of the experiments. We list some assumptions of our experimentation process with respect to usability of our approach. The chapter is concluded with assessment of threats to the validity of our technique.

5.1 Evaluation of test suite effectiveness

We evaluate test suite effectiveness based on MC/DC. We assume that more the MC/DC achieved by a test suite, more is the effectiveness of that test suite.

We compare the test suite effectiveness of our technique against an MC/DC-satisfying test suite generated using AutoGen [10]. AutoGen automatically generates an MC/DC-satisfying test suite for reactive programs in C. Auto-

Gen generates test data for conditions that cover MC/DC and reports cases for which it is not possible to generate test data. AutoGen uses a combination of program analysis and model checking techniques to generate test data. In program analysis, program slicing [85] is performed which helps AutoGen to scale up. Model checking techniques generate the required test data. We assume AutoGen generated MC/DC to be the maximum achievable MC/DC for a program. This is possible for programs where AutoGen scales up. For our experiments, AutoGen is the only tool available that generates tests satisfying MC/DC hence we use it. While other tools exist, like Reactis for C [71] and CoverageMaster WINAMS [81], they were not available to us. Hence all results are compared against AutoGen.

The procedure to compare test suite effectiveness of our technique with a AutoGen generated test suite based on MC/DC is summarized as follows, with reference to Figure 5.1.

1. For an embedded program, generate a MC/DC-satisfying test suite (TS_1) using AutoGen.

2. Using a seed test suite (TS), run the specification mining technique to generate FSM and subsequently a new test suite (TS_2) as in Section 4.2.

3. Apply reduction technique to obtain reduced test suite (TS_3), again as in Section 4.2.

4. Compare test suites TS_1, TS_2 and TS_3 based on MC/DC.

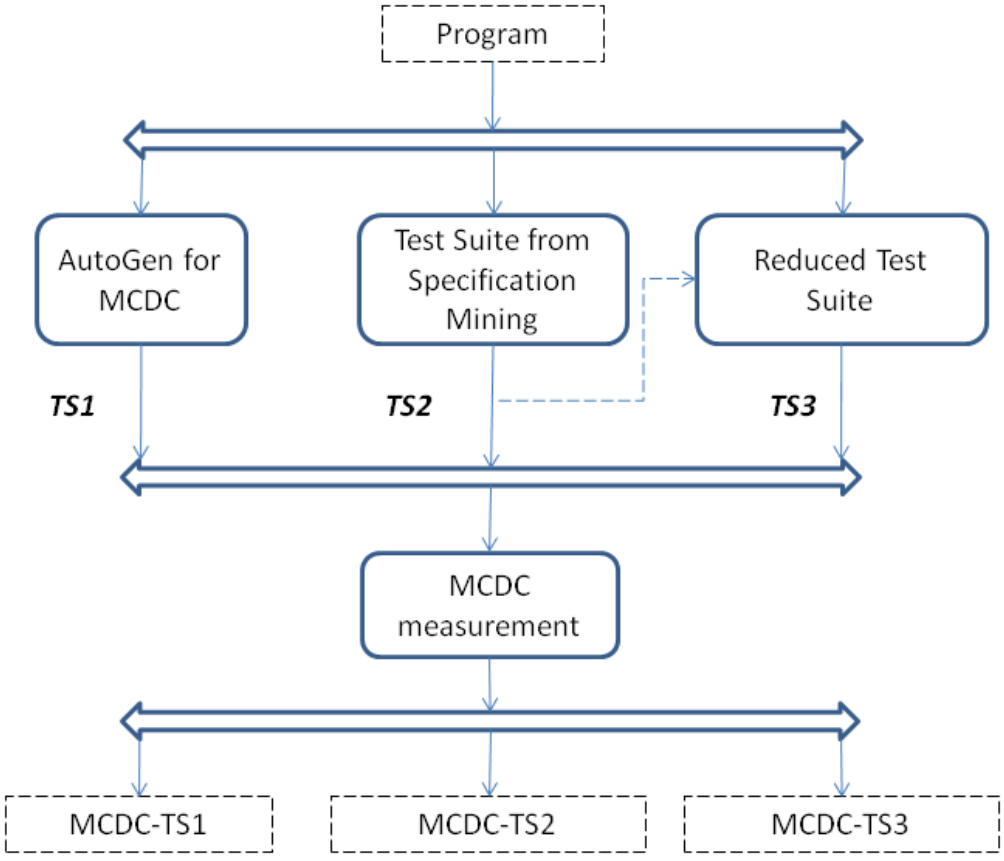

Figure 5.1: Test Suite Evaluation

Table 5.1: Details of Case Study

System	Wiper Control	Turn Indicator	Mem Slave5	Trans-mitter13	Token Ring13
LOC	53	986	752	787	808
Inputs	7	13	33	14	15
Outputs	5	142	45	70	72
Predicates	15	535	218	228	230

5.2 Case Studies

For our experimental analysis, we choose five relevant case studies. Two of the five case studies are from the automotive domain while three of them are from the Kratos benchmark suite [12]. The general characteristics of the programs have been presented in Table 5.1. We use `gcov` [1], a well known code coverage tool, to measure lines of code (LOC) and number of predicates of the programs. Explanation of each case study is as follows:

The first case study from the automotive domain is the wiper control module. This system controls the operation of the wiper in an automobile. It is a relatively small case study, but a timer variable present in the code adds complexity to test suite generation. The system requires fixed values as inputs for certain number of iterations before all parts in the code can be explored.

The second case study from the automotive domain is the turnIndicator system [69]. This system manages the turn indicator functionality of an automobile. UML models were available for the turnIndicator system. We auto-generated stubs from the UML model. Unfortunately, we were unable to generate code from the UML models using any of the freely available code

generators. So we manually wrote the code by extending the stubs generated from the model. An ideal step after this would have been to verify the exact functionality of the code and the UML models. This would have ensured that behaviour of the models and the code was identical. However, strict code correctness with respect to the model was not needed, since it was not relevant to our experimentation. Our aim was to generate tests for the code and not verify the code with models.

We chose the above two case studies since they closely depicted the nature of embedded systems. The next three case studies were from the Kratos benchmark suite [12] for testing performance of model checkers. The first of these benchmarks is the memSlave5 program. The next two case studies are the transmitter13 and tokenRing13 programs. Both these programs have a similar code structure. Overall, these three programs are part of memory management and transaction generation module of a network traffic controller system. They model an abstract bus with blocking input/output behaviour. All these programs are automatically generated and hence have a peculiar code structure. It is unlikely that practitioners would write code in such a manner. For instance, these programs use *goto* as their main control structure. This use of *goto*s makes program analysis difficult. These programs were chosen as AutoGen, our test generation tool, was unable to scale up for these programs to generate test data for MC/DC. For these programs, AutoGen would terminate before generating test data, even when the slicer was turned on and timeout value was set to 10 minutes per condition.

These programs are not very large with respect to lines of code. However, their code is non trivial and large percentage of code is predicates. Some involve floating point computations. Predicates and heavy computations generally make code analysis difficult. The programs exhibit reactive

behaviour which generates a large number of states.

5.3 Resources used in experimentation

We give a short information about the resources used in experimentation. All experiments related to AutoGen were performed on a standard Windows machine, Intel Core 2 Duo PC @ 2.27GHz, 2GB RAM. Other processes, like generation and reduction of test suite, were run on a Linux server, Intel Xeon CPU 32 bit, 8 processors, 8GB RAM. As we are not comparing time taken to conduct the experimentation the use of two different machines does not matter.

5.4 Experimentation

This section explains the actual experimentation process. The process of experimentation for the programs was as in Section 4. For each of the above programs, following steps were performed for test generation and evaluation.

1. Prerequisites:

   For experimenting with the programs, we needed four prerequisites as explained in Section 4: program code, list of input-output variables of the program, an initial test suite and a timeout value. Out of these prerequisites, only the program code was available to us. The rest of the things were either computed or determined.

   We compute list of input-output variables of the program. AutoGen, which is primarily a tool for test generation, has a feature to extract information of input and output variables from the program. It uses static analysis methods to determine the input and output variables.

The static analysis tool conservatively determines the variables. Hence there may be some inputs which are not actually inputs to the system, but are reported as inputs by AutoGen. Thus, after generation of these variables, the inputs need to be manually reviewed and some inputs are removed. This is a one time activity per program. This scenario is when the list of inputs is not known, which was in all of these case studies.

Ideally, an initial test suite should be a requirement-based test suite, that is one which tests the program for requirements. But, since such a test suite was not available, we generated a random test suite for every program and used it as an initial test suite. We choose a random test suite as a seed (initial) test suite since a random test suite is trivial to generate and it has been shown to be effective on embedded programs [55]. While doing random test generation, it is easy to introduce a bias towards a particular program. Some programs are computation intensive, like braking system, where majority of program code determines the amount of brake to be applied. These programs should ideally have large number of test cases with short sequences, to effectively test these systems. Other type of programs require large test sequences to effectively test them, like wiper control. This is usually the case for systems having timer variables, where code is executed depending on timer value.

So, while generating a random test suite, we need to ensure that a particular type of program is not favoured by our initial test suite. If favoured, the comparison of effectiveness across all programs will not be same. Thus, we generate every test suite with 100 test cases and with each test case having 100 iterations.

While determining a timeout value, we want to select a value which

is neither very small, nor very large. A small value will not allow our test generation technique to generate enough tests. A large value of timeout for test generation may not be acceptable in practice. Thus, for timeout, a fixed value of 20000 seconds was chosen for each program.

2. MC/DC test suite generation:

   AutoGen was used to generate an MC/DC satisfying test suite TS_1. For the first two case studies, AutoGen was able to generate such a test suite. However, for the next three programs of Kratos, AutoGen was unable to scale up. Thus we do not have MC/DC satisfying test suite TS_1 for the last three programs.

   To ensure fairness in test suite generation across all programs, we kept the same configuration of AutoGen for all programs. The configuration variables included timeout value per condition, number of maximum iterations of reactive loop and usage of slicer.

3. Test suite generation using specification mining:

   With inputs as provided in prerequisite 1, a new test suite TS_2 was generated using specification mining technique explained in Section 4.2. Except for transmitter13 system, the test generation technique for all systems was stopped due to timeout. For transmitter13, a complete FSM was obtained and hence the test generation was terminated. Note that, a complete FSM does not guarantee a complete specification, since a different input in one of the state may result in generation of a new state.

4. Test suite reduction:

   The test suite produced in the earlier step TS_2 was reduced using the

technique explained in Section 4.2 which gave a reduced test suite TS_3. As seen from the results, except for wiper system, the reduction technique was successful in reducing the test suite without significant losses in its effectiveness. For wiper system, the reduction of test suite size was 96%, however, the MC/DC effectiveness decreased almost 20%.

5. Comparison of test suite effectiveness:

   We compared the effectiveness of TS_1, TS_2 and TS_3 for the first two programs. As explained earlier, TS_1 was not available for the Kratos suite of programs. In that case, we compared TS_2 and TS_3 only. MC/DC was used as a measure of test suite effectiveness. Given a program and a test suite, AutoGen tool has a feature to compute the MC/DC for that test suite. Thus AutoGen was used to compare MC/DC of the programs for each of the test suites.

   Note that AutoGen uses a combination of static analysis (program slicing) and dynamic analysis (model checking) techniques, which are one of the best in their class. Also note that AutoGen is a resource intensive process, which consumes considerable time and memory for generation of test suite. Thus, AutoGen produces a maximum possible MC/DC satisfying test suite for a program.

   We do not wish to compete with AutoGen with respect to generation of an effective test suite. We use AutoGen generated test suite as a benchmark test suite to decide the normal number of test cases required to attain the maximum possible MC/DC for the program.

Table 5.2: Results of Experimentation

System	Test Suite Technique	Test Suite Size		MC/DC (%)
		Test Cases (#)	Test Vectors (#)	
Wiper	AutoGen MC/DC	11	194	96.6
	SpecMining	9989	122212	96.6
	Reduced	358	4812	76.6
TurnIndicator	AutoGen MC/DC	43	498	81.68
	SpecMining	2379	25536	75.79
	Reduced	1001	11378	75.79
MemSlave5	SpecMining	2001	21000	42.89
	Reduced	1001	11000	42.89
TokenRing13	SpecMining	2383	25748	67.39
	Reduced	1109	12549	67.39
Transmitter13	SpecMining	2178	23926	67.54
	Reduced	1262	13961	67.54

5.5 Observation of the Results

The results of our experiments are presented in Table 5.2. The table depicts results for the five case studies. For each case study, the table shows the MC/DC achieved by the test suite which is either generated by AutoGen (AutoGen MC/DC), generated from our specification mining technique (SpecMining) or test suite reduced from the test suite prepared by specification mining technique (Reduced). The number of test cases in each test suite, along with the total number of test vectors, is reported. Observations from the table are summarised below.

- Wiper control was a small case study for which AutoGen was able to generate a complete MC/DC satisfying test suite. The test suite originally produced by our technique was as good as the MC/DC satisfying test suite in effectiveness but had too many test cases. The reduction technique reduced the test suite by 96%, however, there was a considerable loss in MC/DC. Investigations in the code showed that a decision involving four operands (n) and three logical operators modified a single boolean output variable. To achieve MC/DC, there was a need for at least five test cases (n+1). These test cases were generated by AutoGen as well as our technique. However, from a black box perspective, all five test cases performed the same function, which was setting the value of that variable. Thus, our reduction algorithm deemed three of the five test cases as insignificant and discarded them. All five test cases take different paths in the program, but produce just two outputs. Our reduction technique does not have access to paths in the program and thus removes the remaining three test cases. This remains a limitation of our reduction algorithm and would be a limitation for most black

box test suite reduction techniques.

- AutoGen was able to produce 43 test cases covering 81.68% MC/DC for the TurnIndicator code. We assume this to be the maximum achievable MC/DC for that program. AutoGen took a relatively long time to generate these test cases (approximately 27500 seconds). Our technique generated a test suite with 2379 test cases which were reduced to 1001 test cases. The test suite generated by our technique was marginally inferior to the AutoGen generated test suite (i.e., less by 6% MC/DC). Another point to note is that the reduction algorithm halved the number of test cases while maintaining almost the same test suite effectiveness.

- For programs memSlave5, tokenRing13 and transmitter13, AutoGen was unable to scale up. For these programs, our technique was able to generate a test suite which had 42% MC/DC for memSlave5 and almost 67% MC/DC for the other two programs. The reduction technique eliminated 46% to 57% of the test cases based on our reduction technique, while keeping MC/DC unaffected. However, unlike the earlier programs, we do not have any benchmark measure of what coverage was achievable and what was achieved by our technique for these programs.

5.6 Lessons Learnt

Results of our experiments indicate that our technique can generate a *satisfactory* test suite which is comparable in MC/DC effectiveness with a MC/DC satisfying test suite. For cases where AutoGen scaled up, the

MC/DC achieved by the test suite generated by our technique was comparable to the MC/DC achieved by the AutoGen generated test suite. But, to achieve the same level of MC/DC, the number of test cases generated by our technique were much larger than those generated by AutoGen. For cases where AutoGen did not scale up, our technique was able to generate a test suite. We argue that effectiveness of this test suite is satisfactory as follows.

The advantages of our technique are that it is black box (code independent and does not require program analysis) and is scalable for large and complex codes. Thus we recommend that for embedded programs, when code is available and amenable to program analysis, one should use program analysis to generate test suite. For all other cases, our technique is a better option.

Aim of any ideal test suite reduction technique should be that the technique should reduce the size of test suite without compromising effectiveness of the test suite. Experimental results show that our test suite reduction technique reduces the test suite considerably without significant loss in effectiveness. Yet, there is scope for improvement on reducing the test suite further and on decreasing the small losses of effectiveness.

Our technique requires an initial test suite along with the program as input. However, none of our case studies had a test suite. Thus we had to build a test suite for each of the case studies. We selected a random test suite to be our seed test suite since it is easy to generate a random test suite. Also, based on our results from [55], we assumed that a random test suite would help in generating a useful specification. As seen from the results, test suite generated by our black box specification mining technique is comparable to the MC/DC test suite for most of the programs. Note that we measure effectiveness based on MC/DC. This concludes that random test suite as a

seed test suite helps generate a good specification for test suite generation.

Thus, we claim that the black box technique of test generation prepares an effective test suite for embedded programs.

5.7 Threat to Validity

This section explains the threat to validity of the success of our technique.

1. Choice of case study:

   While doing empirical analysis, we have chosen five case studies which, we believe, are representative of embedded domain. However, case studies chosen for experimentation may not be representative of embedded domain. Incase the case studies do not represent all of embedded domain programs, our sample programs may be insufficient samples. Since we have chosen a variety of programs, we assume that patterns depicted by these case studies is common in embedded programs.

2. Choice of test suite evaluation criteria:

   The ideal way of measuring test suite effectiveness is the ability of the test suite to detect number of errors in the program. Embedded programs with the error information were not available to us. As a result we had to use a different metric to measure test suite effectiveness. We could have used Mutation Kill Ratio (MKR) as a metric, which is shown to be the better metric at error detection than code coverage. However, none of the available mutation analysis tools for C programs could successfully analyse our case studies. Thus, we have chosen MC/DC, a code coverage criterion used for safety critical applications, as the metric for test suite evaluation. MC/DC does not guarantee error detection. But it is assumed that higher the MC/DC

of a test suite, more is its probability of detecting errors in programs. Analysis needs to be done to show if increase in MC/DC corresponds to increase in error detection in code.

3. Assumption of structure of embedded programs:
   Embedded systems are mostly reactive. We assumed that the FSM of its behaviours should be a connected graph, that is a graph where each state has an edge to at least one other state. It is a possibility that other embedded programs may not exhibit this behaviour of connected graphs. In those cases, our technique of test generation may not mine a good enough specification, hampering test suite generation. However, in our experience, the structure of programs in the case studies is representative of embedded programs.

4. Choice of test suite reduction technique:
   Even though the test suite reduction technique implemented by us almost halves the number of test cases, the size of the resulting test suite may still not be small enough for practitioners. This is evident when size of test suites generated by AutoGen and generated by our technique are compared. Our test suite reduction technique is very simplistic. A more sophisticated technique may reduce more test cases while maintaining test suite effectiveness.

5. Choice of timeout:
   We have chosen a fixed value of 20000 as the timeout value for our experiments. In four of the five case studies, timeout occurred while generating test data. There may be a different timeout value for which a better specification is mined. We have not experimented with different timeout values. However, our experiments show that a useful

specification can be mined with the current chosen value of timeout. So we choose not to experiment with different timeout values to answer our research question.

In what follows, we would revisit the question that we had started with and see if we have been able to find an answer satisfactorily. We would like to conclude that the results are indicative of the usefulness of our technique. We would also be highlighting some areas for future work in this direction.

6. Conclusion and Future Work

In Section 1.3 we had raised the question "Can black box techniques be as *effective* as white box techniques?". The reason we set out to find an answer to this is the advantages that black box testing enjoys over white box testing. In response to this question of ours, we have presented a black box technique for test suite generation of embedded programs, which is based on specification mining. The idea of specification mining is to extract specifications from existing systems, effectively leveraging the knowledge which is typically encoded as millions of lines of code. These specifications are models of software behaviour and can be used for building, verifying and synthesizing new or revised systems.

In our proposed technique, we first extract program specification using an initial test suite. These specifications are modelled as finite state machines, which consist of a set of states, an initial state, the transition relation and an input alphabet. These specifications are used to guide the process of generating more tests for the program. The tests are designed to explore untested behaviours of the program. After the new test suite has been generated, we attempt to remove redundant test cases from new test suite. Thus, our technique includes a test generation strategy as well as a test suite reduction method. To measure the effectiveness of the generated test suites, we per-

formed experiments on five case studies. The results indicated that our test suite reduction technique reduced the test suite size considerably, without a significant loss in effectiveness.

The effectiveness of the test suites was measured in terms of MC/DC code coverage. Coverage of test suites generated by the white box technique were compared with test suite with black box techniques. Results of the experiment indicate that effectiveness of test suites generated by white and black box technique are comparable, but the size of the test suites are very different. The test suite generated using white box technique is much smaller than that of black box technique. However, for programs where white box techniques do not scale, black box test suite gives 42% - 67% MC/DC code coverage. This code coverage may be acceptable for programs where white box techniques do not scale. Thus, experiments indicate that our technique is suitable for black box test suite generation in embedded programs.

In future, as an improvement to the existing work, we wish to experiment with a number of test suite reduction criteria and propose a reduction algorithm that does not affect the effectiveness of the test suite. Our current algorithm seems to remove some useful test cases while preserving some non-useful ones. Further, for test suite evaluation, it would be insightful to use a "different" metric - like *mutation kill ratio*. However, for the programs that we have considered in this work, there is no tool that would measure mutation kill ratio. While it will be an effort-intensive task to build such a tool or to modify our programs to suite the existing tools, we believe it would be worthwhile to do so.

As an extension of our work, we wish to explore new specification mining techniques in the black box domain to generate test suites with greater effectiveness. Currently, there is no known method to generate black box test

suites using specification mining techniques for reactive programs. There is a need to develop a class of such techniques and to formally validate them.

Bibliography

[1] GCOV - A Structural Code Coverage Measurement Tool.

[2] DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Technical report, 1994.

[3] ISO26262: Road vehicles – functional safety. Technical report, International Organization for Standardization, Geneva, Switzerland, November 2011.

[4] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems*, pages 188–204. Springer-Verlag, 2010.

[5] Larry Apfelbaum and John Doyle. Model Based Testing. In *Proceedings of the 10th International Software Quality Week Conference*, QW'97, pages 296–300, San Francisco, California USA, 1997. ACM.

[6] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 International*

Conference on Testing Software and Systems, ICTSS'10, pages 95–110, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: using Static Analysis to find Bugs in the Real World. *Communication ACM*, 53:66–75, Feb 2010.

[8] Purandar Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *CoRR*, cs.SE/0407038, 2004.

[9] B. Boehm and Hoh In. Identifying quality-requirement conflicts. *Software, IEEE*, 13(2):25–35, 1996.

[10] Prasad Bokil, Priyanka Darke, Ulka Shrotri, and R. Venkatesh. Automatic Test Data Generation for C Programs. In *Proceedings of the Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 359–368. IEEE Computer Society, 2009.

[11] John Joseph Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical report, Office of Aviation Research, 2001.

[12] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos benchmarks. *URL: https://es.fbk.eu/tools/kratos/index.php?n=Main.Benchmarks*.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREE Analyzer. In *Proceedings of the 14th Eu-*

ropean Symposium on Programming, ESOP'05, Edinburgh, UK, 2005. Springer LNCS 3444.

[14] Patrick Cousot and Radhia Cousot. Verification of Embedded Software: Problems and Perspectives. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)*, LNCS 2211, pages 97–113. Springer-Verlag, 2001.

[15] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating Test Cases for Specification Mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis(ISSTA)*, pages 85–96. ACM, 2010.

[16] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.

[17] Guido de Caso, Víctor Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program Abstractions for Behaviour Validation. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 381–390. ACM, 2011.

[18] Márcio Eduardo Delamaro and José Carlos Maldonado. *Proteum/IM 2.0: An Integrated Mutation Testing Environment.* Kluwer Academic Publishers, 2001.

[19] Richard A. DeMillo and A. Jefferson Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17:900–910, September 1991.

[20] Mark Dowson. The Ariane 5 Software Failure. *SIGSOFT Software Engineering Notes*, 22(2):84–94, March 1997.

[21] D. Edberg and L. Olfman. Organizational learning through the process of enhancing information systems. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 4*, HICSS '01, pages 4025–4035, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Jon Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28, October 1999.

[23] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes Theory Computer Science*, 217:5–21, July 2008.

[24] Lami G. QuARS: A tool for analyzing requirements. *Software Engineering Technical Report CMU/SEI-2005-TR-014*, September 2005.

[25] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th International Conference on Software Engineering*, pages 51–60. ACM, 2008.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[27] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 430–440, Washington, DC, USA, 2009. IEEE Computer Society.

[28] Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25:30–37, September 2008.

[29] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452. ACM, 2012.

[30] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, New York, NY, USA, 1975. ACM.

[31] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation using Constraint Solving Techniques. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 53–62, New York, NY, USA, 1998. ACM.

[32] Arnaud Gotlieb and Matthieu Petit. Path-oriented Random Testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 28–35, New York, NY, USA, 2006. ACM.

[33] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions Software Engineering Methodology*, 10(2):184–208, April 2001.

[34] Rothermel Gregg, Harrold Mary Jean, Ostrin Jeffery, and Hong Christine. An empirical study of the effects of minimization on the fault

detection capabilities of test suites. Technical report, Corvallis, OR, USA, 1998.

[35] Michaela Greiler, Arie van Deursen, and Andy Zaidman. Measuring test case similarity to support test suite understanding. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 91–107, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] Wolfgang Grieskamp. Multi-paradigmatic Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin / Heidelberg, 2006.

[37] M. Grottke and K.S. Trivedi. Fighting bugs: remove, retry, replicate, and rejuvenate. *IEEE Computer 40(2)*, 40(2):107–109, 2007.

[38] Dick Hamlet. When Only Random Testing Will Do. In *Proceedings of the 1st International Workshop on Random testing*, pages 1–9. ACM, 2006.

[39] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving Test Suites via Operational Abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 60–71, Washington, DC, USA, 2003. IEEE Computer Society.

[40] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transaction Software Engineering Methodology*, 2(3):270–285, July 1993.

[41] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transaction Software Engineering Methodology*, 2(3):270–285, July 1993.

[42] Mats P. E. Heimdahl and Devaraj George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 176–185, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Mats P. E. Heimdahl, Devaraj George, and Robert Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria. In *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering*, HASE'04, pages 178–186, Washington, DC, USA, 2004. IEEE Computer Society.

[44] Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-driven Software. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 597–607. ACM, 1999.

[45] J.R. Horgan and S. London. A data flow coverage testing tool for C. In *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools, 1992*, pages 2–10. IEEE, 1992.

[46] JeeHyun Hwang, Tao Xie, Donia El Kateb, Tejeddine Mouelhi, and Yves Le Traon. Selection of regression system tests for security policy evolution. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 266–269, New York, NY, USA, 2012. ACM.

[47] Antti Jääskeläinen, Antti Kervinen, Mika Katara, Antti Valmari, and Heikki Virtanen. Synthesizing Test Models from Test Cases. In *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, pages 179–193. Springer-Verlag, 2009.

[48] Prateek Jain, Kunal Verma, Alex Kass, and Reymonrod G. Vasquez. Automated review of natural language requirements documents: generating useful warnings with user-extensible glossaries driving a simple state machine. In *Proceedings of the 2nd India software engineering Conference*, ISEC '09, pages 37–46, New York, NY, USA, 2009. ACM.

[49] Dennis Jeffrey and Neelam Gupta. Test suite reduction with selective redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 549–558, Washington, DC, USA, 2005. IEEE Computer Society.

[50] Yue Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, pages 94–98, 2008.

[51] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 92–102, Washington, DC, USA, 2001. IEEE Computer Society.

[52] Kalpesh Kapoor and Jonathan Bowen. Experimental Evaluation of the Variation in Effectiveness for DC, FPC and MC/DC Test Criteria. In

Proceedings of the 2003 International Symposium on Empirical Software Engineering, pages 185–194. IEEE Computer Society, 2003.

[53] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, NASA Langley Center, 2001.

[54] Shubhangi Khare, Sandeep Saraswat, and Shrawan Kumar. Static program analysis of large embedded code base: An experience. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 99–102, Thiruvananthapuram, Kerala, India, 2011. ACM.

[55] Padmanabhan Krishnan, R. Venkatesh, Prasad Bokil, Tukaram Muske, and Vijay Suman. Effectiveness of random testing of embedded systems. In *Proceedings of the 2012 45th Hawaii International Conference on System Sciences (HICSS)*, pages 5556–5563. IEEE Computer Society, 2012.

[56] Shuvendu Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 509–524. Springer Berlin / Heidelberg, 2009.

[57] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet? In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, pages 95–104. IEEE Computer Society, 2009.

[58] Claire Le Goues and Westley Weimer. Measuring code quality to improve specification mining. *IEEE Transactions in Software Engineering*, 38(1):175–190, January 2012.

[59] Nancy G. Leveson. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26:18–41, 1993.

[60] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 220–229. IEEE Computer Society, 2009.

[61] Patrick Munier. *Polyspace*, pages 123–153. John Wiley & Sons, Inc., 2012.

[62] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-level Models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[63] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[64] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 373–386. ACM, 2012.

[65] A. Jefferson Offutt. A Practical System for Mutation Testing: Help for the Common Programmer. In *Proceedings of the 1994 International Conference on Test*, pages 824–830. IEEE Computer Society, 1994.

[66] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, June 1995.

[67] A. A. Omar and F. A. Mohammed. A survey of software functional testing methods. *SIGSOFT Software Engineering Notes*, 16(2):75–82, April 1991.

[68] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communication ACM*, 31(6):676–686, June 1988.

[69] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev, and Cornelia Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In *Proceedings of the 23rd IFIP WG 6.1 International Conference on Testing Software and Systems(ICTSS)*, pages 146–161. Springer-Verlag, 2011.

[70] Robert M. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[71] Reactis. White Paper - Finding Bugs in C Code with Reactis for C, August 2011.

[72] Stuart C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings of the 4th*

International Symposium on Software Metrics, METRICS '97, pages 64–, Washington, DC, USA, 1997. IEEE Computer Society.

[73] VDC research survey. http://blog.vdcresearch.com/embedded_sw/2011/06/2011-embedded-engineer-survey-results-programming-languages-used-to-develop-software.html. 2011.

[74] Xavier Rival. Understanding the origin of alarms in astree. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 303–319, Berlin, Heidelberg, 2005. Springer-Verlag.

[75] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification, and Reliability*, 12:219–249, 2002.

[76] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static Specification Mining using Automata-based Abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 174–184. ACM, 2007.

[77] Mary Lou Soffa, Aditya P. Mathur, and Neelam Gupta. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ASE '00, pages 219–, Washington, DC, USA, 2000. IEEE Computer Society.

[78] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. On the danger of coverage directed test case generation. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 409–424, Berlin, Heidelberg, 2012. Springer-Verlag.

[79] Rajiv Ranjan Suman and Rajib Mall. State model extraction of a software component by observing its behavior. *SIGSOFT Software Engineering Notes*, 34:1–7, January 2009.

[80] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 35–42, Lisbon, Portugal, 2005. ACM.

[81] GAIO technologies. Coveragemaster winams - http://www.gaio.com/product/dev_tools/pdt07_winams.html.

[82] Princeton University. How to design a finite state machine - http://www.cs.princeton.edu/courses/archive/spr06/cos116/fsm_tutorial.pdf.

[83] A. Van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[84] Tanja E. Vos, Felix F. Lindlar, Benjamin Wilmes, Andreas Windisch, Arthur I. Baars, Peter M. Kruse, Hamilton Gross, and Joachim Wegener. Evolutionary functional black-box testing in an industrial setting. *Software Quality Control*, 21(2):259–288, June 2013.

[85] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[86] Eric Wong, Joseph R. Horgan, Aditya P. Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Proceedings of the 21st Annual International Computer Software and Applications Conference*, pages 522–528, 1997.

[87] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[88] Xianming Wu, J. Jenny Li, David M. Weiss, and Yann-Hang Lee. Coverage-Based Testing on Embedded Systems. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 31–36, Washington, DC, USA, 2007. IEEE Computer Society.

[89] Andreas Zeller. Specifications for Free. In *Proceedings of the Third International Conference on NASA Formal methods*, pages 2–12. Springer-Verlag, 2011.

[90] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis(ISSTA)*, pages 353–363. ACM, 2011.

[91] Jiang Zheng. In regression testing selection when source code is not available. In *Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering*, ASE '05, pages 752–755, New York, NY, USA, 2005. ACM.

[92] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 269–280, Washington, DC, USA, 2004. IEEE Computer Society.

[93] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computer Survey*, 29(4):366–427, December 1997.