1-1-2015

Fail-Safe Testing of Web Applications

Salah Boukhris
University of Denver

Fail-Safe Testing of Web Applications

A Dissertation

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

in Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Salah Boukhris

August 2015

Advisor: Anneliese Andrews

Author: Salah Boukhris
Title: Fail-Safe Testing of Web Applications
Advisor: Anneliese Andrews
Degree Date: August 2015

Abstract

This dissertation introduces an approach to generate tests to test fail-safe behavior for web applications. We apply the approach to a commercial web application. We build models for both behavioral and mitigation requirements. We create mitigation tests from an existing functional black box test suite by determining failure type and points of failure in the test suite and weaving required mitigation based on weaving rules to generate a test suite that tests proper mitigation of failures. A genetic algorithm (GA) is used to determine points of failure and type of failure that needs to be tested. Mitigation test paths are woven into the behavioral test at the point of failure based on failure specific weaving rules. A simulator was developed to evaluate choice of parameters for the genetic algorithm. We showed how to tune the fitness function and performed tuning experiments for GA to determine what values to use for exploration weight and prospecting weight. We found that higher defect densities make prospecting and mining more successful, while lower mitigation defect densities need more exploration. We compare efficiency and effectiveness of the approach. First, the GA approach is compared to random selection. The results show that the GA performance was better than random selection and that the approach was robust when the search space increased. Second, we compare the GA against four coverage criteria. The results of comparison show that test requirements generated by a genetic algorithm (GA) are more efficient than three of the four coverage criteria for large search spaces. They are equally effective. For small search spaces, the genetic algorithm is less effective than three of the four coverage criteria. The fourth coverage criteria is too weak and unable to find all defects in almost all cases. We also present a large case study of a mortgage system at one of our industrial partners and show how we formalize the approach. We evaluate the use of a GA to create test requirements. The evaluation includes choice of initial population, multiplicity of runs and a discussion of the cost of evaluating fitness. Finally, we build a selective regression testing approach based on types of changes (add, delete, or modify) that could occur in the behavioral model, the fault model, the mitigation models, the weaving rules, and the state-event matrix. We provide a systematic method by showing the formalization steps for each type of change to the various models.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Andrews, for her excellent guidance, caring, patience, as well as providing me with valuable advise for doing research. I would never have been able to finish my dissertation without the guidance of Prof. Andrews.

My thanks also go to the members of my major committee, Dr. Matthew Rutherford and Dr. Rinku Dewri for reading previous drafts of this dissertation during my preliminary exam and providing many valuable comments that improved the presentation and content of this dissertation. I am deeply grateful to them for the long discussions that helped me sort out the technical details of my work.

Special thanks go to Dr. Linda Bensel-Meyers, who was willing to participate as an external chair in my final defense committee at the last moment.

I would like to present my sincere thankfulness to my dear mother and my deceased father, who died on December 26, 1997, for their great role in my life and their numerous sacrifices for me and for my brothers and sisters.

Last but not least, I would like to thank my wife for her understanding and love during my years of study and taking care of my four kids Nosiba, Tasnim , Suhayb, and Suhail. Her support and encouragement cheered me up and stood by me through the good times and bad. Also, grateful thanks to my brothers and sisters and all my family and relatives who receive my deepest gratitude and love for their dedication and the many years of support during my study that provided the foundation to continue this journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web applications have became the backbone of today's business life through e-commerce and the communications industry. According to [24], in 2008, 73% of the population used the internet in the United States resulting in over \$204 billion dollars in on-line sales. This also implies that failures in critical web applications could result in losses of millions of dollars. For example, eBay lost more than \$3 million in customer credits and \$4 billion in market capitalization because of 22 hours of system outage. The real cost of system failure is in lost revenue, frustrated customers, and the negative impact on a company's value [59].

This makes it imperative that an adverse impact of failures be avoided. In other words, external failures such as a server or database crash must be properly mitigated. The web application must also be tested whether mitigations work according to requirements.

Unfortunately, existing Model-Based Testing (MBT) techniques emphasize functional testing, but not the testing of proper failure mitigation. Given the large potential losses these failures can carry when they are not properly mitigated, testing failure mitigation is quite important. It is conceptually possible to add failure transitions and mitigation behavior to an existing functional model and then use whatever MBT has been developed for it. One strategy for testing fail-safe behavior alongside functional behavior is to integrate fault models with behavioral models: Marisa et al. [28], [70] integrate Fault Trees (FT) and State Charts (SC) while HyeonJeong et al. [41] integrates UML State Diagrams and FTs. These approaches have their limitations and challenges: Possible mismatches between notations and terminologies used in the behavioral vs. the fault models, requiring a step to make them compatible. In the not so unusual case where a decent number of failures can occur in many behavioral states, when multiple fault trees exist and when mitigation behaviors themselves are non-trivial, this can lead to large, cumbersome, highly connected models that obscure primary functionality. In other words, this can lead to scalability problems. These approaches also cannot leverage an existing test suite. Finally, there is no explicit mitigation model or mitigation patterns. Hence integrating failures and their mitigation into existing models has its drawbacks.

This dissertation proposes a method to enhance an existing MBT technique, FSMWeb [9] that leverages a test suite derived from the model and transforms it into a series of mitigation tests for various failures. This does not require merging primary functional behavior descriptions with mitigation behavior in the face of failures. Rather, mitigation tests are woven into various stages of an existing behavioral test at various points of failure based on weaving rules specifically defined for each mitigation type. This is not unlike weaving code aspects into primary code in aspect-oriented programming [33], although the specific weaving rules differ, of course and we are weaving mitigation test aspects into behavioral tests rather than working with code.

Besides avoiding potential model bloat, this approach also has the advantage that we can proceed with functional testing as usual, so existing test suites do not have to be regenerated, failures can be injected at selected points in the existing test suite, and a mitigation test is created by modifying the functional test at the point of failure with required mitigation behavior.

Web applications also tend to evolve overtime. This requires regression testing of the software, including proper failure mitigation. Rather than testing the entire web application from scratch, it would be useful to have a selective regression testing of proper failure mitigation.

This dissertation has as its major goal to develop a black-box MBT technique for web applications that is able to:

- Leverage an existing testing approach for web applications (FSMWeb [9])
- Systematically models required mitigation behavior for failures via mitigation patterns.
- Generates fail-safe mitigation test requirements (via a genetic algorithm).
- Builds a fail-safe mitigation test suite.
- Develops a selective regression test suite.

We also validate this approach via a case study and simulation experiments and compare it to other approaches.

This dissertation is a combination of different ideas that are related in testing web application and background to build our models (see Figure 1.1):

- Model-based testing:
  - Using finite state machine (FSM) testing [21] [37] [38] [60].
  - FSM in modeling the design of web application [42] [48], especially FSMWeb [9].
  - Testing web applications [64] [66] [50] [9] [8] [61].

- Fail-safe behavior:
  - Failure types and classifications [3] [59] [10] [51] [34].
  - Failure-Exception handling patterns [77] [49] [15] [18].
  - Exception handling patterns for process and work flow models [46] [35] [55] [31] [71].
  - Testing Exception handling [56] [39] [69].
  - Testing Fail-safe behavior [72] [27] [70] [41] [28].
- GA and software testing [2] [58] [68] [53] [13] [74] [54] [43] [23] [75] [40].

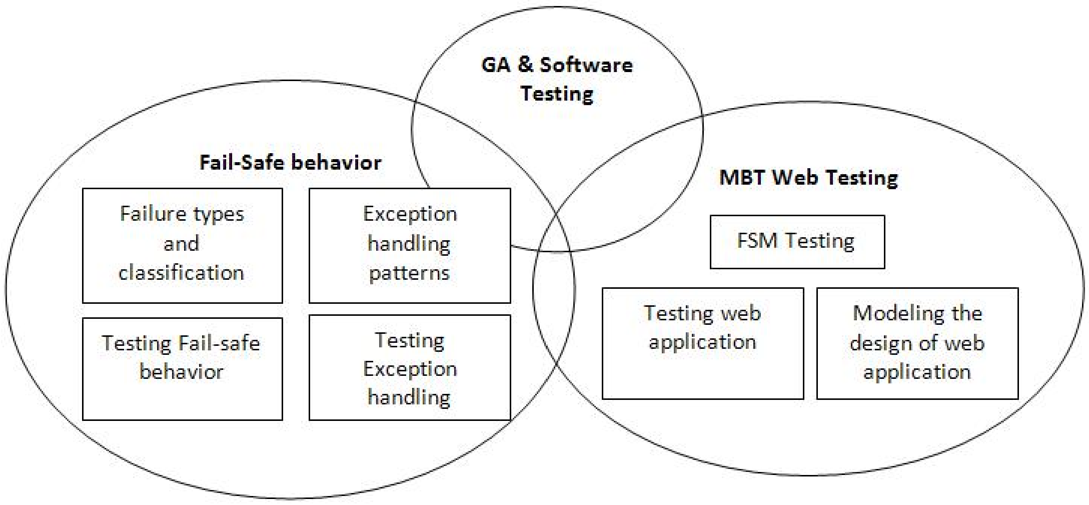

GA & Software Testing
Fail-Safe behavior
Failure types and classification
Exception handling patterns
Testing Fail-safe behavior
Testing Exception handling
MBT Web Testing
FSM Testing
Testing web application
Modeling the design of web application

Figure 1.1: Topics related to our approach.

Table 1.1 illustrates how this work is related to similar work [6], and describes the overlapping areas in more detail.

Table 1.1: Collaboration: Comparison, Basic Strategy

	Domain	Model	Test Requirements	Regression testing	Tools	Experiments
Other work [6]	Safety-critical	(C) EFSMs	Coverage Criteria	Basic Strategy	None	Comparison (GA,CC)
This work	Web App.	FSMWeb	GA	Basic Strategy/ Full Formalization	Simulator	Comparison (GA,CC, R), Tuning GA, Evaluate initial Pop. vs. R, Single vs. multiple runs

This dissertation is organized as follows: Chapter 2 describes related work in functional testing of web applications, fault and failure taxonomies for web applications, exception handling as a means of failure mitigation and general approaches to test fail-safe behavior. We also discuss the existing work in Genetic Algorithms (GA) used to find defects in software via black box testing, since part of our approach uses a GA. Chapter 3 describes our MBT approach to fail-safe testing of critical web applications, including a testing process. Chapter 4 evaluates the performance of the GA approach with a series of experiments. Chapter 5 presents a comparison with respect to effectiveness and efficiency (using GA, coverage criteria [6] or random selection to generate mitigation test requirements). Chapter 6 applies the approach to a major case study. Regression testing is introduced in Chapter 7. Chapter 8 suggest further work. Chapter 9 draws conclusion.

Chapter 2

Background

2.1 MBT Web testing

Two major techniques have been proposed for functional (black box) testing of web applications, Di Lucca et al. [50] and Andrews et al. [9, 8, 61]. The first revolves around the creation and use of decision tables. It is suitable for unit and integration test. The second is based on a hierarchical collection of compressed FSMs that, in conjunction with a test database, allow both integration and subsystem testing. The FSMWeb model combines both behavioral and structural characteristics (as defined in Di Lucca et al. [50]), since it models web inputs and navigation between pages, in addition to behavioral characteristics. Its attempts at compression make it a desirable candidate to investigate for fail-safe testing of web aplications. The FSMWeb approach falls into the larger category of FSM based techniques used for testing. Testing with FSM models has a long and varied history [21, 37, 38, 60]. In addition to FSM models used for testing, there is also extensive work on testing FSMs for correct behavior. For a survey on the latter see David and Mihalis [44]. Other methods use non-FSM models for testing web applications such as using a UML meta model [64], but the model neither validates the scalability nor clarifies the difference between static web sites and dynamic web sites. Some other studies use user session data for initial test data generation such as in Sreedevi et al. [66]. FSM-based test generation has been used to test a large number of application domains including compilers, real-time process control systems, networking, data processing, and telephony. FSMs have also been used to model the design of web sites and web applications [42, 48]. Kung et al. [42] propose to generate test cases from an object-oriented test model that uses FSMs to model functional behavior.

A key limitation of many of these finite state approaches is state explosion which limits their scalability. FSMWeb addresses this issue through FSM compression with savings of orders of magnitude in size [8]. This advantage is one of the reasons why we chose to investigate the use of FSMWeb for fail-safe testing.

2.2 Fail-Safe Behavior

As defined in Ammann and Offutt [3], a fault is a static defect in a software artifact, e.g. incorrect instructions or requirements. Failures can also be events that are external to the software under test, such as a database crash, loss of network connection, or unavailability of a server on which the software depends. Fail-safe mitigation testing is interested in the latter.

Several papers try to classify failures or provide a fault taxonomy, some associate required mitigation actions with types of faults or failures. Pertet and Narasimhan [59] address causes and effects of web failures. Most failures are caused by: software failures, human/operator errors, hardware/environmental failures, and security violations. Their effects are unavailable systems, exceptions, access violations, incorrect answers, data loss and corruption, and poor performance. Several papers have attempted to classify faults in web applications. Ardagna et al. [10] classifies web fault types into infrastructure and middle-ware faults, web service faults, and web application faults. In addition, they specify types of recovery actions as retry services, substitute services, completion of missing parameters, reallocation of services and changing process structure. Similarly, Ma and Tian [51] categorize web service failures as host, network, or browser failures, source or content failures, and user errors. Underlying error types include permission denied, no such file or directory, stale NSF file handle, etc. Guo and Sampath [34] consider logic faults and compatibility faults. Logic faults include subcategories: browser interaction faults, session faults, paging faults, server-side parsing faults, etc. While these papers are useful in classifying web faults and failures, only one ([10]) considers both failures and recovery.

Zeng et al. [77] addresses recovery in the form of exception management for composite web services. They consider application exceptions and process defined exceptions. A service can be unavailable or fail. Service execution can be delayed, time out, or experience QoS degradation. Recovery action types include retry, skip, replace, try alternative, compensate, and time out. Lu et al. [49] provides a formal definition of exceptions and exception handling policies for state charts. The policies include skip, abort, retry, try alternative, compensate, replace, and timeout. Finally, Brambilla et al. [15] classifies exceptions as user-generated, application generated or infrastructure related. Exception handling follows five policies: accept, reject, abort, ignore, and resume. Cabral and Marques [18] provides an analysis of how often Java and .Net applications use the following types of exception handlers: empty, log, alternative, throw, continue, return, rollback, close, assert, delegates, and others.

In addition to recovery and/or exception handling policies (which are almost always informally defined), more formal exception handling patterns have been defined for process and work flow models ([46],[35],[55],[31],[71]). However, none of these address testing exception handling (or fault/failure mitigation). In Oprisa [56] an exception handling FSM is constructed and the W-method is used to generate test cases. Jiang and Yuanpeng [39] construct an exception control flow graph and use DU coverage criteria on variables related to an exception. Sinha and Harrold [69] also investigates white box testing of programs with exception handling constructs using control flow and data flow analysis. However, no exception handling policies or patterns are defined in these studies. Moreover, these categorizations overlap without being comprehensive or consistent in the absence of a set of widely accepted failure-mitigation testing models. Unmitigated or improperly mitigated failures can be costly.

2.3 Testing Fail-Safe Behavior

Fault Tree Analysis (FTA) is commonly used in safety critical system analysis to recognize the potential causes of unsafe conditions. FTA is a top-down deductive analysis technique used to detect the specific causes of possible hazards [72][27]. The top event in a fault tree is the system hazard. FTA works downward from the top event to determine potential causes of a hazard. It uses boolean logic to represent these combinations of individual faults that can lead to the top event [27]. However, Constructing a Fault Tree (FT) can be a time consuming task[41]. One strategy for testing fail-safe behavior alongside functional behavior is to integrate fault models with behavioral models: [28] and [70] integrate State Charts and FTs, while [41] integrates UML State Diagrams and FTs. These approaches have a variety of limitations and challenges. These include: (1) possible mismatches between notations and terminologies used in the FT vs. the behavioral model, requiring a step to make the FT and the behavioral model compatible; (2) potential scalability problems when multiple or large FTs exist or a fault can occur in a large portion of behavioral states; (3) they cannot leverage an existing behavioral test suite; and (4) there is no formal mitigation model, nor exception handling patterns.

2.4 Genetic Algorithms and Software Testing

Genetic algorithms and Genetic programming are based on Evolutionary Algorithms (EA). These algorithms are soft computing techniques inspired by Charles Darwin's principle of the survival of the fittest [63]. There are four well-established main types of EA and most widely used techniques: Genetic Algorithms (GA), Genetic Programming (GP), Evolution Strategy (ES), and Evolutionary Programming (EP) [1]. Genetic algorithms (GA), which were originally developed by John Holland in the 1960s, are search algorithms based on natural selection and natural genetics. Genetic programming (GP) was introduced by John Koza [63], who had the idea to allow a computer to solve problems without being explicitly programmed to do so. Genetic programming can be seen as an expansion of Genetic algorithms. GP creates a computer program as the solution while GA will create a string that represents the solution [63]. Evolution strategy (ES) and Evolutionary Programming (EP) allow varing length of individuals in the population; the selected individuals are subjected to mutation to produce children with no crossover [63]. The main differences between the four kinds of algorithms are the representation of the solutions and the use of variation operators [1].

GA has been used in different areas such as optimization, automatic programming, machine learning, economics, immune systems, ecology, population genetics, evolution and learning, and social systems [63]. The main goal of GA is to develop successive generations of ever better combinations of parameters which improve the overall solution. The GA as adaptive search technique is not guaranteed to find the optimal solution, however it often finds a good solution in a brief time [63].

Algorithm 1: The pseudo code for GA

```
Require: Search Space
Ensure: Finding Solution
  population representation;
  Initialize(population);
  Evaluate(population) using Fitness function;
  while stopping condition not satisfied do
    Selection(population)
    Crossover(population)
    Mutate(population)
    Evaluate(population)
  end while
```

Figure 2.1: The pseudo code for GA

The GA usually performs the following cycle [63] [52] (see a basic algorithm for a GA as shown in **Algorithm 1**):

1. A randomly initialized population of individuals is generated. Each individual is usually represented as a string of bits called a chromosome. Each bit is a gene. A chromosome is a possible candidate solution of a given problem.
2. A fitness function is associated with each individual. It is used to evaluate the adequacy and quality of an individual.
3. A selection process is used to extract a subset from the current population.

4. A new generation is created by a crossover operation that takes two individuals and exchanges their information at a randomly selected position.

5. In order to prevent individuals from becoming too similar, a mutation process is applied by randomly modifying some information of a selected individual.

6. Each individual of the new population is evaluated again, and the procedure is repeated (step 2) until a specific termination condition is fulfilled.

The key to the successful use of a GA is to represent the problem as well as its solution in terms of a chromosome. The most commonly used representation is a binary string. However, representations with a more complicated data structure have been used [63]. Crossover is used to increase the quality of the reproduction populations and force convergence while mutation processing guarantees the entire search space will be searched and restores lost information or adds information to the population. The balancing between exploration where bad solutions have a chance to go to the next generation, and exploitation where good solutions go to the next generation more often than bad ones, is more important within the mechanism of the selection. Different selection strategies significantly affect the performance of the GA [62]. Tournament selection, roulette wheel, and rank-based roulette wheel selection are the most common selection schemes. In tournament selection, the number of individuals are selected randomly from the population, and the selected ones will compete against each other based on the highest fitness to be included in the next generation. In proportional roulette wheel, the selection of the population is based on a proportion of their fitness values which corresponds to a portion of a roulette wheel. Finally, in rank-based roulette wheel selection, the selection is based on its fitness rank relative to the whole population and a selection probability according to their ranks instead of their fitness values.

The use of GA in software testing is known as a type of search-based software testing (SBST) that includes many other meta-heuristic optimizing search techniques. Meta-heuristic search algorithms have been used to automate a variety of software testing activities such as test case generation, test case selection, and test case prioritization. There are several reasons that make the use of GA popular in software testing [2]. First, based on the survey paper Harman et al. [36], GAs have been widely studied, experimented and applied in software testing. Large amounts of empirical data are available for different parameter settings. This helps to select the appropriate parameters for solving a specific problem making it easier for researchers to learn how to adapt a GA to a given problem [2]. Second, the performance of GA has shown better results than local search algorithms, although there is no proof indicating that GA outperforms other global search algorithms [2]. Finally, GA has many well-known implementations and resource tools that significantly facilitate their practical application [2].

Ali et al. [2] analyze the use and results of search-based algorithms in test generation, including Genetic Algorithms (GA). They report that the majority of techniques (78%) have been applied at the unit level and do not target specific faults (as we are interested in), but focus on structural coverage criteria like node, branch, or path coverage. By contrast, our goal is to target specific fault types and to test at various points of the test suite whether mitigation works properly.

Patton et al. [58] suggest a GA approach for focused software usage testing. Their goal is to test based on usage frequency with the objective of finding failures and then suggest further similar, but different test cases that reveal faults. Their work is inspired by the failure pursuit sampling of Schultz et al. [68]. The latter generated fault scenarios for testing intelligent controllers for autonomous vehicles.

Patton et al. try to maximize two objectives: the likelihood of occurrence (usage frequency) and failure intensity (consisting of a combination of failure density and severity). This requires a multi-objective GA technique. They avoid creating single dominant individuals using niching [53]. A niche represents a subpopulation that is similar, but different.

Berndt and Watkins [13] and Watkins et al. [74] also introduce a multi-objective fitness function that changes based on results from previous testing cycles, i.e a relative fitness function changes as the population evolves, based on the knowledge gained from prior generations (the "fossil record"). Individuals are rewarded based on novelty, proximity, and severity. Novelty encourages new areas of the search space to be explored, regardless of the type of error generated. Proximity measures how close an individual is to previously found defects. Severity measures its impact valuing detection of defects with severe consequences more highly. Three types of tests are generated: explorers (high uniqueness), prospectors (some uniqueness, but also close to a found defect), and miners (close to found defects). McCart et al. [54] attempt to reduce the computational burden of the fitness function calculation (such as distances from entries in the fossil record and distance calculations from individuals in the fossil record that detected errors) by using sampling, adjusting the frequency of sampling, defining defect boundaries and keeping individuals with similar distance to the origin in a bin.

While this test generation approach has only been used for system testing of complex distributed systems via long sequence testing, its goals have similarities with our objectives and we decided to see whether a similar strategy could help in selecting positions in a test suite and failure types to inject at that position so as to generate tests that explore the search space, prospect in the larger vicinity of found mitigation defects, and mine for further mitigation defects in the immediate vicinity of a found defect. It is worth exploring whether a similar strategy that combines exploring with prospecting and mining using a fossil record might be useful for our purposes.

Last et al. [43] describe an extension of GA algorithms that uses a varying probability of crossover depending on the age of an individual (young, middle, old). Age is represented as a Fuzzy Logic function. Very young and old off-spring have a lower probability of crossover, thus enabling more exploration in the young and avoiding a local optimum due to premature convergence in the old. A Fuzzy Rule base directs crossover of mixed age populations. The inputs to the programs under test are 100 Boolean expressions using AND, OR, and NOT. A single error is injected by randomly selecting an AND or OR operator and flipping it. Test cases consist of 100 bits (one for each expression). The fitness function of a test is one, if it can distinguish between the correct and incorrect expression, zero otherwise. Experiments showed an increased capability of recognizing the erroneous expression. While our testing problem is very different, this approach represents another way to distinguish between exploration and searching in the vicinity.

A series of papers [23, 75, 40] have addressed generating tests for a path through an EFSM using GA, such as Kalaji et al. [40]. Since we assume that the test suite exists, this is less applicable to our problem. However, any of a number of search-based or non-search based technqiues could be used to find inputs for the mitigation paths once they are woven into the behavioral test suite (cf. Chapter 2.1). Partial regeneration as suggested in [4] from the point of failure injection is also an option.

2.5 Regression Testing

Web applications maintenance cycles are similar to those of other software systems. They include corrective, perfective, or adaptive maintenance as well as product evolution through enhancements. The Standard for Software Engineering-Software Maintenance, ISO/IEC 14764 [14] includes four categories of maintenance:

- Corrective maintenance: unscheduled modification to correct discovered faults of a software product in order to keep a system operational, e.g. the removal of errors in the code.
- Adaptive maintenance: modification of the software product due to changes in the environment or changes to requirements, e.g. the modification of software for a new operating system.
- Perfective maintenance such as improving efficiency and performance to prevent problems in the future, e.g. modification of code to improve performance.
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults to prevent malfunctions in the future.

ISO/IEC 14764 [14] classifies adaptive and perfective maintenance as enhancements, and groups together the corrective and preventive maintenance categories into a correction category. Regression testing is a very vital task after each maintenance round in order to verify and validate the modified web application and ensure that both new and existing features are working properly. This process is often done with existing test cases from previous release(s). Regression testing can be very expensive especially for very large web systems, for which a retest all approach [73] is too costly. Corrective maintenance does not change any of the models used in our approach. Hence, one could simply rerun the entire fail-safe test suite. Given its expense, this wastes resources. A better approach is to trace model elements to code modifications and execute tests that include these model elements only. The other types of maintenance activities cause model changes. This can cause some test paths and associated test cases to become obsolete. In addition, new tests may have to be generated to cover added model elements. Rothermel and Harrold have defined five steps for selective regression testing [65]:

- "*Select $T' \subseteq T$ (T' is a set of test cases after modification of the program P' , and T is a test suite).*
- *Test P' using T', establishing P''s correctness with respect to T'.*
- *If necessary to achieve coverage requirements, create a set of new test cases T'' for P'.*
- *Test P' with T'', establishing P''s correctness with respect to T''."*

In our case, depending on which parts of our models change, T can be the behavioral test suite BT, mitigation test suite MT_j, or even the failure mitigation test suite FMT and hence T', T'' can be also refer to any of these. This dissertation describes a method for selective regression testing of fail-safe testing in web applications based on the test generation methodology used in chapter 3. We also adopt the classification of test cases and the formalization of the changes to the FSMWeb models based on Leung and White [47] and Andrews et al. [7]. The framework is based on classifying behavioral tests as retestable, obsolete, and reusable. Retestable tests are those that are still valid and test portions of the application that may be affected by the change. Obsolete tests are those that are no longer applicable. Behavioral tests that do not test software modification and produce the same result are reusable tests.

Andrews and Do [4] explain in detail how to use partial regeneration for an FSMWeb model. The goal is to replace obsolete test cases with tests without full regeneration. In the study, they define thresholds to determine whether partial or full regeneration is warranted. Our goal here is to also develop an approach for partial regeneration of obsolete tests.

Chapter 3

Approach

3.1 Process

The black box test generation process for testing fail-safe behavior consists of the following steps (see Figure 3.1)

1. Generate test cases from the behavioral model (section 3.2).
2. Identify failure events and their mitigation (section 3.3).
3. Generate mitigation tests from the mitigation models (section 3.4).
4. Generate test requirements using a GA (section 3.5).
5. Apply weaving rules at the points of failure in the behavioral test suite to generate fail-safe test paths (section 3.6).
6. Generate, execute and validate tests (section 3.7).
7. Evaluate Fitness (section 3.8).
8. If necessary, generate more test requirements (section 3.9).

The proposed failure mitigation test process assumes that a technique for black box testing of required functionality exists via a behavioral model (BM), associated

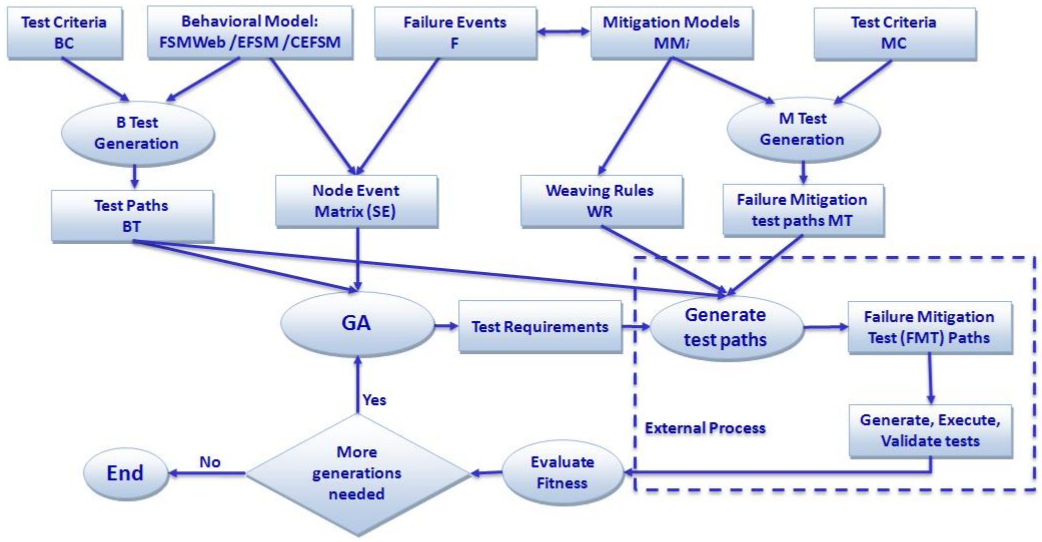


Figure 3.1: Test Generation Process

behavioral testing criteria (BC), and behavioral test paths (BT). In our case we will use an existing web application MBT approach, FSMWeb [9]. We assume that system requirements exist that identify types of failure events and any required mitigation actions, e.g via hazard and risk analysis [30]. This is used to build failure mitigation models (MM) for which mitigation coverage criteria (MC) can be identified and mitigation test paths (MT) can be created. Since all external failures are not possible in all behavioral states, a State Event Matrix (SE) determines which failure types are possible in which behavioral states. This matrix and the test paths BT are then used in a heuristic search (GA) to determine mitigation test requirements. The failure mitigation test paths (FMT) are then created based on selecting an appropriate mitigation test ($m \in MT$) and weaving it into the behavioral test according to the weaving rules associated with the selected mitigation test. These are then transformed into executable tests, executed, and validated. This is a process external to the GA and partly manual, hence orders of magnitude more expensive than generating test requirements. The result of validation is used to determine values of the fitness function and to determine a new generation of test requirements, or to determine that enough tests were generated (i.e. the GA terminates). The following sections describe each of these steps in more detail.

3.2 FSMWeb: the Behavioral Model

Functional testing for web application follows the approach in [9]:

- Build a hierarchical model *HFSM*:
  - Partition the web application into clusters (*C*s).
  - Define Logical Web Pages (*LWP*s) and Input-Action constraints for each.
  - Build FSMs for clusters as a multi-level hierarchy.
  - Build an Aggregate FSM (*AFSM*) to represent the top level of the application.
- Generate tests from the *HFSM*.
  - Generate paths through each FSM that meet the coverage criteria.
  - Aggregate paths to form abstract tests.
  - Choose inputs along the paths to create executable tests.

The term *cluster* is used to refer to collections of software modules/web pages that implement a logical, user level function. The first step partitions the web application into clusters. At the highest level of abstraction, clusters represent functions that can be identified by users. At a lower level, clusters represent cohesive software modules/web pages that work together to implement a portion of a user level function.

Many web pages contain HTML forms, each of which can be connected to a different back-end software module. To facilitate testing for these modules, web pages are modeled as multiple *Logical Web Pages* (LWPs). A LWP is either a physical web page or the portion of a web page that accepts data from the user through an HTML form and then sends the data to a specific software module. FSMWeb is a functional model meant for black-box testing. Hence the web application can be written in any language appropriate for web applications(e.g. HTML, JavaScript, .. etc.). LWPs are abstracted from the presentation defined by the HTML and are described in terms of their sets of *inputs* and *actions*. All inputs in a LWP are considered atomic: data entered into a text field is considered to be only one user input symbol, regardless of how many characters are entered into the field. There may be rules about the inputs. Some inputs may be required; others may be optional; users may be allowed to enter inputs in any order; or a specific order may be required. Table 3.1 shows the input constraints for both types while Table 3.2 shows how typical input types found in web applications are represented as constraints on (single) edges in an FSMWeb model.

Table 3.1: Constraints on inputs

Input Choice	**Order**
Required (**R**)	Sequence (**S**)
Required Value (**R**(parm))	Any (**A**)
Optional (**O**)	
Single Choice (**C1**)	
Multiple Choice (**Cn**)	

This edge annotation via input-action constraints is one of the saving sources for an FSMWeb model because options for input selection and sequencing no longer need to be coded explicitly (which would inflate a traditional state-based model). Figure 3.2 shows how an FSM model that represents a selection with only three choices is reduced to two nodes and one transition in FSMWeb.

The lowest level cluster FSMs are generated with only LWPs and navigation between them. Input-action constraints annotate each edge [9]. Higher level FSMs

Table 3.2: FSMWeb constraint of typical input types

Input Type	**FSMWeb Edge Annotation**
Text Field Text Area Field	R (input name)
Optional Text Field Optional Text Area Field Optional Checkbox	O (input name)
Radio Box Drop Down Box (with n options)	C1 (option 1, ..., option n)
Optional Radio Box (with n options)	O (C1 (option 1, ..., option n)
Set of Checkboxes Multi-Select Box (with n options requiring 0 to n selections)	O (Cn (option 1, ..., option n)) A (option 1, ..., option n)

represent FSMs from a lower level cluster by a single node and may contain LWP nodes as well. Ultimately, a top level Aggregate Finite State Machine (AFSM) is formed and represents a finite state model of the application at the highest level of abstraction.

Test sequences are generated during phase 2 of the FSMWeb method. A test sequence is a sequence of transitions through the application FSM and through each lower level FSM. FSMWeb's test generation method first generates paths through each FSM based on some graph coverage criterion such as *edge coverage*. These paths are then aggregated based on an aggregation criterion for each FSM's paths, such as *all combinations* or *each path at least once* [9].

This process results in a set of aggregate paths. We call them *abstract tests*. The final step of test generation is selecting inputs to replace the input constraints for the transitions of the aggregate paths.

Input selection uses a technique [61] that builds two databases: a *synthetic database*, which consists of values that are consumed during testing, and an *appli-*

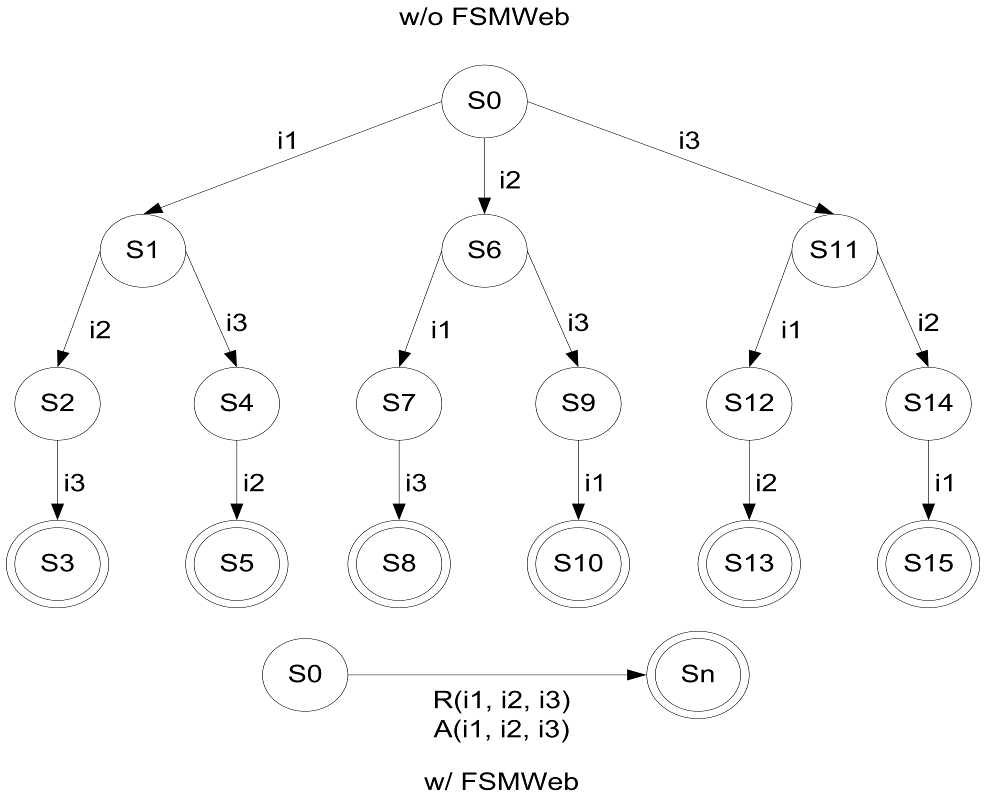


Figure 3.2: Three Optional Inputs, Any Order

cation database, which contains values previously inserted by the application being tested. Values are saved into the application database during execution and saved into the synthetic database during testing. Details about the database creation and input selection can be found elsewhere [61].

Hence, an $HFSM = \{FSM_i\}_{i=0}^{n}$ with a top level $FSM_0 = AFSM$. Each FSM has nodes that represent LWPs or clusters. Edges are internal or external to an FSM. External nodes span cluster boundaries. (They become internal at the next higher level.) External edges can either enter or leave a cluster FSM.

The FSM Tool parses HTML files and builds a FSMWeb model. The user can select coverage criteria such as node, edge, edge-pair, simple round trip and prime path coverage. The FSM Tool then generates test paths through each cluster that satisfy the selected criteria. For aggregation criteria, FSM Tool offers all-combinations, each choice and base choice coverage.

3.2.1 Example

Figure 3.3 shows three FSMs and two levels of hierarchy[1]. This is an example of a behavioral model (BM). Solid circles represent LWP nodes, the others are cluster nodes (i.e c_1 and c_2 in AFSM). It also shows $FSM1$ and $FSM2$ for c_1 and c_2 clusters. Table 3.3 shows paths through each FSM that achieve edge coverage. These test paths are aggregated to form abstract tests through the AFSM. As aggregation coverage criterion we use all combinations [9]. We illustrate this on $t_{01} = n_1c_2n_2$. Substituting t_{21} and t_{22} for c_2 results in two paths: $p_1 = n_1n_5n_7n_2$ and $p_2 = n_1n_5n_6n_7n_2$. Both paths consist of LWP nodes and do not have to be aggregated further.

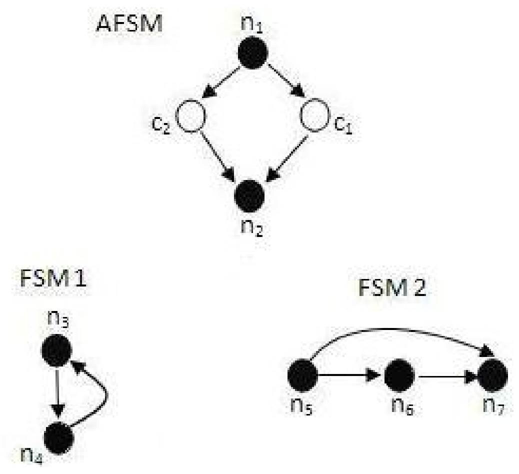


Figure 3.3: Behavioral Models BM

[1]For simplicity, we omitted input predicates

Table 3.3: Test paths through AFSM,FSM1,FSM2

FSM	Test paths
$AFSM$	$t_{01} = n_1c_2n_2$, $t_{02} = n_1c_1n_2$
FSM_1	$t_{11} = n_3n_4n_3$
FSM_2	$t_{21} = n_5n_7$, $t_{22} = n_5n_6n_7$

Aggregation of t_{02} which visits cluster node c_1 requires aggregation of one test path through FSM_1. This results in one path. There are 3 paths when test paths are fully aggregated. Table 3.4 shows these paths, including derivation rules used and test path lengths.

Table 3.4: Test Paths for BM

Test	Test Paths BT	Derivation Rules	Full test path	Length
$t_{01} : n_1c_2n_2$				
bt_1	$n_1\mathbf{t_{21}}n_2$	$c_2 \to t_{21}$	n_1, n_5, n_7, n_2	4
bt_2	$n_1\mathbf{t_{22}}n_2$	$c_2 \to t_{22}$	n_1, n_5, n_6, n_7, n_2	5
$t_{02} = n_1c_1n_2$				
bt_3	$n_1\mathbf{t_{11}}n_2$	$c_1 \to t_{11}$	n_1, n_3, n_4, n_3, n_2	5

3.3 Failures (F), State-Event matrix (SE)

External failures can occur in web application as a result of physical (network and system domain) failures, application failures, or client error (user generated/Interaction) [20, 32]. Examples of fault-type taxonomies related to web applications are described in [34, 10, 51]. Many systems, including safety critical systems and web applications have requirements for mitigating failures. This may include specified exception handling such as retry or exception-driven rework.

Examples of possible external failures are:

- f_1: Unavailability, e.g. no network connection.
- f_2: Time out, e.g. web session has no activity for a period of time.
- f_3: Parameter incompatibility, e.g. input data mismatch (e.g. integer vs. string).
- f_4: Response error, e.g. Database server is busy or does not save the data.
- f_5: Misunderstood behaviour, e.g. try to access web service that requires a different user type.
- f_6: Workflow inconsistency, e.g. client using back and forth browser navigation.
- f_7: Incorrect order, e.g. some service should have been completed before a specific step.
- f_8: Browser incompatibility.
- f_9: Interface change, e.g. data mapping of an external web service is changed.
- f_{10}: Incorrect service, e.g. wrong response from accessing an external web service .

We only need to test proper failure mitigation for those failure types that have mitigation requirements. Let $F = \{f_1, ..., f_k\}$ be the failure types, and $S = \{s_1, ..., s_n\}$ be the behavioral states. Failures may not be applicable in all behavioral states. For example, a failure type such as no network connection is applicable in all states, but a failure type such as expired session is not applicable in the entry portal state. Hence, we need to define which failure types can occur in which behavioral states.

We express this in a State-Event matrix SE where element se_{ij} is given by:

$$se_{ij} = \begin{cases} 1, & \text{if failure type j applies in node } i \text{ in } S \\ 0 & \text{Otherwise.} \end{cases}$$

Using our example in section 3.2.1, we assume 4 failure types are applied. SE is defined as shown in Table 3.5.

Table 3.5: State-Event Matrix SE

Behavioral States (N)/ Failure Type (f)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	**dpe**
1	1	0	0	1	1	0	1	**0.58**
2	0	0	1	0	1	1	0	**0.43**
3	0	1	0	1	1	1	1	**0.72**
4	0	1	0	0	0	0	1	**0.29**
dps	**0.25**	**0.50**	**0.25**	**0.50**	**0.75**	**0.50**	**0.75**	

3.4 Mitigation Requirements and Mitigation Models

We assume that the failure events and any mitigation actions are stated explicitly in the requirements. If they are not, two situation may occur:[2]

1. Mitigation requirements are implicit and the tester needs to take the extra step to make them explicit.

[2]Ammann and Offutt [3] mention these situations as not uncommon, hence causing issues for many testing techniques.

2. The requirements document is silent about whether mitigation is required. In this case, a tester may assume that mitigation is not required and proceed accordingly.

Mitigation requirements can be expressed in the form of mitigation models. For example, try other alternatives is shown in Figure 3.4.

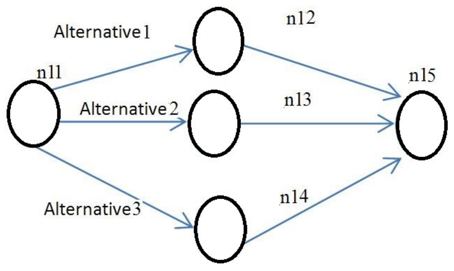


Figure 3.4: Try Other Alternatives: Mitigation Model.

Each failure f_j is associated with a corresponding mitigation model MM_j where j = 1,...,k. We assume that the models are of the same type as the behavioral model BM (e.g. an FSMWeb model). Graph-based [3], mitigation coverage criteria MC_j can be used to generate mitigation test paths $MT_j = mt_{j_1}, \ldots, mt_{j_{h_l}}$ for failure f_j. Assuming MC as "edge coverage" for the mitigation model in Figure 3.4, the following three mitigation test paths fulfill MC: MT=$\{mt_1$, mt_2, $mt_3\}$ where $mt_1 = \{n_{11}, n_{12}, n_{15}\}$, $mt_2 = \{n_{11}, n_{13}, n_{15}\}$, $mt_3 = \{n_{11}, n_{14}, n_{15}\}$.

Mitigation models can be very small for some failures and the mitigation can be an "empty action". For example, if there is a rollback to state s_b with immediate stop, the mitigation action only consists of adding a transition from s_b to s_f, the final state. The weaving rule would specify what node to rollback to, in this case s_b. On the other hand, some mitigation models may consist of a full set of alternative behaviors that completely replace the remainder of the original test. We will illustrate this in the section 3.6.

Using the example of failure types in section 3.3, the corresponding mitigation requirements are summarized in Table 3.6 with the corresponding mitigation models.

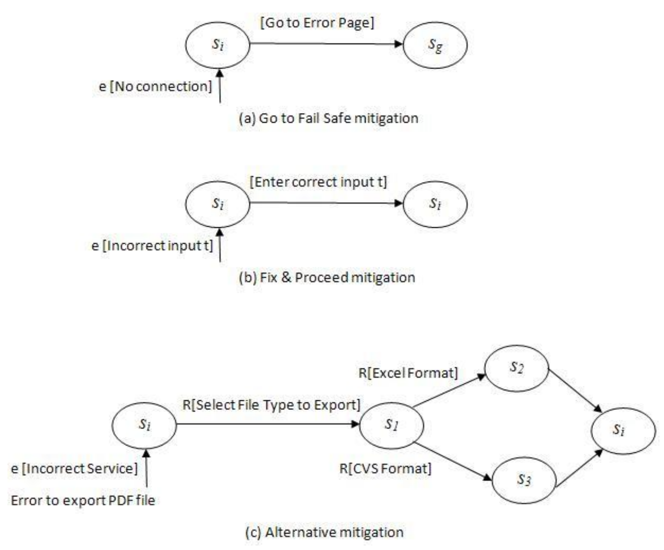


Figure 3.5: Mitigation Models.

Table 3.6: Mitigation Requirements

MM	Explanation	Model
MM1	Go to Fail Safe State: keep the system running even if there is no connectivity	see Figure 3.5-a, $MT_1 = \{mt_{11}\}$ *where* $mt_{11} = s_i, s_g$ and $s_g = LWP : errorpage$
MM2	End All: session expire, so start over from the start node	$MT_2 = \phi$ and $s_b = n_1$,where s_b is the start node
MM3	Fix & proceed: parameter incompatibility such as data mismatch	see Figure 3.5-b, $MT_3 = \{mt_{31}\}$ *where* $mt_{31} = s_i, s_i$
MM4	Alternative: incorrect service	see Figure 3.5-c , $MT_4 = \{mt_{41}, mt_{42}\}$ *where* $mt_{41} = s_i, s_1, s_2, s_{i+1}$ and $mt_{42} = s_i, s_1, s_3, s_{i+1}$

3.5 Determine Black Box Test Requirements

The goal in this step is to define what types of failures should be injected at which point in the behavioral test suite. Since large test suites and many failure types offer a sizeable number of combinations to chose from, it makes sense to use a genetic algorithm (GA). We want to emphasize here that we use the GA to search for test requirements, not for test cases. The test requirements state where in the execution of the behavioral test a failure $f \in F$ needs to be injected. These test requirements need to be transformed into test paths, executable test cases and then need to be executed and validated. The result of validation provides necessary information to compute the fitness of the test requirements. This makes fitness evaluation expensive compared to generating test requirements and hence we need to carefully consider this cost of fitness evaluation. The next subsections describe each step in more detail.

3.5.1 Representation of Population

The black box test requirements need to define:

- A test path where a failure event occurs
- A position in the test where the failure event is injected during test execution.

Since the test paths vary in length, a 3-dimensional population is not ideal. However, the length of the Behavioral test paths *(BT)* is fixed. The behavioral test path suite *(BT)* is a set of test paths:

$$BT = \{bt_1, bt_2, ..., bt_l\} \text{ where } l \textit{ is number of test paths.}$$

We arrange the behavioral test path suite *(BT)* into a single dimension (I) by concatenating the test paths: $I = (bt_1 \circ bt_2 \circ ... \circ bt_l)$

Each test path is a sequence of nodes. Thus,

$I = (s_{11}, ..., s_{1n_1}, s_{21}, ..., s_{2n_2}, ..., s_{l1}, ..., s_{ln_l})$

$Length(I) = \sum_{i=1}^{l} len(t_i)$ where $1 \leq i \leq l$

Hence, we encode the test path suite rather than individual test paths. Now, the position in the test path suite identifies both test and position in a particular test. This leads to a two-dimensional representation. The first dimension of the search space is the number of possible positions p in I ($1 \leq p \leq Length(I)$). The second dimension is the possible number of failure types where the types of failures are given by:

$F = \{f_1, f_2, ..., f_k\}$ and $1 \leq e \leq |F|$ are the possible values for dimension 2.

Individuals in the population are then defined as a pair of one position p in the test path suite and one failure type e: (p, e).

The search space is the cartesian product of all possible positions p in the test suite and failure types e. The search space is: (p, e) where $1 \leq p \leq Length(I)$ and $1 \leq e \leq |F|$.

Feasible Region: Not all combinations of (p, e) are feasible, since it is possible that some failures are not applicable in some behavioral states. To account for this, we use the SE matrix defined in section 3.3 to help define the feasible region. Let $node(p)$ be the index of the behavioral state at position p, then the Feasible Region is defined as:

$$\{(p, e) | 1 \leq p \leq Length(I), 1 \leq e \leq |F|, se_{(node(p),e)} = 1\}$$

Using the example in Figure 7.2 and Table 3.4 in section 3.2.1, the concatenation of behavioral tests BT results in $I = bt_1 \circ bt_2 \circ bt_3$, and $length(I) = 4 + 5 + 5 = 14$

where $I = bt_1 \circ bt_2 \circ bt_3 = n_1, n_5, n_7, n_2, n_1, n_5, n_6, n_7, n_2, n_1, n_3, n_4, n_3, n_2$.
The index of the node in position $p = 2$ is $node(2) = 5$.

Given the 4 failure types, the search space size is: $SP = 14 * 4 = 56$ (see Table 3.7). There are 27 feasible pairs.

Table 3.7: Search Space SP

I	bt_1				bt_2					bt_3				
Position (p)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F/N	n_1	n_5	n_7	n_2	n_1	n_5	n_6	n_7	n_2	n_1	n_3	n_4	n_3	n_2
f_1	1	1	1	0	1	1	0	1	0	1	0	1	0	0
f_2	0	1	0	0	0	1	1	0	0	0	1	0	1	1
f_3	0	1	1	1	1	1	1	1	1	0	0	1	0	1
f_4	0	0	0	1	0	0	0	1	1	0	0	0	0	1

3.5.2 Initial population selection

We use defect potential to create the initial population. Higher defect potential is assumed to carry a higher probability of finding a defect. We use the defect potential of an individual (p, e) for selecting the initial population. It is a measure of the likelihood of a mitigation defect to occur. A mitigation defect is more likely to be found if the failure that triggers the defective mitigation occurs. Hence, the probability of a failure occurring contributes to the defect potential. Likewise, the more types of failures can occur in a state, the higher its defect potential. We compute defect potential as the percentage of possible states and failures types that can occur respectively. For state $s_i \in S$, the defect potential is:

$$dps(i) = \frac{1}{|F|} \sum_{j=1}^{|F|} se_{ij}$$

where $|F|$ is the number of failure types and se_{ij} is an element in the state-failure event matrix SE. Similarly, for failures f_j of type j the defect potential is:

$$dpe(j) = \frac{1}{|S|} \sum_{i=1}^{|S|} se_{ij}$$

where $|S|$ is the number of behavioral states and se_{ij} is an element in the state-failure event matrix SE. In section 3.3, Table 3.5 represents an example of the SE and values for dps and dpe.

Algorithm 3.6 specifies how to select the initial population creates individuals (p, e) that give behavioral state coverage and failure type coverage. It begins by determining for each failure type j the behavioral state s_i with the highest $dps(i)$. It then determines the earliest position p where this state occurs and adds the pair (p, j) to the initial population. It removes s_i from further consideration. After all failure types have been matched with a position, if there are still states left whose index i' has not been chosen, we determine the failure type j' with the highest $dpe(j')$, and the earliest position p' where the state occurs and add (p', j') to the population until all states are covered.

Back to our example, $I = (n_1, n_5, n_7, n_2, n_1, n_5, n_6, n_7, n_2, n_1, n_3, n_4, n_3, n_2)$, $1 \leq p \leq 14$. Note that indexing is required, since state n_1 occurs three times (in $p = 1$, $p = 5$ and $p = 10$). The example from Table 3.5 has 4 failure types, hence $1 \leq e \leq 4$.

By applying **(Algorithm 2)** to this example, the initial population is selected as follows:

1. Determine each failure type with Max $dps(s)$ and select the first position p in I where s occurs: (2,1); (7,2); (8,3); (4,4);
2. For the remaining states determine state with Max $dpe(e)$ and select first position p in I: (1,1); (11,2); (12,3);

Algorithm 2: Algorithm for selecting initial population

```
Require: $se_{ij}$ matrix, $dps$, $dpe$, $I$
Ensure: Compute Initial Population (p, j) for GA
  Cover all failures, all states
  Pop=$\phi$
  Set S of states in BM
  for j=1 to $|F|$ do
    Determine state s with $max_{i=1}^{|S|}(dps(i))$ for failure type j and $SE(s,j) = 1$
    p ← Select first such position p in I
    S=$S \setminus \{s\}$ /* remove s from S */
    Pop=$Pop \cup \{(p,j)\}$
  end for
  while $S \neq \phi$ do
    Select $s \in S$
    Determine failure type j with $max_{j=1}^{|F|}(dpe(j))$ for $s$
    p ← Select first position p with s in I
    S=$S \setminus \{s\}$ /* remove s from S */
    Pop=$Pop \cup \{(p,j)\}$
  end while
```

Figure 3.6: Algorithm for selecting initial population

To ascertain whether using defect potential as defined here actually is reasonable (rather than multiple runs with a random initial population), we performed a series of experiments where we compared both. These are reported in section 4.2.

3.6 Generating the Failure Mitigation Test Paths (FMT)

Now that we have the initial test requirements, (i.e the position -failure type pairs (p,e)), we know where (p) in the test suite to inject which type of failure (e). We use mitigation test paths MT_e for the failure and weave them into the behavioral test at position (p) subject to weaving rules. The weaving rules are based on the type of mitigation. F is a set of failure types: $F = \{f_1, f_2, .., f_k\}$. Each failure is linked with a mitigation model such that:

$$f_j :: (M_j, MT_j, wr_j)$$

where f_j is a failure of type j

MM_j is a mitigation model for it

MT_j is a test suite for M_j

wr_j is a weaving rule based on mitigation M_j

Assume we have $t \in BT$, $p \in I$, $f_e \in F$ and $mt \in MT_e$. We now build a failure mitigation test path $fmt \in FMT$ using this information and the weaving rules wr_e $\in WR$ as follows:

- keep path represented by t until failure position p.
- apply failure of type e (f_e) in p.
- select appropriate $mt \in MT_e$. For example, if the aggregation criteria specifies that all $mt \in MT_e$ need to be covered, we need to select each and create a mitigation test for each.
- apply weaving rule wr_e to construct fmt.

We now explain weaving rules more formally for each type of mitigation. Let $t = \{s_1 \ldots s_b \ldots node(p) \ldots s_f \ldots s_k\}$. let s_g be a fail safe state.

1. Fix

Option 1: Compensate ((Partial) Fix and proceed) mitigates a failure and continues with the remainder of the behavioral test. So, $fmt = s_1 \ldots node(p)\ mt\ node(p) \ldots s_k$. mt may be zero, if mitigation does not require user involvement (inputs). See rule 4.

Option 2: Go to fail-safe state (Fix and stop) mitigates a failure and ignores the remainder of t: $fmt = s_1 \ldots node(p)\ mt\ s_g$. mt may be empty if there is no fix.

2. Rollforward

Option 1: Rollforward mitigates the failure, and proceeds.

$fmt = s_1 \ldots node(p)\ mt\ s_f\ \ldots s_k$ where s_f is the node in t to which we rollforward. If only rollforward and no other actions are required mt is empty and $fmt = s_1 \ldots node(p) s_f\ \ldots s_k$.

Option 2: Deferred fixing. If the failure can only be fixed after reaching the rollforward node s_f then smt becomes: $fmt = s_1 \ldots node(p)\ s_f\ mt\ s_{f+1} \ldots s_k$.

Note that further variants of this weaving rule can exist, like a state s_{df} between s_f and s_k at which the failure mitigation mt is inserted. $t = s_1 \ldots s_b \ldots node(p) \ldots s_f \ldots s_{df} \ldots s_k$. $fmt = s_1 \ldots node(p) s_f \ldots s_{df}\ mt \ldots s_k$.

3. Rollback

Option 1: Rollbackward. Apply mitigation path mt from point of failure and rollback to node where failure occurred and continue with remainder of behavioral test. $fmt = s_1 \ldots node(p)\ mt\ s_b \ldots s_k$ where s_b is a node before node (p).

Option 2: Rollbackward and stop.

$fmt = s_1 \ldots node(p)\ mt\ s_b$.

Option 3: Retry once. $fmt = s_1 \ldots node(p)\ node(p)^r \ldots s_k$ where r=1.

4. Internal compensate (no user action required)

Test immediate system fix. For example, this can happen if a system switches automatically to different backup web server. To test this merely requires applying the failure and continuing to execute the original test t. In this case, we do not have to modify the original test at all (note that the assumption is that the system deals with the failure internally without any change in black-box behavior).

We analyze the fault taxonomy for web applications in [34, 59] for possible mitigations. As a result, it is found there are eight themes or patterns of mitigations (see Figure 3.7). Some of them show similarity to [49, 46, 15].

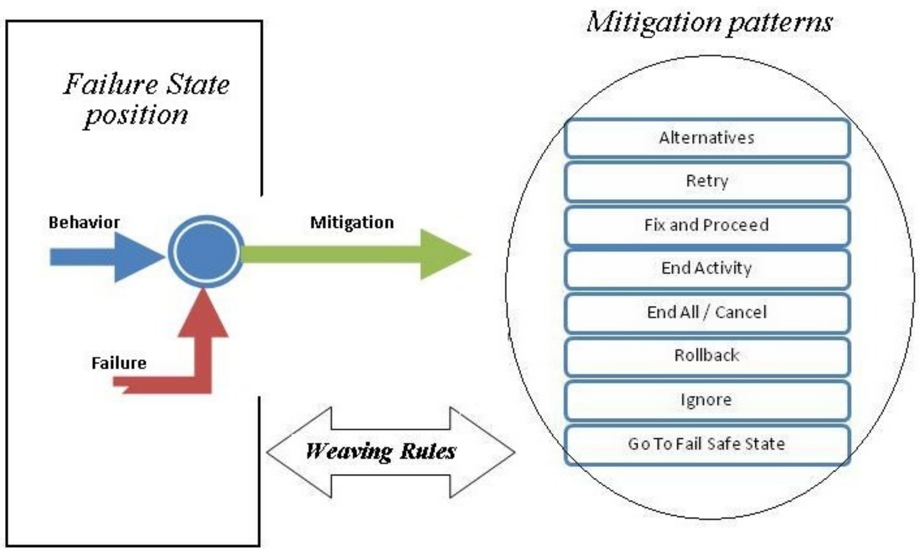


Figure 3.7: Mitigation Patterns.

While weaving rules in this section are representative, they are not meant to be comprehensive. We expect that, over time, we may find some more or find that some are more common than others. Table 3.8 summarize all weaving rules.

Let $PE = \{(p, e)|$ pair selected by $GA\}$. Weaving $mt_i \in MT_i$ into a test at position p (pair(p,i)) results in a failure mitigation test path *fmt*. However, in case of multiple mitigation test paths (i.e $|MT_i| > 1$), we need to decide what weaving criterion for MT_i to use. We have two options:

- All combinations, that is we weave each $mt_i \in MT_i$ for every occurrence of pair (k,i) , $1 \leq k \leq length(I)$, $(k, i) \in PE$.
- We merely cover all $mt_i \in MT_i$, but not at every possible position $(k, i) \in PE$.

The first option results in at least $|MT_e|$ fail-safe mitigation test cases for each (p,e) pair, hence is more costly. If there are x such (p,e) pairs in PE the number of $fmts$ is $|MT_e| \times x$. The second criterion results in $max\{|MT_e|, x\}$ $fmts$.

Using the example in section 3.2.1, Table 3.9 shows the selected pairs and the $fmts$ created based on them. The first column in Table 3.9 numbers each failure mitigation test ($fmt_1 - fmt_9$). The second column lists each (p, e) pair in PE. The third column refers to the failure type whose mitigation is tested. The fourth column states the node at position p. The fifth column identifies the behavioral test used in constructing fmt_i ($i = 1, \cdots, 9$). The sixth column identifies which mitigation model is used as described in Table 3.6. The seventh column lists which mt_{ij} is used as described in Table 3.6. The last column shows the failure mitigation tests.

3.7 Generate Executable Tests, Execute and Validate

The set of failure mitigation test paths now have to be transformed into executable tests. For the FSMWeb model, this means resolving the input predicates

Table 3.8: Mitigation Patterns and Weaving Rules

Mitigation pattern	**Weaving Rule Name**	**WR#**
Alternative	Fix - option 1	1
$FMT = s_1 \ldots node(p)\ mt\ node(p) \ldots s_k$		
Retry	Rollback - option 3	2
$FMT = s_1 \ldots node(p)\ node(p)^r \ldots s_k$		
Fix and Proceed	Fix - option 1	3
$FMT = s_1 \ldots node(p)\ mt\ node(p) \ldots s_k$		
End Activity	Rollforward - option 1	4
$FMT = s_1 \ldots node(p)\ mt\ s_f \ldots s_k$		
End All	Rollback - option 2	5
$FMT = s_1 \ldots node(p)\ mt\ s_b$		
Rollback	Rollback - option 1	6
$FMT = s_1 \ldots node(p)\ mt\ s_b \ldots s_k$		
Ignore	No user action required	7
Internal compensate		
Go to fail-safe	Fix - option 2	8
$FMT = s_1 \ldots node(p)\ mt\ s_g$		

(*cf.* section 3.2). The sequences of inputs form the executable tests. These are now executed. The failure is injected at the proper position in the test execution. Execution monitoring is used to reveal mitigation defects. We record any failure mitigation defects found for each test requirement (p, e). This information is used to determine the value of the fitness function.

Table 3.9: Selected (p, e) Pairs and resulting FMT.

#	pairs	Failure	Node	BT used	MM used	mt_{ij} used	FMT
1	(2,1)	f_1	n_5	bt_1	MM1	mt_{11}	n_1, n_5, s_g
2	(11,2)	f_2	n_3	bt_3	MM2	mt_{21}	n_1, n_3, n_1
3	(7,3)	f_3	n_6	bt_2	MM3	mt_{31}	$n_1, n_5, n_6, n_6, n_7, n_2$
4	(8,4)	f_4	n_7	bt_2	MM4	mt_{41}	$n_1, n_5, n_6, n_7, s_1, s_2, n_7, n_2$
5	(8,4)	f_4	n_7	bt_2	MM4	mt_{42}	$n_1, n_5, n_6, n_7, s_1, s_3, n_2$
6	(12,3)	f_3	n_4	bt_3	MM3	mt_{31}	$n_1, n_3, n_4, n_4, n_3, n_2$
7	(8,1)	f_1	n_7	bt_2	MM1	mt_{11}	n_1, n_5, n_6, n_7, s_g
8	(6,2)	f_2	n_5	bt_2	MM2	mt_{21}	n_1, n_5, n_1
9	(4,3)	f_3	n_2	bt_1	MM3	mt_{31}	n_1, n_5, n_7, n_2, n_2

3.8 Fitness evaluation and determination of next generation

3.8.1 Fossil Record (FR):

For each test requirement and its associated test case(s) test validation results indicate whether it helped find a failure mitigation defect or not. For each generation, we record this information in the fossil record. A fossil record entry is a triplet (p, e, d) where (p, e) is a test requirement from a given generation and d is a boolean value that is 1 if a mitigation defect was found through the test requirement and 0 if it was not.

The fossil record stores the old population of test requirements for further comparison with new population data. The fitness function uses this information when it tries to determine either very novel test requirements (compared to those from prior generations) or mine around test requirements that helped find mitigation defects (i.e. generate new test requirements that are similar). This makes the fitness function relative, rather than absolute.

3.8.2 Fitness Function:

Similar to James et al. [54], it provides for exploring, prospecting and mining. Exploring rewards novelty as being farther away from individuals in the fossil record, while prospecting and mining use functions that either measure the distance from known mitigation defects (farther is better) or search closely around existing defects. First, the fitness function is using the novelty (S) of an individual (p, e). It is computed as the distance from each new individual to each entry in the current fossil record:

$$S(p_c, e_c) = \sum_{q=0}^{|FR|-1} \sqrt{\frac{(p_c-p_q)^2}{length(I)} + \frac{(e_c-e_q)^2}{|F|}}$$

where p_c is a position (point of failure) to be evaluated

e_c is a failure type at the point of failure to be evaluated

p_q is the position of the fossil record

e_q is the failure type of the fossil record

$length(I)$ is the total number of positions

$|F|$ is total number of failure types

$FR = \{(p_q, e_q, d_q)|(p_q, e_q, d_q)\ in\ Fossil\ Record\}$

$|FR|$ is the total number of (p, e) pairs in FR

The second part of the fitness function uses proximity (R) of individuals to individuals with known mitigation defects. It helps the search to focus attention on error-weighted regions of the search space. Based on the individuals in the fossil record that triggered defects, R is calculated as the distance for the new individual from all individuals in the fossil record that triggered a defect.

$$R(p_c, e_c) = \sum_{(p,e,1)\in FR_d} \sqrt{\frac{(p_c-p)^2}{length(I)} + \frac{(e_c-e)^2}{|F|}}$$

where p, and e are the position and failure type in the fossil record that triggered a mitigation defect. $FR_d = \{(p, e, 1)|(p, e, 1) \in FR \textit{ and (p,e) found mitigation defect}\}$ After executing the test cases associated with a given individual (p, e), R is used as follows, depending on whether (p, e) triggered a mitigation defect:

$$R(p_c, e_c) = \begin{cases} R(p_c, e_c) & \text{if } (p_c, e_c) \text{ found a mitigation defect;} \\ \frac{1}{R(p_c, e_c)} & \text{Otherwise.} \end{cases}$$

That is, if a mitigation defect was found, we prospect away from the known defects, if (p, e) did not trigger a mitigation defect, we mine around individuals that are known to trigger defects. The overall fitness function tries to balance exploration, prospecting and mining as in James et al. [54] as follows:

$$Fitness = (w_s \times S^{1.5})^2 \times (w_r \times R^{1.5})^2$$

where w_s is a weight for exploration and w_r is a weight for prospecting and mining.

3.8.3 Generate New Generation

This step consists of 3 parts. First, the GA ranks the individuals in the current population by their fitness values and selects the top half (i.e. we use the median of the fitness values as a cut off). These are candidate pairs for the new generation.

Second the GA selects a proportion of these most fit individuals for crossover, based on a crossover rate (CR). We chose a CR=0.5. That means the GA selects 50% of the most fit individuals and exchanges their positions. Crossover between two individuals (p_1, e_1) and (p_2, e_2), is accomplished by exchanging the positions, thus creating new individuals (p_1, e_2) and (p_2, e_1).

The third step is to mutate a certain percentage of the current population. This percentage is called mutation rate (MR). For example, if the mutation rate is 30% to mutate (p, e) into the (p', e), the GA randomly selects 30% of the individuals and randomly selects a $p' \neq p$ as the new position for them. The algorithm removes duplicates and individuals that are infeasible based on the SE matrix.

Note that the population sizes of successive generations are dynamic similar to [57, 12].

3.9 Stopping Criteria

The GA will continue as long as new generations are created, or the number of test requirements has not reached its limit (i.e we have a limited test budget). If the test budget has not been exhausted, we attempt to create a new generation, subject to the following condition: there is at least one individual in the current generation with a higher fitness function than individuals in the fossil record.

If the mutation and crossover do not produce new individuals (i.e. because the test requirements are not feasible according to the SE matrix, or are duplicates of individuals in the Fossil Record) the algorithm also stops.

Algorithm 3.8 shows the pseudocode of the approach and algorithm 3.9 shows the pseudocode of creating a new generation.

.

Algorithm 3: GA Approach

```
Require: State-Event Matrix ($SE$), Novelty Weight $w_s$, Proximity Weight $w_r$, Crossover rate ($CR$), Mutation Rate ($MR$), Failure types $F$, Concatenate of behavioral test ($I$)
Ensure: Selected Best Pairs $(p, e)$
1: Generation of pairs $Gen = \phi$ /* $Gen$ is an Array of $(p, e, d, FF)$ where d is defect found and FF fitness function Initial $\forall(p, e) : d = false$ and $FF = null$ */
2: Fossil Record $FR = \phi$
3: $NoGeneration = 0$
4: while More Generation Needed do
5:   if $NoGeneration = 0$ then
6:     /* compute initial population based on $dpe, dps$ */
7:     $Gen \leftarrow ComputeInitialPopulation(SE)$ /* see Algorithm No. 4 */
8:   else
9:     /* Generate New Generation */
10:    /* use crossover rate, mutation rate, old $Gen$ and search space to generate new $Gen$ */
11:    /* if i>1, $Gen$ is the best individual selected on line 46 */
12:    $Gen \leftarrow GenerateNewPopulation(Gen, CR, MR, I)$ /* see Algorithm No. 3 */
13:  end if
14:  $NoGeneration = NoGeneration + 1$
15:  $Results = \phi$
       /* External Process */
       $\forall(p, e) \in Gen : d = false$
       Loop $\forall(p, e) \in Gen$
         Generate Tests $(p, e)$
         Execute Tests $(p, e)$
         Evaluate Tests $(p, e)$
         /* Record Defect (p,e,d) where d is defect found */
         IF defect found using (p,e) Then
           $d = true$
           $Results \leftarrow (p, e, d)$ /* Enter $(p, e, d)$ into $Results$ */
16:    End Loop
17:  if $NoGeneration > 1$ then
18:    /* Compute Fitness Function */
19:    for $c = 0$ to $length(Gen)$ do
20:      $S = 0 : R = 0$
21:      /* Compute Novelty S */
22:      $Sum = 0$
23:      for $l = 0$ to $length(FR)$ do
24:        $Sum \leftarrow Sum + ([(Gen[c].p - FR[l].p)^2/length(I)] + [(Gen[c].e - FR[l].e)^2/length(F)])$
25:      end for
26:      $S \leftarrow SQR(Sum)$
27:      /* Compute Proximity R */
28:      $Sum = 0$
29:      for $d = 0$ to $DefectFound$ do
30:        $Sum \leftarrow Sum + ([(Gen[c].p - FR[IndexDefect[d]].p)^2/length(I)] + [(Gen[c].e - FR[IndexDefect[d]].e)^2/length(F)])$
31:      end for
32:      if $Gen[c].d$ is true then
33:        $R \leftarrow SQR(Sum)$
34:      else
35:        $R \leftarrow 1/SQR(Sum)$
36:      end if
37:      Fitness value for individual
38:      $Gen[c].FF = (w_s \times S^{1.5})^2 \times (w_r \times R^{1.5})^2$
39:    end for
40:    /* Select individuals with highest $FF$ for crossover and mutation */
41:    /* These individuals will be used for crossover and mutation */
42:    $Gen \leftarrow \{Gen|Gen.FF \geq Median(Gen.FF)\}$
43:  end if
44:  /* Store best Selection in $FR$ */
45:  $FR \leftarrow Gen$
46:  $DefectFound = 0$
47:  for $f_1 = 0$ to $length(FR)$ do
48:    for $f_2 = 0$ to $length(Results)$ do
49:      /* defect found */
50:      if $FR[f_1] = Results[f_2]$ and $Results[f_2].d$ is true then
51:        /* Keep defect index position to be used in Fitness Function */
52:        $IndexDefect[DefectFound] \leftarrow f_1$
53:        $DefectFound \leftarrow DefectFound + 1$
54:      end if
55:    end for
56:  end for
57: end while
58: Output: $FR$ includes Best Pairs
```

Figure 3.8: GA Approach

Algorithm 4: Algorithm for creating New Generation

```
Require: Generation of pairs ($Gen$), Crossover rate ($CR$), Mutation Rate ($MR$),
  Length of concatenate a behavioral test ($I$)
Ensure: Generate New Population for GA
  /* Crossover */
  $Split \leftarrow Integer(length(Gen) \times CR)$ /* Split index position in $Gen$ */
  /* Swap positions */
  $p_1[] \leftarrow \phi$
  $p_2[] \leftarrow \phi$
  /* Split the position vector from first position to Split index position in $Gen$ */
  $p_1 \leftarrow Slice(Gen.p, 0, Split)$
  /* Split the position vector from Split index position to the end in $Gen$ */
  $p_2 \leftarrow Slice(Gen.p, Split, length(Gen))$
  /* Merge two position vectors $p_1$ and $p_2$ */
  $Gen.p \leftarrow Merge(p_1, p_2)$
  /* Mutation */
  Number of Mutation positions $MutP \leftarrow Integer(length(Gen) \times MR)$
  for $pm = 0$ to $MutP$ do
    $SelectedPair \leftarrow Random(Gen)$ /* Random Selection of individual from $Gen$ /
    Selected position $p' \leftarrow Random(length(I))$ /* Random Selection of position
    from $I$ /
    /* Replace position $p$ in selected individual with $p'$ */
    $Gen[SelectedPair] \leftarrow \{(p', e) | p' \in I \wedge p' \neq Gen[SelectedPair].p, 1 \leq p' \leq length(I)\}$
  end for
```

Figure 3.9: Algorithm for creating New Generation

Chapter 4

Validation of GA approach

There are several parts of this approach that need to be analyzed and evaluated. The most important is the GA approach, since it is at the core of the proposed test approach. In this section we address the following questions:

1. What mitigation defect density rates occur in practice and what are good weights w_r and w_s to use for them?
2. For commonly occurring mitigation defect densities, does the GA algorithm generate individuals which when converted to fail-safe mitigation tests trigger mitigation defects?
3. Is the approach scalable?
4. How do the results compare with random generation of individuals both in terms of effectiveness and efficiency.

4.1 Description of Simulator

We built a simulator that allows to vary the following variables:

1. Test suite size (*length(I)*)

2. Failure types (|F|)
3. Mitigation defect density (D)
4. Applicability level (AL)(percentage of ”1” entries in the state-event matrix)
5. Duplication factor (DF). This variable is defined as $DF = length(I)/|S|$. that is, the length of the concatenated test suite divided by the number of states in the behavioral model. It computes the average number of times a state occurs in I.
6. Crossover rate (CR)
7. Mutation rate (MR)
8. Exploration weight (w_s)
9. Prospecting and mining weight (w_r)
10. Number of runs per problem (NR)
11. Type of (p,e) pair (test requirement) generation. The simulator supports the following approaches: GA, Random, and CC (Coverage Criteria).

The first two variables describe the problem size and characteristics. *length(I)* and |F| determine the size of the search space. The mitigation defect density is used to determine how many mitigations are defective. These are marked with (d), so it can be determined later whether a selected (p,e) pair uncovers a mitigation defect or not. The applicability level is used to determine the number of ’1’ in the state event matrix which is generated next. Using the duplication factor, the simulator determines the number of states in the behavioral model and generates a concatenated test suite as a sequence of nodes.

The remainder of the variables are used to configure the GA and to determine the number of runs for each problem.

The simulator selects one or more of the (p,e) pair generation approaches (GA, Random) and determines the set of test requirements ((p,e) pairs). Then it determines whether or not they found a defect and computes defect coverage for each approach. We use the following dependent variables:

- Test requirements ((p,e) pairs)
- The number of test requirements.
- The set of mitigation defects found
- The number of mitigation defects found
- The percentage of mitigation defects found.

The simulator computes the (p,e) pairs needed for testing, determines how many mitigation defects were found and the proportion of mitigation defects found. It also selects the same number of (p,e) pairs randomly and determines the number of mitigation defects found through random selection so we can compare GA performance a against random selection.

We use the simulator to tune the weight of the fitness function, compare our (deterministic) initial population to random selection, and compare GA against random selection of test requirements (section 4.4). Finally, since the cost of fitness evaluation on an actual case study is expensive and caused us to abandon the traditional GA approach of multiple runs, we compared single run performance against multiple runs.

After we set up the simulator's parameters values, the simulator will automatically build the state-event matrix (SE) based on applicability level percentage value (AL). The value of "1" entries in the state-event matrix will be distributed randomly. Next, the simulator constructs a defect state event matrix based on the mitigation defect density (D) by multiplying the number of '1' in the SE matrix with the defect density D and rounding to the next integer. The result is the number of defective mitigations. For example, if the SE matrix contains 100 of '1' entries and D=5%, there are 5 mitigation defects. These are randomly selected and marked in the defect state event matrix. Whenever a (p,e) pair's fitness is evaluated, the simulator looks up the node in position p and checks whether the defect state event matrix for *(node(p),e)* is marked as having a defect.

4.2 Initial Population

Since we determine the initial population deterministically, rather than randomly using multiple runs, we need to evaluate how good the initial population is. Table 4.1 shows the results comparing initial population selected via defect potential (Algorithm **??**) against multiple runs of randomly selected initial populations (10 runs). The first column lists the number of generations generated. The second and third columns list results for using defect potential for selecting the initial population (pairs needed in each generation and cumulative percentage of defects found, respectively). Columns 4 and 5 list the results for selecting the initial population randomly.

The simulator's parameters values are selected based on our case study (section 6) as follows:

1. Test suite size (length(I))=169
2. Failure types (|F|)=10
3. Mitigation defect density (D)=5%
4. Applicability level (AL)=42%

5. Crossover rate (CR)=0.5

6. Mutation rate (MR)=0.3

7. Exploration weight (w_s)=3

8. Prospecting and mining weight (w_r)=1

9. Number of runs=10

The results show that using the defect potential is more efficient. Using defect potential finds all mitigation defects in fewer generations with fewer (p, e) pairs (last row of Table 4.1). The results show that random selection of initial population needs 5 additional generations and 182 additional test requirements compared to using defect potential. In addition using defect potential finds earlier (first defect found in generation 1 vs. 3).

Table 4.1: Random vs. defect potential

				(based on 10 runs)
# of Generation	Initial population based on defect potential		Random Initial Population	
	# of pairs	Defect Found %	# of pairs	Defect Found %
intial popluation	16	33%	12	0%
2	20	33%	15	0%
3	21	33%	18	33%
4	23	33%	21	33%
5	26	33%	23	33%
6	29	33%	25	33%
7	30	33%	28	33%
8	33	67%	32	33%
9	36	67%	34	33%
10	37	67%	36	33%
11	40	67%	37	33%
12	41	100%	38	33%
13			40	67%
14			41	67%
15			43	67%
16			44	67%
17			47	100%
Total pairs	352		534	

4.3 Tuning of the GA

An important question in making the GA approach described in section 3 was what values to use for exploration weight and prospecting weight. Basically higher defect densities profit from a higher weight for prospecting and mining ($w_r > w_s$) while lower defect densities show GA results when the exploration weight is higher ($w_s > w_r$).

To determine what are "high" and "low" defect rates, we rely on Sawaelpong *et al.* [67] who report that 19-23% of exception handling routines have defects. They consider 20% a high defect rate and 5% a low one. In their tuning experiments, they vary w_s and w_r for search spaces varying from 800-2000 and found that for high defect rates $w_r = 3$ and $w_s = 1$ had the highest defect coverage while for low defect rates $w_r = 1$ and $w_s = 3$ performed better. When defect rates are in between (neither high, nor low) unequal weights do not fare as well. This is summarized in Table 4.2. For the high defect rate of 20% biasing towards prospecting and mining uncovers between 88-92% of the defects, while for the lower defect rate of 15%, only 78% of the defects are found. Similarly, biasing towards exploration for the low defect density of 5% finds between 95%-100% of the mitigation defects while this biasing does not work as well for the higher defect rate of 10%: only between 73-85% of the mitigation defects are found. It is thus important to consider expected mitigation defect rate when selecting these weights in the fitness function.

We next turn to the selection of values for crossover and mutation rate. We used a crossover rate of CR=0.50 and a mutation rate of MR=0.30. Typical crossover rates based on De Jong's simulations are between 0.5-0.6 [16, 22]. As for the mutation rate, theoretical work reports a rule of thumb of $1/N$ (N is the number of genes, in our case $N = 2$) [11]. By contrast typical mutation rates in the literature are

Table 4.2: Percentage of Mitigation Defects Found

	w_r=**3** w_s=**1**		w_r=**1** w_s=**3**	
length(I) $\times\|F\|$	**20%**	**15%**	**10%**	**5%**
800	92	78	80	95
1000	88	78	85	100
2000	92	78	73	95

around 0.15 [16]. The mutation rate we chose is a compromise between the two. As is common practice in GA experiments [2], we used 10 simulation runs per problem.

4.4 Comparsion GA vs. Random

We performed a series of simulation experiments to compare the performance of the GA with random selection of test requirements ((p, e) pairs). Table 4.3 shows the values of the parameters chosen. $length(I)$ is the length of the test suite, $|F|$ is the number of failure types. $length(I) \times |F|$ is the size of the potential search space (the GA removes infeasible pairs based on the SE matrix), D is the mitigation defect density. This was chosen to be small, since it makes for a more difficult search problem. The GA parameters were chosen as explained in section 4.3.

The results are shown in Table 4.4. The leftmost column shows $length(I) \times |F|$. The second column reports the number of (p, e) pairs needed to find the defect percentage reported in column three for the GA. The last column reports percentage of mitigation defects found by random search using the same number of (p, e) pairs. The GA finds all mitigation defects while random search never finds all defects and is unable to find any defects for 6 of the 10 problems.

The next question was whether random generation was just less efficient and would have eventually caught up with the GA approach if allowed to generate more

Table 4.3: Parameter settings for GA vs Random Comparison

length(I)	F	*length(I)* $\times\lvert F\rvert$	D	AL	CR	MR	w_s	w_r	NR
40-400	5	200-2000	5%	80%	0.5	0.3	3	1	10

Table 4.4: Effectiveness: GA vs. Random

$length(I) \times \lvert F\rvert$	**# of pairs (p,e)**	**GA**	**Random**
		% Defect Found	**% Defect Found**
200	31	100	50
400	34	100	50
600	39	100	25
800	34	100	0
1000	38	100	0
1200	38	100	0
1400	39	100	0
1600	32	100	0
1800	32	100	0
2000	35	100	33

individuals. Table 4.5 explores this. It lists the number of individuals needed to reach the first defect, to reach 50% of the defects, and to reach 100% of the defects for both GA and Random based generation. As before we varied population between 200 and 2000. If random generation did not reach a detection level, the table reports n/a. For smaller search spaces (populations 200, 400, 600) Random generation initially finds mitigation defects faster, but, unlike the GA approach it plateaus and is unable to find all mitigation defects. We also show initial faster finding of mitigation defects for population 2000, but, again Random generation is unable to achieve detection of all mitigation defects. For all other population sizes GA is more efficient throughout.

4.5 Single Runs vs. Multiple Runs

The results of selecting the population for a generation are a set of test requirements (i.e. (p, e) pairs). They are used to generate failure mitigation paths and executable test cases. These must be executed and validated. All of these steps are manual, hence orders of magnitude more expensive than generating test requirements. This also makes the cost of multiple runs prohibitive. Multiple runs are possible when the use of a GA is explored with a simulator as in [58][13]. However, when actual test cases need to be generated, executed, and validated to determine a test requirements' fitness, this GA evaluation cost becomes prohibitive for multiple runs. We then must be willing to accept a local rather than global optimum as long as the mitigation defects are found, For quantitative results on evaluation cost see section 6.5 in our case study. Note also the global minimum in terms of number of test requirements is equal to the number of mitigation defects that exist.

Table 4.5: Efficiency: GA vs. Random

Search Space		**First defect Found**	**First reach 50%**	**First reach 100%**
200	**GA**	9	14	29
	R	7	12	n/a
400	**GA**	24	24	30
	R	21	21	n/a
600	**GA**	25	25	28
	R	21	21	n/a
800	**GA**	17	23	32
	R	28	48	n/a
1000	**GA**	37	37	37
	R	n/a	n/a	n/a
1200	**GA**	32	32	36
	R	n/a	n/a	n/a
1400	**GA**	27	27	34
	R	49	49	n/a
1600	**GA**	24	32	36
	R	87	n/a	n/a
1800	**GA**	20	24	26
	R	68	n/a	n/a
2000	**GA**	22	28	31
	R	17	93	n/a

Cantú-Paz [19] explore whether multiple runs of a GA can reach solutions of higher quality or reach acceptable solutions faster. Their results suggest that with a fixed evaluation budget a single run reaches a better solution than multiple independent runs.

We used the simulator with the data of subsystem CD of our case study in Chapter 6 to evaluate single vs. multiple runs. We perform independently 10 different runs. Table 4.6 shows the results. The first column identifies the generations. Next, for each of the runs (Run #1 to Run #10) two results are reported per generation: the number of individuals in a given generation (test requirements) and the cumulative population of defects found so far.

Table 4.6: Multiple Runs of GA

	Run#1		Run#2		Run#3		Run#4		Run#5		Run#6		Run#7		Run#8		Run#9		Run#10	
G1	11	0%	11	33%	11	33%	11	33%	11	33%	11	33%	11	33%	11	33%	11	33%	11	33%
G2	16	33%	15	33%	17	67%	16	33%	16	33%	16	33%	16	33%	15	33%	15	33%	15	33%
G3	18	33%	17	33%	21	67%	19	33%	19	67%	19	33%	18	33%	19	33%	18	33%	19	33%
G4	20	33%	18	33%	23	67%	20	67%	21	67%	20	33%	19	67%	22	33%	19	33%	21	33%
G5	23	33%	21	67%	24	67%	23	67%	24	67%	21	33%	23	67%	24	33%	21	67%	24	33%
G6	23	33%	23	67%	26	67%	26	67%	26	67%	26	33%	26	67%	26	33%	25	67%	27	33%
G7	24	33%	26	67%	28	67%	28	67%	27	67%	29	33%	27	67%	28	33%	26	67%	29	33%
G8	25	33%	30	67%	29	67%	31	67%	28	67%	30	33%	28	67%	29	33%	28	67%	30	33%
G9	27	33%	31	67%	33	67%	32	67%	29	67%	33	67%	30	67%	30	33%	30	67%	31	33%
G10	28	33%	32	67%	33	67%	33	67%	31	67%	36	67%	31	67%	30	33%	32	67%	32	33%
G11	30	33%	33	67%	35	67%	36	67%	33	67%	37	67%	32	67%	33	33%	35	67%	32	33%
G12	32	33%	34	67%	36	67%	38	67%	36	67%	40	67%	33	67%	35	33%	36	67%	34	33%
G13	34	33%	36	67%	38	67%	40	67%	38	67%	41	67%	34	67%	37	33%	37	67%	35	67%
G14	35	33%	37	67%	39	67%	40	100%	40	67%	41	100%	35	67%	38	33%	38	67%	38	67%
G15	38	33%	39	67%	41	100%			41	67%			36	67%	39	33%	40	67%	39	67%
G16	39	33%	40	67%					42	67%			37	67%	40	33%	40	100%	40	67%
G17	42	33%	41	67%					43	67%			38	67%	42	100%			41	100%
G18	44	33%	42	67%					43	100%			39	67%						
G19	45	100%	42	100%									40	67%						
G20													42	67%						
G21													42	100%						
	554		568		434		393		548		400		595		498		451		498	

As for effectiveness, each run finds all mitigation defects. The runs differ in efficiency. Run #7 needs the most generations (21), while Run #4 and Run #6 need the least (14). Similary, Run #7 generates the most test requirements (595) while Run #4 needs the fewest (392). By only performing one run, we risk sacrificing some efficiency. However, any of these runs requires fewer test requirements than an exhaustive search. As Chapter 6 will demonstrate, the cost of exhaustive search is prohibitive in practice.

Chapter 5

GA vs. Coverage Criteria

5.1 Coverage Criteria (CC)

We did compare the performance of the GA to random selection of fail-safe scenarios, and the results show GA performance was better than random selection. In this chapater, we compare the approach against Coverage Criteria defined by Andrews et al. [6]:

Criteria 1 (C1): All combinations, i.e. all positions p, all applicable failure types e (test everything). This is clearly infeasible for all but the smallest models. It would require $length(I) \times |F|$ pairs if SE contains all "1"s.

Criteria 2 (C2): All unique nodes, all applicable failures. This only requires: $\sum_{j=1}^{k} \sum_{i=1}^{|S|}$ (SE(i,j)=1) combinations i.e. the number of one entries in the State-Faiture matrix (SE). When some nodes occur many times in a test suite only one needs to be selected by some scheme. This could lead to not testing failure recovery in all tests. An alternative is to require covering each test as well.

Criteria 3 (C3): All tests, all unique nodes, all applicable failures. Here we simply require that when unique nodes need to be covered they are selected from tests that have not been covered.

A weaker criterion is not to require covering all applicable failures for each selected position. Note that C2 and C3 require the same number of pairs. Its effectiveness is the same, whether a failure f occurs in position p or p' where $node(p) = node(p') = s$ is irrelevant for mitigation failure detection. That is, a test would find (or not find) a mitigation defect for a failure in state s, no matter where in the test suite s occurs.

Criteria 4 (C4): All tests, all unique nodes, some failures (only one failure per position, but covering all failures). Some failure means that collectively all failures must be paired with a position at least once, but not with each selected position as in criteria 3.

Coverage criteria are attractive, since they allow for systematic algorithmic generation of points of failure and failure types. Depending on the criteria, a strong one may not scale to larger problems while a weaker one may not be effective. For example, requiring all applicable failure events in all states in each test to be covered, is infeasible for all but the smallest behavioral models and a small number of failure types. For smaller test problems, test criteria are feasible and may be better.

5.2 Design of Experiments

We extended the simulator in section 4.1 to run experiments to investigate the following research questions:

- Under which conditions can we use the GA or the coverage criteria?
- How does the GA test strategy compare to the use of the four coverage criteria (C1-C4) both with respect to efficiency and effectiveness?

- Is there a difference in performance when we have a small search space vs. a large one?

The simulator takes the same independent and dependent variables as described in section 4.1.

We report on a series of simulation experiments that compares the two strategies with respect to effectiveness and efficiency. Specifically, we compare the two strategies in light of different problem sizes. By problem size we mean the combination of number of failure types and size of the test suite. GAs tend to work well in large search spaces, so the question becomes at what point is the problem too small for a GA to be used meaningfully. Conversely, depending on the test criteria, some may become rather expensive, i.e leading to a large number of testing requirements.

We investigate and compare the two strategies for large and small problems and considers both efficiency and effectiveness. We do this via simulation experiments. To increase validity, we also present a comparisons of GA and coverage criteria for generating test requirements via a series of case studies:

- Three web applications (student services, a mortgage system, and one of its subsystems)
- Four safety critical systems (Two versions of a railroad crossing control system (RCCS), an insulin pump, and an aerospace launch vehicle)

The case studies are investigated by seeding mitigation defects and evaluating whether the GA and/or the coverage criteria find the defects and how many test requirements are needed.

5.2.1 Parameter Settings

For the GA parameters, we use the same parameters as were identified and experimented with in section 4.1: mutation rate MR=0.3, crossover CR=0.5, number of runs NR=10. We chose a defect rate D that is close to the one reported by Sawaelpong *et al.* [67]: D=20%. Based on the results of tuning the GA in section 4.3 (see Table 4.2) we selected exploration weights w_r=3 and w_s=1. We vary the size of $length(I) \times |F|$ from 10 to 2000. We keep the applicability level at the same level as in the prior experiments. The duplication factor is kept at 2 for the small search space (10-100). That means on average a specific state in the behavioral model occurs in the test suite twice. Note that the duplication factor of one of the RCCS example is about 3. The duplication factor for the case studies reported in section 5.3 for the small search spaces vary from 2-14 (rounded to closes integer). A lower duplication factor makes for a more difficult search problem since there are fewer opportunities to find a mitigation defect for a failure that occurs in a particular state. For the larger search spaces, we set the duplication factor to 20. This is smaller than the duplication factor of 31 for the large case study reported in section 5.3, again, to make it a harder problem, so as to be conservative. Table 5.1 summarizes the parameter selection for the typical mitigation defect density of 20%.

Table 5.1: Simulation parameter for typical D=20%

$length(I) \times \|F\|$	DF	CR	MR	w_r	w_s	NR
10-100	2	0.5	0.3	3	1	10
200-2000	20	0.5	0.3	3	1	10

Table 5.2 summarizes the simulation parameters for the low defect density of D=5%. Note that D=5% for $length(I) \times |F|$=10 is not possible as it results in less than one mitigation defect.

Table 5.2: Simulation parameters for low defect density D=5%

$length(I) \times \|F\|$	DF	CR	MR	w_r	w_s	NR
20-100	2	0.5	0.3	1	3	10
200-2000	20	0.5	0.3	1	3	10

We now use the simulator with these parameters to compare GA performance against coverage criteria (C1-C4).

5.2.2 GA vs. C1

C1 represents the equivalent of an exhaustive search as the test requirements ((p, e) pairs) state that all feasible combinations be tested. C1 guarantees 100% mitigation defect coverage. Table 5.3 shows simulation results for a mitigation defect density of 20%. The first column shows the potential search space $length(I) \times |F|$ (the GA removes infeasible pairs based on the SE Matrix). It varies from 200-2000. The second column shows the number of pairs the GA generates. The third column shows the percentage of mitigation defects found by the pairs reported in column 2. Columns 4 and 5 report this information when using coverage criterion C1.

Both GA and C1 find all mitigation defects, but GA does so much more efficiently. Column 6 shows this quantitatively by computing the fraction of pairs needed by the GA vs C1. The GA only needs between 3.4%-5% of the pairs required by C1, it is clearly more efficient. Both approaches are equally effective.

Next, we consider the typical mitigation defect rate of 20% with the small search space. Table 5.4 shows these results. It is organized the same as Table 5.3. While the GA again generates fewer pairs it is not able to find all mitigation defects until $length(I) \times |F|$ reaches 70. Hence C1 is more effective for $length(I) \times |F| \in \{10, 20, 30, 40, 50, 60\}$. It is also interesting to note that the relative efficiency (column 6) is not as good as for the larger search spaces. The GA now needs between 33.3%-62.5% of the number of pairs C1 requires. We have to conclude then that for this experiment, C1 is recommended over GA as long as $length(I) \times |F| \leq 60$.

Next, we turn to the low mitigation defect rate of 5%. Table 5.5 shows the results for $I \times |F|$ ranging between 200-2000 while Table 5.6 shows it for $length(I) \times |F|$ ranging from 20-100. Both tables are organized identical to Table 5.3.

A defect density of 5% represents a more difficult search problem. Table 5.5 shows that for the larger potential search space the GA finds all mitigation defects. What is interesting is that it does not need many more pairs to do this than for the larger mitigation defect rate. The relative efficiency compared to the larger mitigation defect ratio is also similar (Table 5.3 shows a slightly wider range [3.4-5.0] versus [3.8-4.6] in Table 5.5).

We next investigate the low mitigation defect rate D=5% for the small search space. Results are reported in Table 5.6 (organized the same as Tables 5.3-5.5). Note that it was not possible to investigate $length(I) \times |F| = 10$, since with a D=5% this would result in less than one defect, hence is not possible.

As before for the higher defect rate (Table 5.4), the GA is unable to detect all mitigation defects for the very small search spaces (less than 80 in this case), hence C1 is more effective. When the defect rate decreases, it takes a larger potential search space (80 vs. 70) to become successful at finding all defects. Note that we do not report relative efficiency until both GA and C1 find all defects since comparing efficiency of an ineffective approach compared to one that does find all defects is pointless.

Table 5.3: Large Search Space: GA vs. C1 - 20% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C1** pairs required%	**C1** defect	**GA/C1**
200	7	100%	160	100%	4.3%
400	11	100%	320	100%	3.4%
600	17	100%	480	100%	3.5%
800	25	100%	640	100%	4.0%
1000	28	100%	800	100%	3.5%
1200	30	100%	960	100%	3.5%
1400	39	100%	1120	100%	3.1%
1600	53	100%	1280	100%	5.0%
1800	57	100%	1440	100%	4.0%
2000	70	100%	1600	100%	4.3%

5.2.3 GA vs. C2/C3

Next, we compare GA performance to coverage criteria C2 and C3. C2 and C3 result in the same number of test requirements. Although positions in a pair may differ, the same combinations of states and failures are tested (see section 5.1). In other words, C2 and C3 do not differ in what states and failures need to be covered, only in whether states can be selected from any $t \in BT$ or whether all $t \in BT$ need to be covered. This results in selection of different positions p for C2 vs. C3.

Table 5.4: Small Search Space: GA vs. C1 - 20% defect density

$length(I) \times \lvert F \rvert$	**GA** # of pairs	**GA** Defect %	**C1** pairs required%	**C1** defect	**GA/C1**
10	5	50%	8	100%	62.5%
20	7	50%	16	100%	43.8%
30	8	50%	24	100%	33.3%
40	12	50%	32	100%	37.5%
50	16	67%	40	100%	40.0%
60	23	83%	48	100%	47.9%
70	26	100%	56	100%	46.4%
80	28	100%	64	100%	43.8%
90	32	100%	72	100%	44.4%
100	35	100%	80	100%	43.8%

Table 5.5: Large Search Space: GA vs. C1 - 5% defect density

$length(I) \times \|F\|$	**GA**	**GA**	**C1**	**C1**	**GA/C1**
	# of pairs	Defect %		defect	
			pairs required%		
200	6	100%	160	100%	3.8%
400	12	100%	320	100%	3.8%
600	22	100%	480	100%	4.6%
800	27	100%	640	100%	4.2%
1000	34	100%	800	100%	4.3%
1200	43	100%	960	100%	4.5%
1400	46	100%	1120	100%	4.1%
1600	60	100%	1280	100%	4.7%
1800	61	100%	1440	100%	4.2%
2000	72	100%	1600	100%	4.5%

Table 5.6: Small Search Space: GA vs. C1 - 5% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C1** pairs required%	**C1** defect	**GA/C1**
20	9	0%	16	100%	
30	11	0%	24	100%	
40	15	0%	32	100%	
50	16	50%	40	100%	
60	20	50%	48	100%	
70	24	50%	56	100%	
80	30	100%	64	100%	46.9%
90	35	100%	72	100%	48.6%
100	37	100%	80	100%	46.3%

As before, we first analyze the performance of GA vs coverage criteria for the typical mitigation defect rate of 20% (both large and small problem), then for the low mitigation defect rate of 5%. All other parameter values are as indicated in section 5.2.1.

Table 5.7 shows the results for D=20% and $length(I) \times |F| \in \{200,...., 2000\}$. It is organized identical to the earlier result tables. Both GA and C2/C3 are able to find all mitigation defects (see columns 3 and 5). The number of pairs does not differ as much as for GA vs. C1 (section 5.2.2), although GA needs fewer pairs. The relative efficiency of the GA is between 62.5% and 87.5%.

We next turn to the small potential search space. Table 5.8 shows results for the small search spaces ranging from 10-100 and a mitigation defect density of

Table 5.7: Large Search Space: GA vs. C2/C3 - 20% defect density

$length(I) \times \|F\|$	**GA**	**GA**	**C2/C3**	**C2/C3**	GA/(C2/C3)
	# of pairs	Defect %		defect	
			pairs required	%	
200	7	100%	8	100%	87.5%
400	11	100%	16	100%	68.8%
600	17	100%	24	100%	70.8%
800	25	100%	32	100%	78.1%
1000	28	100%	40	100%	70.0%
1200	30	100%	48	100%	62.5%
1400	39	100%	56	100%	69.6%
1600	53	100%	64	100%	82.8%
1800	57	100%	72	100%	79.2%
2000	70	100%	80	100%	87.5%

20%. The GA does not reach 100% effectiveness until a search space of 70, showing C2/C3 is more effective for search spaces ranging from 10-60. The number of pairs is comparable: C2/C3 ranges from 5-40 pairs, GA ranges from 5-35 pairs. We thus recommend to use C2/C3 for potential search spaces of less than 70, and GA for larger ones.

Table 5.8: Small Search Space: GA vs. C2/C3 - 20% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C2/C3** pairs required	**C2/C3** defect %	GA/(C2/C3)
10	5	50%	5	100%	
20	7	50%	8	100%	
30	8	50%	12	100%	
40	12	50%	16	100%	
50	16	67%	20	100%	
60	23	83%	24	100%	
70	26	100%	28	100%	92.9%
80	28	100%	32	100%	87.5%
90	32	100%	36	100%	88.9%
100	35	100%	40	100%	87.5%

As before, we next consider the lower mitigation defect rate of D=5%. Table 5.9 shows results for the large potential search space while Table 5.10 shows results for the small one. As with the higher mitigation defect rate, both GA and C2/C3 are able to find all defects for the larger search space.

What is interesting is that decreasing the defect rate does not appear to affect the number of pairs the GA needs very much although the trend is for an increased number of pairs for the smaller mitigation defect rate. This is also illustrated by comparing relative efficiency: for the lower defect rate, GA needs up to 93.8% of pairs C2/C3 needs while for the higher one GA needs no more than 87.5% of the number of pairs C2/C3 requires.

Similarly, when comparing results for the small search space between the lower (Table 5.10) and higher (Table 5.8) defect rates, initially the GA is not able to find all defects ($length(I) \times |F| \leq 80$). When it finally does, it requires slightly fewer pairs than C2/C3.

Overall, it appears that lowering the defect rate from 20% to 5% does not have a huge impact in effectiveness or efficiency. C2/C3 and GA are equally effective, with GA being only slightly more efficient.

In summary, for large search spaces GA and C2/C3 are equally effective; the GA has a slight advantage in efficiency over C2/C3. Based on the results, we recommend using coverage criteria for small search spaces (until 70 for D=20% and until 80 for D=5%) over using a GA.

5.2.4 GA vs. C4

C4 is the weakest of the four coverage criteria and requires the fewest (p,e) pairs. We first compare results for large search spaces and defect rates of 20% (Table 5.11) and 5% (Table 5.13). In both cases the results are very strong: The GA finds all defects while C4 is unable to find any. C4's relative higher efficiency is hence irrelevant. The criteria is too weak. We turn to the small search space next. Table 5.12 reports results for D=20% while Table 5.14 summarizes results for D=5%. The GA does not find all defects until $length(I) \times |F|$ reaches 70 (D=20%) and 80 (D=5%). Therefore, we can not recommend GA for small potential search spaces.

Table 5.9: Large search space of GA vs. C2/C3 - 5% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C2/C3** pairs required	**C2/C3** defect %	GA/(C2/C3)
200	6	100%	8	100%	75.0%
400	12	100%	16	100%	75.0%
600	22	100%	24	100%	91.7%
800	27	100%	32	100%	84.4%
1000	34	100%	40	100%	85.0%
1200	43	100%	48	100%	89.6%
1400	46	100%	56	100%	82.1%
1600	60	100%	64	100%	93.8%
1800	61	100%	72	100%	84.7%
2000	72	100%	80	100%	90.0%

Table 5.10: Small Search Space: GA vs. C2/C3 - 5% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C2/C3** pairs required	**C2/C3** defect %	GA/(C2/C3)
20	9	0%	8	100%	
30	11	0%	12	100%	
40	15	0%	16	100%	
50	16	50%	20	100%	
60	20	50%	24	100%	
70	24	50%	28	100%	
80	30	100%	32	100%	93.8%
90	35	100%	36	100%	97.2%
100	37	100%	40	100%	92.5%

What is interesting, however, is that C4 does not detect all defects for any of the potential search spaces, although it does better than for the larger potential search spaces. For the 20% defect rate (Table 5.12) it starts out with detecting 50% of the defects for $length(I) \times |F| = 10$ and decreases steadily to 20% when $length(I) \times |F| = 90$. It appears that as the potential search space increases, its ability to find mitigation defects decreases. When considering the lower defect rate D=5%, the results do not show a trend (Table 5.14, column 5) such as decreasing effectiveness with increasing potential search space.

In summary, the GA is more effective than C4 for large search spaces for both defect rates. It has issues finding defects for small search spaces. Overall, we cannot recommend C4.

Table 5.11: Large search space: GA vs. C4 - 20% defect density

$length(I) \times \lvert F \rvert$	**GA** # of pairs	**GA** Defect %	**C4** pairs required %	**C4** defect
200	7	100%	2	0%
400	11	100%	4	0%
600	17	100%	6	0%
800	25	100%	8	0%
1000	28	100%	10	0%
1200	30	100%	12	0%
1400	39	100%	14	0%
1600	53	100%	16	0%
1800	57	100%	18	0%
2000	70	100%	20	0%

Table 5.12: Small Search Space: GA vs. C4- 20% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C4** pairs required	C4 defect
10	5	50%	2	50%
20	7	50%	5	50%
30	8	50%	8	50%
40	12	50%	10	50%
50	16	67%	13	50%
60	23	83%	15	43%
70	26	100%	18	43%
80	28	100%	20	38%
90	32	100%	23	20%
100	35	100%	25	20%

Table 5.13: Large search space: GA vs. C4 - 5% defect density

$length(I) \times \lvert F \rvert$	**GA**	**GA**	**C4**	**C4**
	# of pairs	Defect %		defect
			pairs required %	
200	6	100%	2	0%
400	12	100%	4	0%
600	22	100%	6	0%
800	27	100%	8	0%
1000	34	100%	10	0%
1200	43	100%	12	0%
1400	46	100%	14	0%
1600	60	100%	16	0%
1800	61	100%	18	0%
2000	72	100%	20	0%

Table 5.14: Small Search Space: GA vs. C4- 5% defect density

$length(I) \times \|F\|$	**GA** # of pairs	**GA** Defect %	**C4** pairs required	C4 defect
20	9	0%	5	50%
30	11	0%	8	50%
40	15	0%	10	0%
50	16	50%	13	50%
60	20	50%	15	0%
70	24	50%	18	0%
80	30	100%	20	0%
90	35	100%	23	33%
100	37	100%	25	0%

5.3 Comparison of Various Case Studies and Model Types

In this section, we compare different case studies with respect to size of PE and effectiveness and efficiency of using GA vs. C1-C4 when seeding 5% of the mitigations with defects. We chose D=5% to make it a more difficult search problem. Table 5.15 summarize the case studies. The first column identifies the type of model used. There are three case studies using FSMWeb [9], one case study using an EFSM (Extended Finite State Machine), and three case studies using a Communicating Extended Finite State Machine (CEFSM). Column 2 identifies the case studies and provides a reference where details of the case study can be found. The first is a student services web application (CSIS), the second a large mortgage application,

Table 5.15: Summary of case studies using different behavioral models

Behavioral Model	**Case Study**	**# of States**	**# of Transitions**	**# of F**	**Size of BT**	***Length(I)***	AL
FSMWeb [9]	CSIS [9]	16	20	3	6	70	33.33%
	Mortgage Syatem	127	224	10	266	3998	40.00%
	Closing Documents Sub System	12	9	10	12	169	41.67%
EFSM[45]	RCCS -1 [6]	4	8	4	4	24	81.25%
CEFSM [45]	Launch System [26]	21	34	14	5	49	45.92%
	Insulin Pump [26]	15	23	4	11	74	61.67%
	RCCS-2 [26]	14	19	4	11	58	60.71%

underwriting, and management system, the third is one of its subsystems that deals with closing documents. These are all web applications. The Railroad crossing control system (RCCS-1) in EFSM format is a simplified version of the RCCS-2 in CEFSM format. Additionally, we also use CEFSM models of an aerospcae launch vehicle and an insulin pump. These are all examples of safety critical systems. The next two columns show the number of states and transitions, respectively. They vary from 4 in the RCCS-1 to 127 in the mortgage system, and from 8 transitions in RCCS-1 to 224 for the mortgage system. Column 5 shows the number of failure types. CSIS only has 3, while the launch vehicle has 14. Column 6 lists the number of tests for each case study, ranging from 4 to 266. The length of the test suite is given in column 7. It ranges from 24 to 3998. The product of columns 5 and 7 defines the size of the search space (minus infeasible pairs). The last column shows the applicability levels for each case study. Applicability levels are usually lower when certain failures only apply in certain phases of processing. For example, in the launch vehicle case study, failure types are specific to a launch phase and are not applicable to earlier or later phases. The highest is for the RCCS-1.

The mortgage system is the largest. The insulin pump and RCCS-2 have the same number of test cases and the same number of failures, but they differ in the size of the search space. RCCS-1 has the smallest search space. As mentioned before, we seeded each case study with 5% defective mitigations. Table 5.16 shows the size of the search space $length(I) \times |F|$ in column 2 for each case study summarized in Table 5.15. The remaining columns show the number of (p, e) pairs needed when using GA, C1, C2, C3, and C4, respectively. Except for C4, all find 100% of the defective mitigations. The last column shows the percentage of defective mitigations found by C4.

As we showed in the simulation experiments, using the GA is much more efficient than C1. It compares in efficiency with C2 and C3. C4 is too weak a testing criterion and is unable to find all mitigation defects except for RCCS-1. For four of the seven case studies, it is unable to find any of the mitigation defects.

Table 5.16: Efficiency Comparison

Application	$length(I) \times \|F\|$	**GA**	**C1**	**C2**	**C3**	**C4**	**C4 Defect Found %**
CSIS	210	14	97	16	16	9	0%
Mortgage System	39980	485	27986	508	508	127	0%
CD	1690	41	638	50	50	12	0%
RCCS-1	96	17	76	13	13	4	100%
Launch System	686	128	386	135	135	21	0%
Insulin Pump	296	25	189	37	37	15	50%
RCCS-2	232	29	138	34	34	14	50%

Chapter 6

Case Study - Mortgage System

6.1 General Description

We illustrate our approach on a commercial web application. The mortgage system is an example of a critical web application as failures can be drastic: borrowers lose their home, the company loses its value, and employees lose their jobs. Not all system components are critical, e.g. components not related to the loan process. The system provides different services in each stage of the loan process. It includes the following functions:

1. Create the loan.
2. Acknowledge the loan based on the type of pricing.
3. Review the loan to make an approval decision.
4. Request legal documentation via external web services (Document disclosure).
5. Keep and update accounting information for funding and selling the loan through selected warehouse banks and investors who buy loans.
6. Close the loan by shipping and tracking the loan's data with the investor.

7. Manage various data used in the system e.g. adding/editing/deleting users or investors via an administration tool.

8. Provide utilities for loan processing e.g. import/export loan data.

First, the loan officer users (LO) start creating the loan in the system by entering borrower information and loan data. The system integrates with other big warehouse mortgage data systems to import loan data via an external web service. After creating the loan, the quality control users (QC) will price the loan and set up the loan type and loan program for acknowledgement. Next, reviewer users (RU) evaluate the loan data and provide the decision (approve or decline). In some situations, the reviewer could decide to suspend the loan. Once the loan is approved, the loan specialist users (SU) make sure the loan documents are legal and assign loan details. Loan specialist will request the legal document for the loan agreement using an external web service. Accounting users (AU) fill out the required information for funding the loan and assigning the loan to a warehouse bank based on the credit available, such as the funding amount and funding date. Accounting users will also enter selling information once the loan is sold to a target investor. The closer users (CU) ship the loan to the investor and track investor deficiencies. The loan tool helps the users to generate and import loan data using different formats. Finally, a management tool exists to administer loan data in the system (e.g. users, banks, and investors data). This is accessible only for Admin users. The system requires a user name and password to login. Based on user type different functions are available. Figure 6.1 shows the loan processing by user type. The system is built using ASP.net, C#, Telerik Rad Controls for ASP.net AJAX, Hibernate technology, SQL server 2008R, and IIS7 technologies. The system was built by 20 sub-projects.

It consists of 6887 files with a total size of about 1.48 Gigabytes. Table 6.1 shows some technical details about the system.

Table 6.1: Technical details of Mortgage System

Cyclomatic Complexity	170476
Max. Depth of Inheritance	9
Max. Class Coupling	1490
Lines of Code	257592
Number of Files	6887
Size in Gigabytes	1.48
SQL database Tables	204
Number of web pages	127

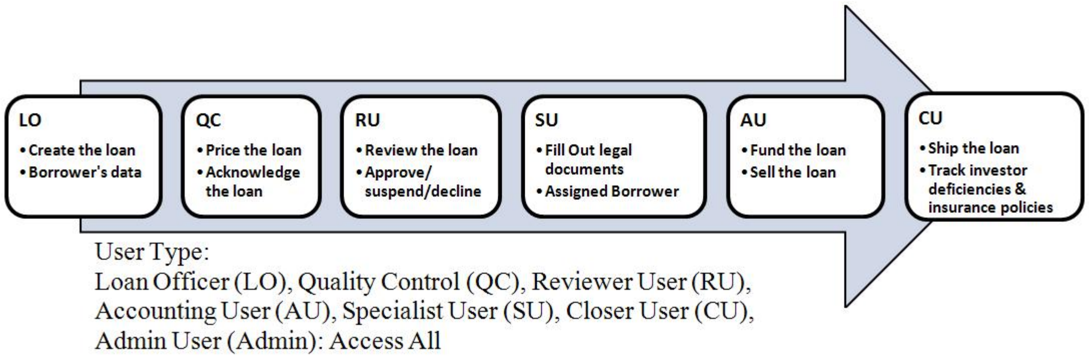

LO
• Create the loan
• Borrower's data
QC
• Price the loan
• Acknowledge the loan
RU
• Review the loan
• Approve/ suspend/decline
SU
• Fill Out legal documents
• Assigned Borrower
AU
• Fund the loan
• Sell the loan
CU
• Ship the loan
• Track investor deficiencies & insurance policies
User Type:
Loan Officer (LO), Quality Control (QC), Reviewer User (RU),
Accounting User (AU), Specialist User (SU), Closer User (CU),
Admin User (Admin): Access All

Figure 6.1: Loan processing user types and access privileges.

6.2 FSMWeb Behavioral Model

6.2.1 Partition FSMWeb model

Figure 6.2 shows a logical view of mortgage system, and its HTML links. The system requires the user to enter the user name and password. Then web services are available based on user type. The home page shows all loans that will be closed in the current week. The user can also navigate to show loans in the next or previous

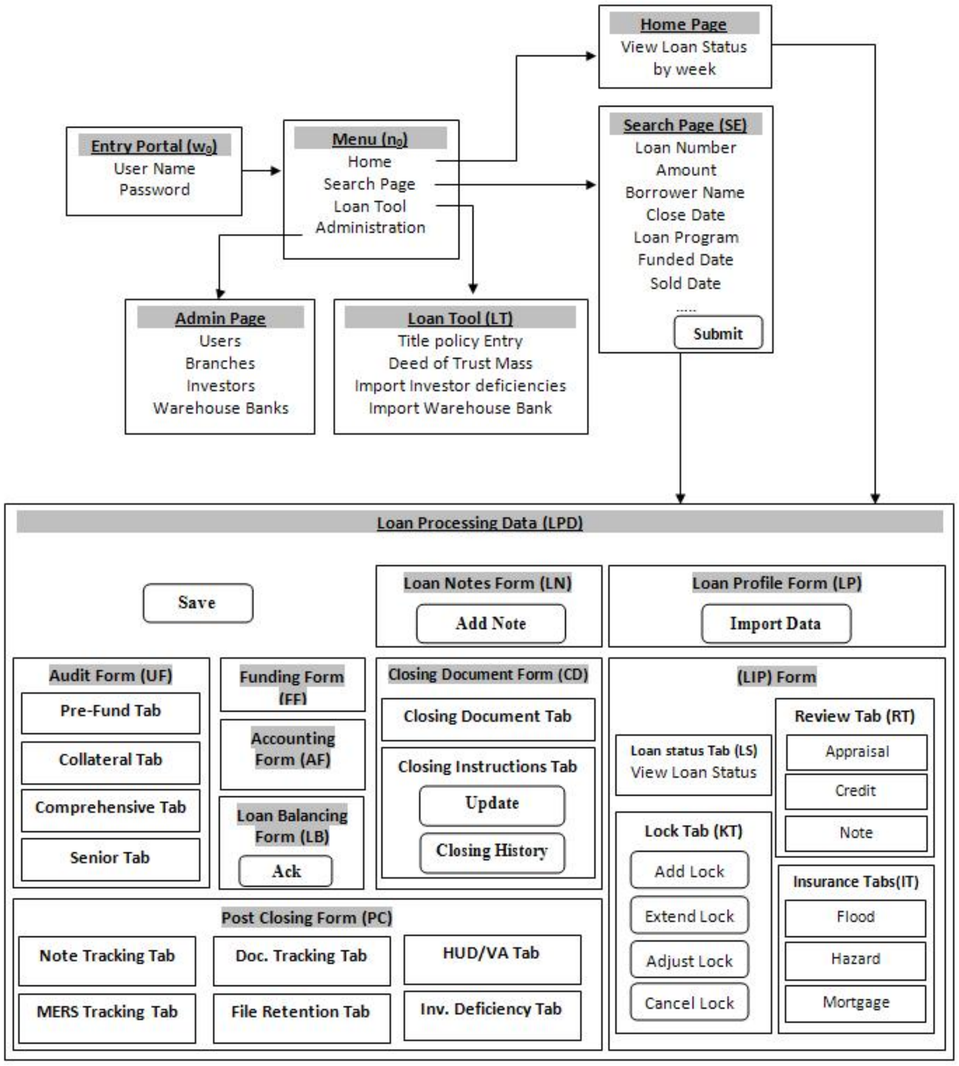


Figure 6.2: Mortgage System Logical View

week. Some user types can see all system loans in the home page except LO, SU and RU who can only access their loans since every loan is tied to the LO, SU and RU in the loan processing life cycle. Figure 6.3 illustrates valid navigation across the top level partitions of the system. Loan processing data (LPD) can be accessed either from the home page or by searching a specific loan. From the home page, the user can access specific LPD by clicking on the loan number that appears in a selected week. In the search page, the result of a search is a list of loans. The user can access any loan from the list by clicking on the loan number, or if the search resulted in just one explicit loan, the system will be automatically directed to view the LPD for that loan. Loan processing data (LPD) includes all data information that is related to the loan processing as shown in Figure 6.1. LPD is divided into nine different sub web services: Loan Profile (LP), Loan in process (LIP), Closing Document (CD), Funding Form (FF), Accounting Form (AF), Audit Form (UF), Loan Balancing (LB), Post Closing (PC), and Loan Notes (LN). The LP page represents all main loan information such as loan number, borrower name, loan amount etc. Loan in process (LIP) is a cluster service derived from the LPD cluster. LIP is further divided into four clusters that include loan status (LS), lock screen (KT), review screen (RT), and insurance screen (IT). The Loan status (LS) form shows all important dates related to the loan processing. The Lock tab (KT) represents all required data for acknowledgment of the loan prices. The Review tab (RT) contains conditions that need to be met prior to closing and the disposition data for the loan that shows if the loan is approved or not. All information related to insurance companies are located on insurance tab (IT). Next, the Closing Document (CD) cluster contains all pages related to legal loan documents and fees as well as closing instructions. Funding information is showing in the Funding Form (FF) cluster, and Accounting data related to the loan is represented in the Accounting Form (AF) partition. A list of all audit questions related to the loan are in the Audit Form (AF) cluster. The Loan Balance form (LB) cluster shows the loan balance. All tracking and shipping data are in the Post Closing form (PC) cluster. Any comment and notes are represented on the Note Form (LN) cluster. Navigation among the LPD clusters is shown in Figure 6.4. The Loan Tool (LT) cluster includes all utility pages that relate to import or export of data to and from the system. Finally, the Administration (Admin) cluster represents all management data used in the system. Table 6.2 identifies who will have the right to access a particular loan processing service within the loan status which is required for every single loan function. For example, the loan status must be in Open status and the user type has to be a quality control user (QC) in order to lock the loan and have access to the lock tab (KT) cluster in the LDP cluster. Table 6.2 shows all cluster services at various levels next to the user type and loan status for each service.

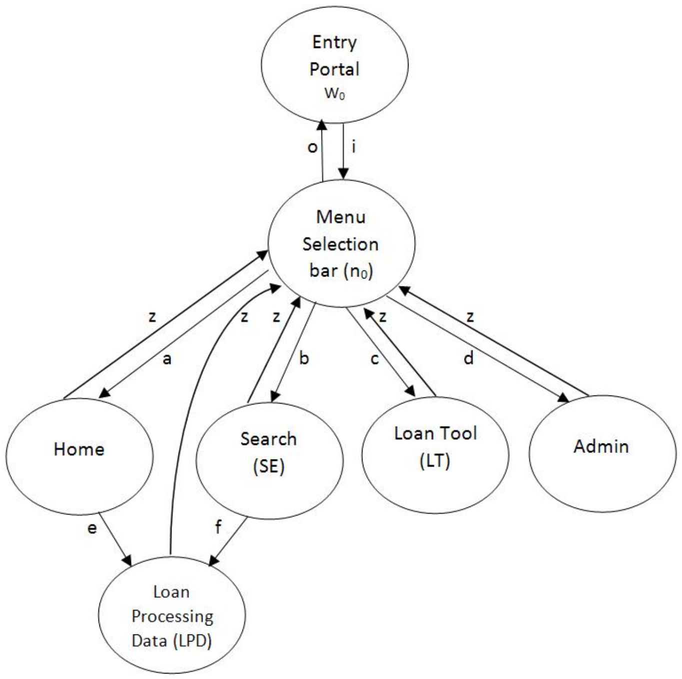


Figure 6.3: Aggregate FSMs with Partition and Top Level Navigation

Table 6.2: Decomposition of Mortgage system into Partitions

AFSM	**Cluster**		**User Type**	**Loan status**
Entry Portal (w_0)			All	Any
Home Page			All	Any
Search Page (SP)	Simple Search /Advance Search		All	Any
Loan Processing Data (LPD)	Loan Notes Form (LN)		All	Any
	Loan Profile Form (LP)		LO	Open
	LIP	Loan status tab (LS)	All	Any
		Lock Tab (KT)	QC	Open-Lock
		Review Tab (RT)	RU	Lock-Suspended Approve with condition
		Insurance Tab (IT)	RU	Lock-Suspended Approve with condition
	Closing Document Tab (CD)		SU	Approve to close Approve with condition
	Funding Form (FF)		AU	Approve to close Funding - Sold
	Accounting Form (AF)		AU	Approve to close Funded – Sold
	Audit Form (UF)		QC	Not Funded or Sold
	Post Closing Form (PC)		CU	Funded – Sold
	Loan Balance Form (LB)		QC/LO	Approve to close Approve with condition
	Note (LN)		All	Any
Loan Tool (LT)	Title policy Entry (LT1)		CU	Funded - Sold
	Deed of Trust Mass (LT2)		CU	Funded - Sold
	Import Investor deficiencies (LT3)		QC	Funded - Sold
	Import Warehouse Bank (LT4)		QC	Funded - Sold
Admin	Users (A_1)		Admin	NA
	Branches (A_2)			
	Investors (A_3)			
	Warehouse Banks (A_4)			

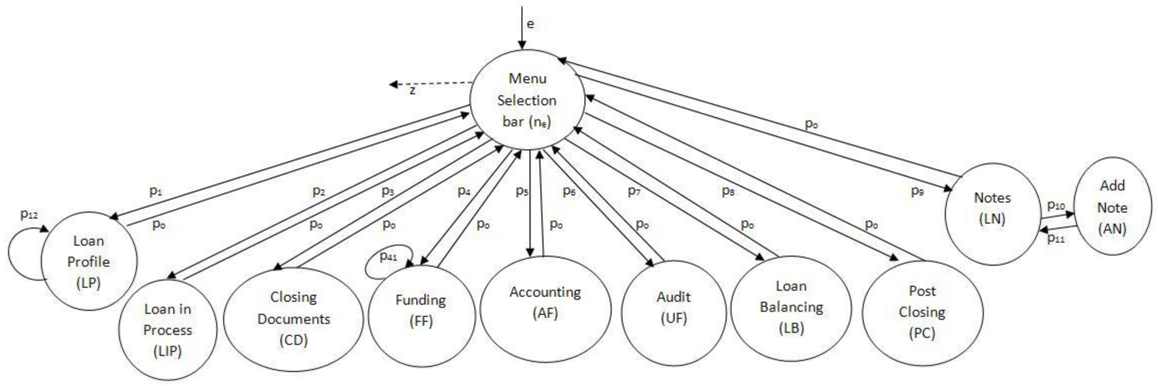


Figure 6.4: Loan Processing Data (LPD) Cluster

The full model of this commerical web application consists of (127) LWPs, (22) clusters, and (224) transitions. Due to page limitations, we present only a key portion to illustrate the approach. We will detail the Closing Documents (CD) cluster as an example. Figure 6.5 shows the navigation among the logical web pages that represent the Closing Documents (CD) service. Table 6.3 shows the nodes and logical web pages related to (CD). Table 6.5 shows the notations of the aggregate FSMs that is presented in Figure 6.3. The navigation is based on the following design decisions for the mortgage system:

- The Loan status should be Approved by (RU) user.
- Only (SU) user role has access to the service.
- SU can request the loan documents via an calling external web service either through the Documents to Close page (DC) or the Closing instructions (CI) page.

Obviously, other choices for which navigation is allowable could be defined, but this initial set is used to illustrate the testing technique. We also decided to simplify the design somewhat. For example, we are not showing cancel buttons for each web page. Thus the example shows how one would deal with this kind of navigation without the extra clutter of showing all cancel buttons one might have.

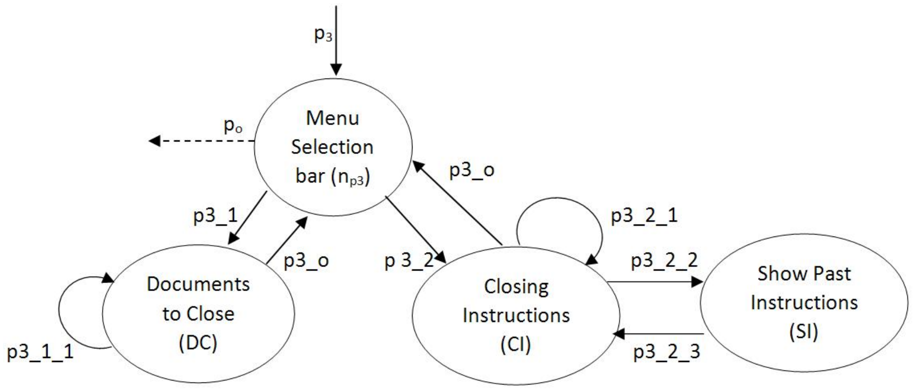


Figure 6.5: FSM For Closing Documents (CD) Service

Table 6.3: Nodes for CD Form-FSM

Node	LWP	Explanation
n_{p3}	Selection tab menu	Select service from tab
DC	Documents to close	Request legal document and showing requested documents
CI	Closing Instructions	Request legal document and showing closing fees
SI	Show past instructions	Show the history of requested documents

Similarly, we define logical web pages and navigation among them for the Home cluster. There is only one cluster node in the Home cluster as shown in Figure 6.6. Next, we describe the logical web pages for the Search Page (SE) cluster. Figure 6.7

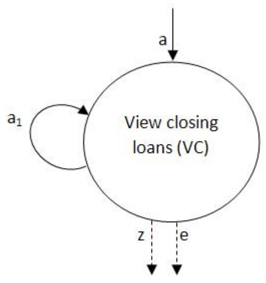


Figure 6.6: Logical Web pages and Navigation for Home Cluster

shows the logical web pages and the navigation among them. Table 6.4 summarizes these nodes and corresponding logical web pages.

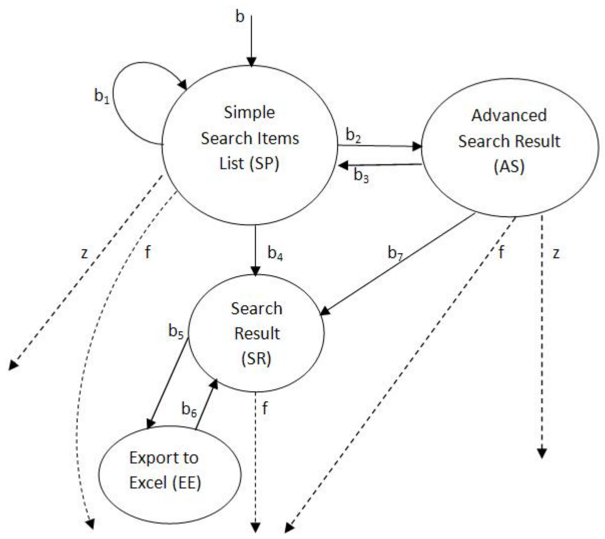


Figure 6.7: Logical Web pages and Navigation for Search Cluster

Table 6.4: Nodes for SE Form-FSM

Node	LWP	Explanation
SP	Simple Search Page	use simple query form to search for loan
SR	Search Result Page	Display the result of search
EE	Export to Excel	Export the result of the search to Excel
AS	Advance Search Page	Search for any loan based on a description

6.2.2 Input Constraints for Logical Web Pages

Table 6.5 shows all annotations on Aggregate FSM transitions for the top level navigation of the mortgage system (see Figure 6.3). Table 6.6 and Table 6.7 show the input constraints for Home and Search clusters. The FSM transition constraints for LPD of Figure 6.4 are shown in Table 6.8. Table 6.9 demonstrates all FSM transition constraints for the CD cluster as shown in Figure 6.5. In the transition annotation tables, the leftmost column encodes each set of constraints (Σ). The second column describes the action of the transitions, the third identifies the constraints, the forth column identifies transitions by their pre- and post-nodes, and the fifth column (Ω) lists the next node, or output. This information is used to provide a partial test oracle for the test input. As an example, the first row in table 6.9 is a transition from n_{p3} (menu selection bar) to access the Documents to Close page (DC) tab. The codes are assigned arbitrarily as a matter of convenience: as a reference for the full constraint in the constraint columns (column 3). The outputs (in the column marked Ω) are the target states of the transitions. These will be used during execution as a test oracle to check against the state the application actually reached.

6.2.3 Generate Test Paths through Clusters

For the Closing Documents (CD) service, we apply transition coverage to the FSMs, generating test sequences to cover each transition. A test sequence is a sequence of FSM edges and their annotations (constraints). Test sequences for the `CD` FSMs is shown in Table 6.10. The first column indicates which edges are covered, the second column indicates the constraint sequence (using the input alphabet defined in the Σ columns in Table 6.9. Thus, the first sequence in Table 6.10 for the `CD` FSM covers edges:(n_{p3},CI) (CI,SI)(SI, CI)(CI,n_{p3}) from Table 6.10 and the FSM in Figure 6.5,and uses the test sequence of constraints: p3_1, p3_o. We apply edge coverage to FSMs for generating test cases as given in Table 6.11. For more detail on graph based testing criteria see [3]. Test paths of the LPD cluster are given in Table 6.12. Table 6.13 and Table 6.14 represent the test paths for the Home and Search pages. Test paths for Closing Documents (CD) are shown in Table 6.15.

Table 6.5: Annotation of Aggregate FSMs for Mortgage System

Σ	**Actions**	**Constraints**	**Transition**	Ω
z	Dummy edge, reflect access to menu bar	R(n_0,Click)	(Home/SE/LT/ LPD/Admin,n_0)	n_0
a	Access Home tab	R(tab=Home,Click), Any(User Type,Loan Status)	(n_0,Home)	Home
b	Access Search tab	R(tab=SP,Click), Any(User Type,Loan Status)	(n_0,SE)	SE
c	Access Loan Tool tab	R(tab=LT,Click), R(User Type in (CU,QC,AU,RU), Loan Status in (Open,Lock, Approve to Close,Funded,Sold))	(n_0,LT)	LT
d	Access Admin tab	R(tab=Admin,Click), R(User Type=Admin)	(n_0,Admin)	Admin
e	Access to LPD	R(A(Borrower Name,Click)), A(User Type, Loan Status)	(Home,LPD)	LPD
f	Access to LPD	O(R(Loan Number,Click), R(Search Result match Specific Loan,Click))	(SE,LPD)	LPD
i	Login the system	R(User Name,Password)	(w_0,n_0)	n_0
o	Log out the system	R(Logout,Click)	(w_0,n_0)	w_0

Table 6.6: Input Constraint for Home Cluster

Σ	**Actions**	**Constraints**	**Transition**	Ω
a_1	View Next or Previous Week	R(C1(left,right),Click)	(VC,VC)	VC

Table 6.7: Input Constraint for Search Cluster (SE)

Σ	**Actions**	**Constraints**	**Transition**	Ω
b_1	Clear Search, Reset the search page	R(Clear,Click)	(SP,SP)	SP
b_2	Switch to Advanced Search	R(Advanced Search,Click)	(SP,AS)	AS
b_3	Switch to Simple Search	R(Simple Search,Click)	(AS,SP)	SP
b_4	SP's Result of searching match List of loans	R(A(Search Query List),Click)	(SP,SR)	SR
b_5	Excel Export the list result of searching	R(List Search Result,Click)	(SR,EE)	EE
b_6	Dummy edge, reflect access back to SR	R(O(Export, Cancel),Click)	(EE,SR)	SR
b_7	AS's Result of searching match List of loans	R(A(Search Text),Click)	(AS,SR)	SR
f	Result of searching match one loan	O(R(Search List=Specific Loan, Click), R(Search Text=Specific Loan,Click), R(Search Loan Number,Click))	(SR/AS/SP, LPD)	LPD
z	Back to n_0	R(Logout, Click)	(SP/AS,n_0)	n_0

Table 6.8: Annotated Aggregation FSMs for LPD Cluster

Σ	**Actions**	**Constraints**	**Transition**	Ω
p_o	Dummy edge, reflect Back to n_e	R(n_e,Click)	(LP/LIP/CD/ FF/AF/UF/LB/ PC/LN,n_e)	n_e
p_1	Access LP tab	R(tab=LP,Click) Editing: R(User type=LO, Loan status=Open) Viewing: A(User type , Loan status)	(n_e,LIP)	LIP
p_2	Access LIP tab	R(tab=LIP,Click), A(User type,Loan Status)	(n_e,LIP)	LIP
p_3	Access CD tab	R(tab=CD,Click) Editing: R(User type=SU, Loan Status in Approve to Close, Approve with Condition) Viewing: A(User type,Loan status)	(n_e,CD)	CD
p_4	Access FF tab	R(tab=FF,Click) Editing: R(User type=AU, Loan Status in Approve to Close,Funded,Sold) Viewing: A(User type,Loan status)	(n_e,FF)	FF
p_5	Access AF tab	R(tab=AF,Click) Editing: R(User type=AU, Loan Status in Approve to Close, Funded,Sold) Viewing: A(User type,Loan status)	(n_e,AF)	AF
p_6	Access UF tab	R(tab=UF,Click) Editing: R(User type=QC, A(Loan Status)) Viewing: A(User type,Loan status)	(n_e,UF)	UF
p_7	Access LB tab	R(tab=LB,Click) Editing: R(User type in QC,LO, Loan Status in Approve to Close,Approve with Condition) Viewing: A(User type,Loan status)	(n_e,LB)	LB
p_8	Access PC tab	R(tab=PC,Click) Editing: R(User type=CU,Loan Status in (Funded,Sold)) Viewing: A(User type,Loan status)	(n_e,PC)	PC
p_9	A(User type,Loan Status)	R(tab=LN,Click) Editing/Viewing: A(User type,Loan Status)	(n_e,LN)	LN
p_{10}	Add note	R(Add,Click)	(LN,AN)	AN
p_{11}	Back to Notes	R(O(Save,Cancel),Click)	(AN,LN)	LN
p_{12}	Update Loan profile by calling external web service	R((import button,Click), User type in Admin,LO, Loan Status =Open)	(LP,LP)	LP
p_{13}	Revert Funding	R ((Revert button,Click), User type=AU, Loan Status = Funded)	(FF,FF)	FF
z	Back to n_0	R(n_0,Click)	(n_e,n_0)	n_0

Table 6.9: Input Constraint for Closing Documents (CD) FSM

Σ	Actions	Constraints	Transition	Ω
p3_1	Access CD	R((tab= CD, Click), User type=SU, Loan Status in Approve to close,Approve with condition)	(n_{p3},DC)	DC
p3_2	Access CI	R((tab= CI, Click), User type=SU, Loan Status in Approve to close,Approve with condition)	(n_{p3},CI)	CI
p3_1_1	Calling external web service	R((Sync from IDS,Click), User type=SU, Loan Status in Approve to close,Approve with condition)	(DC,DC)	DC
p3_2_1	Calling external web service	R((Sync from IDS,Click), User type=SU, Loan Status in Approve to close,Approve with condition)	(CI,CI)	CI
p3_2_2	Access SI	R((Show Past Instructions ,Click), User type=SU, Loan Status in Approve to close,Approve with condition)	(CI,SI)	SI
p3_2_3	Back to CI	R(Close,Click)	(SI,CI)	CI
p3_o	Back to n_{p3}	R(n_{p3},Click)	(DC/CI,n_{p3})	n_{p3}
p_o	Back to n_e	R(n_e,Click)	(n_{p3},n_e)	n_e

Table 6.10: Test Sequence for Closing Documents (CD) FSM

Edge Sequence	Constraint Sequence	Constraint
(n_{p3},DC) (DC,n_{p3})	p3_1,p3_o	R((tab= CD,Click), User type=SU, Loan Status in Approve to close,Approve with condition)
(n_{p3},CI) (CI,SI)(SI,CI)(CI,n_{p3})	p3_2,p3_2_2,p3_2_3,p3_o	O(R((tab= CI,Click),R((Show Past Instructions ,Click)), User type=SU, Loan Status in Approve to close,Approve with condition)
(n_{p3},CI) (CI,CI) (CI,n_{p3})	p3_2,p3_2_1,p3_2_2,p3_o	R((tab= CI,Click), User type=SU, Loan Status in Approve to close,Approve with condition)

Table 6.11: Test Paths of Aggregate FSMs for Mortgage System

Test #	**Path**
T_0	w_0,n_0,Home,n_0,w_0
T_1	w_0,n_0,Home,LPD,n_0,w_0
T_2	w_0,n_0,SE,n_0,w_0
T_3	w_0,n_0,SE,LPD,n_0,w_0
T_4	w_0,n_0,LT,n_0,w_0
T_5	w_0,n_0,Admin,n_0,w_0

Table 6.12: Test Paths of FSMs for LPD Cluster

Test #	**Path**
T_{LPD1}	n_e,LP,LP,n_e
T_{LPD2}	n_e,LIP,n_e
T_{LPD3}	n_e,CD,n_e
T_{LPD4}	n_e,FF,FF,n_e
T_{LPD5}	n_e,AF,n_e
T_{LPD6}	n_e,UF,n_e
T_{LPD7}	n_e,LB,n_e
T_{LPD8}	n_e,PC,n_e
T_{LPD9}	n_e,LN,AN,LN,n_e

Table 6.13: Test Paths for Home page

Test #	**Path**
T_{Home}	VC,VC

Table 6.14: Test Paths for Search (SE)

Test Case	**Path**
T_{SE_1}	SP,SR,EE,SR
T_{SE_2}	SP,SP,AS,SR
T_{SE_3}	SP,AS,SP,AS,SR

Table 6.15: Test Paths for Closing Documents (CD) Service

Test Case	**Path**
$T_{LPD_{CD1}}$	n_{p3},DC,DC,n_{p3}
$T_{LPD_{CD2}}$	n_{p3},CI,SI,CI,n_{p3}
$T_{LPD_{CD3}}$	n_{p3},CI,CI,n_{p3}

6.2.4 Aggregate Paths to Generate Abstract Tests

The aggregation sequence is based on the input constraint abbreviations of Table 6.9. This means that each input needs to be replaced with the corresponding constraint in Table 6.9. For each node in the mortgage system services aggregate sequence, the test sequences for the lower level FSMs must be substituted. For example, the LPD cluster appears twice in two different abstract test cases of Aggregate FSMs. The LPD cluster is covered by the nine test sequences given in Table 6.12. If the test criterion were to generate sequences with all combinations of paths, substitution would result in 9+9=18 paths.

Table 6.16 summarizes the number of tests through individual cluster FSMs, their length as well as the number of aggregated abstract test paths and their length. Note that it takes approximately 2 months to test the whole system.

As a more detailed example, we will perform the substitution for T_1 and T_3 paths from Table 6.11. This means substituting test sequences in Table 6.12 for LPD in

Table 6.16: Statistics of Tests Size

	Mortgage System	CD Cluster
Number of tests through FSMs	106	3
Size of test suite (Number of nodes)	127	12
number of aggregated tests	266	12
Size of aggregated test suite	3998	169

order to resolve all test paths for (CD). This results in a total of 12 abstract test paths as shown in a Table 6.17 and Table 6.18.

Table 6.17: Paths generated by substitution of T_1

Clusters	**Test No.**	**Test Path**
Home-LPD-CD	T_{CD1}	w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0
Home-LPD-CD	T_{CD2}	w_0,n_0,VC,VC,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0
Home-LPD-CD	T_{CD3}	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0

6.3 Failure Applicability and Mitigation Requirements

Again using the Mortgage system, we will detail the cluster Closing Documents(CD). Table 6.19 lists mitigations for all failure types and gives an example of each.

Corresponding mitigation requirements are summarized in Table 6.20 which also specifies the corresponding mitigation models and associated weaving rules for each failure type. The last column in the table refers to the weaving rule number defined in Table 3.8. Table 6.21 shows the State-Event matrix for the (CD) cluster. It

Table 6.18: Paths generated by substitution of T_3

Clusters	**Test No.**	**Test Path**
SE-LPD-CD	T_{CD4}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD5}	w_0,n_0,SP,SP,AS,SR, n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD6}	w_0,n_0,SP,AS,SP,AS, SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD7}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD8}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD9}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD10}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD11}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0
SE-LPD-CD	T_{CD12}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0

indicates that not all failure types are applicable in all states although some are. For example, f_1 (no network connection) is applicable in all states. f_2 (session is expired) is not applicable only for the entry portal web page (node w_0). Similarly,f_6 (user switches back and forth in the browser) can occur in all states except the entry portal web page w_0. In the (DC) state, all failure types except f_{10} can occur (DC doesn't export data). The last row and column show *dpe* and *dps* which are used to construct the initial population.

Table 6.19: Failure types in Cluster Closing Documents(CD)

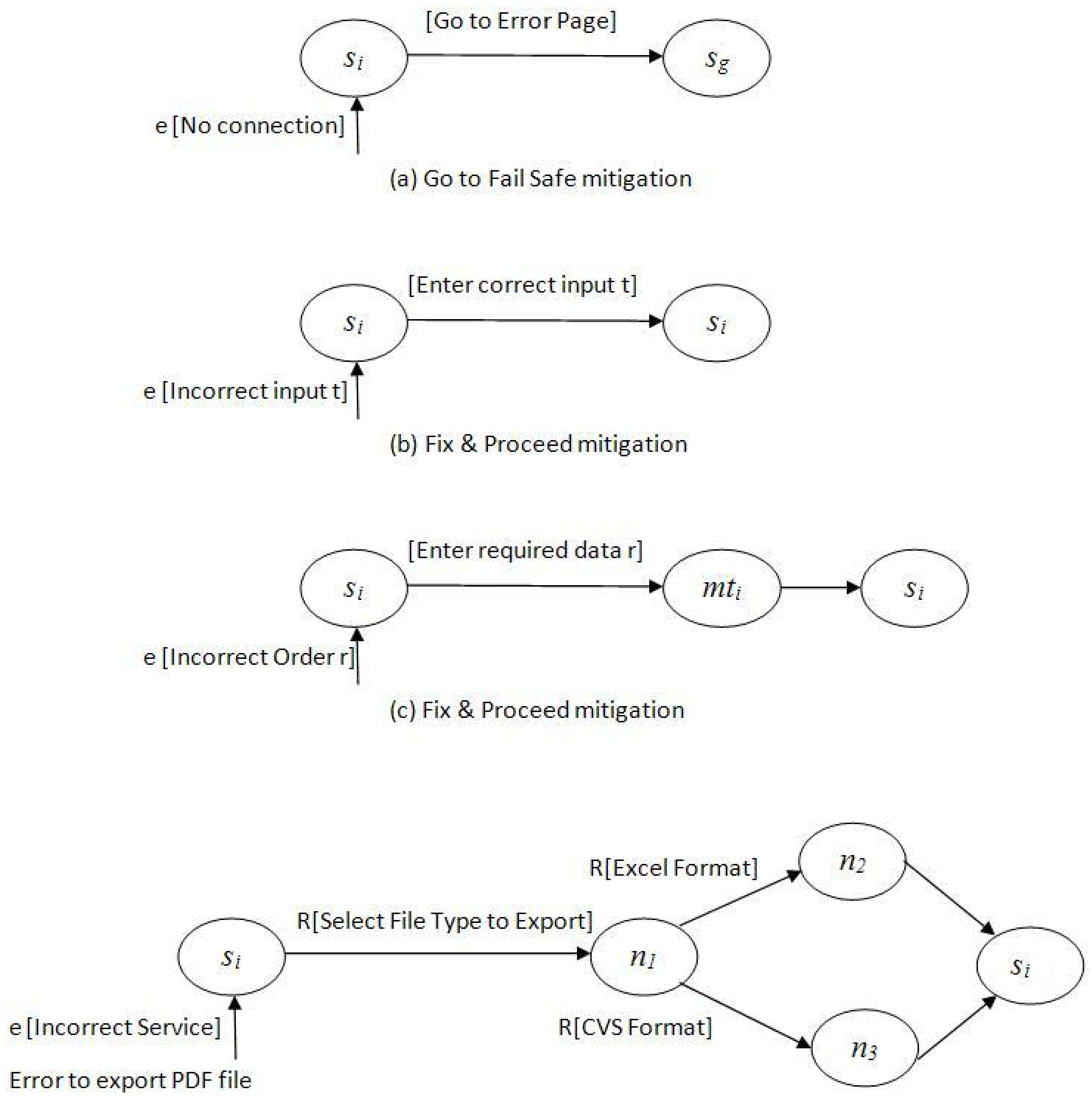
Failure Type	**Mitigation**	**Example**
f_1: unavailability	Go to Fail Safe State	No network connection
f_2: time out	End ALL	Session expire
f_3:Parameter incompatibility	Fix & proceed	Input error (Integer vs. string)
f_4: response error	Rollback	Database server response error
f_5: Misunderstood behaviour	End Activity	Access tab needs specific user role.
f_6: Workflow inconsistency	Ignore	back and forth user browser navigation
f_7: incorrect order	Fix & proceed	Required loan process step
f_8: Browser incompatibility	Retry	Java Script for viewing does not work correctly
f_9: Interface change	Roll Back	External service changes the mapping
f_{10}: incorrect service	Alternative	incorrect service to export data grid into a file

Table 6.20: Mitigation Requirement

MM	Explanation	Model	WR#
MM1	Go to Fail Safe State: keep the system running even if there is no connectivity	see Figure 6.8-a, $MT_1 = \{mt_{11}\}\ where\ mt_{11} = s_i, s_g$ and $s_g = LWP : errorpage$	8
MM2	End All: session expire, so start over from the start node	$MT_2 = \phi$ and $s_b = w_0$,where s_b is the start node	5
MM3	Fix & proceed: parameter incompatibility such as data mismatch	see Figure 6.8-b, $MT_3 = \{mt_{31}\}\ where\ mt_{31} = s_i, s_i$	3
MM4	Rollback: database server error	$MT_4 = \phi$ where $s_b = DC$,and s_b is DC state where trying to save data	6
MM5	End Activity: misunderstood behaviour such as try to access CD cluster without having SU user role	$MT_5 = \phi$ where $s_f = n_{p3}$,and s_f is a menu selection bar of CD cluster	4
MM6	Ignore: workflow inconsistency such as using browser navigation	Internal compensate	7
MM7	Fix & proceed: incorrect order	see Figure 6.8-c , $MT_7 = \{mt_{71}\}\ where\ mt_{71} = s_i, mt_i, s_i$	3
MM8	Retry: Java Script error	$MT_8 = \phi$ where $node(p)^r$	2
MM9	Rollback: Interface change	$MT_9 = \phi$ where $s_b = n_{p3}$	6
MM10	Alternative: incorrect service	see Figure 6.8-d , $MT_{10} = \{mt_1, mt_2\}\ where\ mt_1 = s_i, n_1, n_2, s_i$ and $mt_2 = s_i, n_1, n_3, s_i$	1

Table 6.21: State-Event Matrix for CD Cluster

Behavioral States/ Failure Type (f)	w_0	n_0	VC	n_e	n_{p3}	SP	SR	AS	EE	DC	CI	SI	**dpe**
1	1	1	1	1	1	1	1	1	1	1	1	1	**1.0**
2	0	1	1	1	1	1	1	1	1	1	1	1	**0.9**
3	1	0	0	0	0	1	0	0	0	1	0	0	**0.3**
4	0	0	0	0	0	1	0	1	0	1	0	0	**0.3**
5	0	0	0	0	0	0	0	0	0	1	1	1	**0.3**
6	0	1	1	1	1	1	1	1	1	1	1	1	**0.9**
7	0	0	0	0	0	0	0	0	0	1	0	0	**0.1**
8	0	0	0	0	0	0	0	0	0	1	1	0	**0.2**
9	0	0	0	0	0	0	0	0	0	1	1	0	**0.2**
10	0	0	0	0	0	0	1	0	0	0	1	1	**0.3**
dps	**0.2**	**0.3**	**0.3**	**0.3**	**0.3**	**0.5**	**0.4**	**0.4**	**0.3**	**0.9**	**0.7**	**0.5**	



(a) Go to Fail Safe mitigation

(b) Fix & Proceed mitigation

(c) Fix & Proceed mitigation

(d) Alternative mitigation

Figure 6.8: Mitigation Models.

6.4 Generate Test Requirements

In our case study and from Table 6.16, we know that the CD cluster has a test suite of length 169. Given the 10 failure types, the total search space including infeasible positions is 169*10=1690. However, the number of infeasible pairs in search space is 1052 which is about 62% of the total search space in the CD cluster. Because not all failures are applicable in behavioral states, the feasible search space is 638 which is about 38% of all pairs. By contrast, the Mortgage system overall has a test suite length of 3998 with a total search space of 39980 (including infeasible pairs). The feasible search space is 13034 which is about 32% of all pairs.

To illustrate our approach, we use the GA on the CD cluster using the abstract tests determined in Table 6.17 and 6.18. Table 6.22 shows the behavioral test paths BT for them. It identifies the clusters it covers, the test number, the sequence of nodes and the length of each test path. The last row shows the length of the concatenated test paths.

Table 6.22: Total Length of abstract test paths T_1 and T_3

Clusters	**Test No.**	**Test Path**	Total Length
Home-LPD-CD	T_{CD1}	w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	12
Home-LPD-CD	T_{CD2}	w_0,n_0,VC,VC,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	12
Home-LPD-CD	T_{CD3}	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	13
SE-LPD-CD	T_{CD4}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	14
SE-LPD-CD	T_{CD5}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	14
SE-LPD-CD	T_{CD6}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	15
SE-LPD-CD	T_{CD7}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	15
SE-LPD-CD	T_{CD8}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	15
SE-LPD-CD	T_{CD9}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	16
SE-LPD-CD	T_{CD10}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	14
SE-LPD-CD	T_{CD11}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	14
SE-LPD-CD	T_{CD12}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	15
Total length of $\|I\|$:			**169**

The initial population creates individuals (p, e) as follows:

- Pick each failure type with Max(dps). This results in 10 pairs:
  $(DC, f_1); (CI, f_2); (SP, f_3); (AS, f_4); (SI, f_5); (SR, f_6);$
  $(DC, f_7); (DC, f_8); (DC, f_9); (SI, f_{10})$
- Pick remaining nodes with Max(dpe). This results in 6 pairs:
  $(w_0, f_1); (n_0, f_1); (VC, f_1); (n_e, f_1); (n_{p3}, f_1); (EE, f_1)$

Table 6.23 shows the failure mitigation test paths (FMT) for the first generation. The first column identifies the test, the second column shows the test requirement (p,e) used to construct the test. The next two columns state failure type and node at position p. The fifth column identifies the mitigation model as described in Table 6.20. The last two column states the resulting fail safe mitigation test *fmt* and which behavioral test *bt* was used to create it. Notice that f_{10} has two mitigation test paths for MM10 as explained in Table 6.20 and hence two failure mitigation tests for pair (105,10).

Next, we explore effectiveness of the GA for this case study. We assumed a mitigation defect rate of 5% and seeded the CD subsystem with three defects, similar to what used in [5] as the lower defect density. Since the number of ones in the state-event matrix (see Table 6.21) is 50, $50 \times 5\% = 2.5$. The number of seeded defects after rounded up to 3. It represents a more difficult search problem for the GA that helps to assess the robustness of the approach.

Table 6.24 shows how many pairs and number of generations were needed to detect all defects. The GA generated 47 (p,e) pairs to find all 3 mitigation defects. There are a total of 21 generations.

Table 6.23: Initial FMT for first generation

#	(p,e)	Failure	Node	MM used	FMT	BT used
1	(46,1)	f_1	DC	MM1	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,s_g	T_{CD4}
2	(19,2)	f_2	CI	MM2	w_0,n_0,VC,VC,n_e,n_{p3},CI,w_0,n_0,VC,VC,n_e,n_{p3},CI,CI,n_{p3},n_e, n_0,w_0	T_{CD2}
3	(40,3)	f_3	SP	MM3	w_0,n_0,SP,SP,SR,EE,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	T_{CD4}
4	(56,4)	f_4	AS	MM4	w_0,n_0,SP,SP,AS,AS ,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	T_{CD5}
5	(32,5)	f_5	SI	MM5	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,n_{p3}	T_{CD3}
6	(41,6)	f_6	SR	MM6	T_{CD4}	T_{CD4}
7	(61,7)	f_7	DC	MM7	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},DC,mt_i,DC,DC,n_{p3},n_e,n_0,w_0	T_{CD5}
8	(75,8)	f_8	DC	MM8	w_0,n_0,SP,AS,SP,AS, SR,n_e,n_{p3},DC,DC,DC,n_{p3},n_e,n_0,w_0	T_{CD6}
9	(8,9)	f_9	DC	MM9	w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3}	T_{CD1}
10	(105,10)	f_{10}	SI	MM10	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,n_1,n_2,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD8}
10	(105,10)	f_{10}	SI	MM10	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,n_1,n_3,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD8}
11	(1,1)	f_1	w_0	MM1	w_0,s_g	T_{CD1}
12	(2,1)	f_1	n_0	MM2	w_0,n_0,s_g	T_{CD1}
13	(3,1)	f_1	VC	MM3	w_0,n_0,VC,s_g	T_{CD1}
14	(5,1)	f_1	n_e	MM4	w_0,n_0,VC,VC,n_e,s_g	T_{CD1}
15	(6,1)	f_1	n_{p3}	MM5	w_0,n_0,VC,VC,n_e,n_{p3},s_g	T_{CD1}
16	(42,1)	f_1	EE	MM6	w_0,n_0,SP,SR,EE,s_g	T_{CD4}

From Table 6.24, the GA finds the first defect in the first generation and it takes 7 generations to find the next defect, but it takes only 4 more generations to find the third defect.

Table 6.24: Effectiveness of GA

# of Generation	# of pairs	Defect Found %
initial population	16	33%
2	20	33%
3	21	33%
4	23	33%
5	26	33%
6	29	33%
7	30	33%
8	33	67%
9	36	67%
10	37	67%
11	40	67%
12	41	100%

Table 6.25 shows the FMTs for the last generation. There are 41 pairs which are converted into failure mitigation test paths.

Table 6.26 shows the *fmts* that detected the 3 mitigation defects. Pair (7,1) finds the defect for f_1 (unavailability of network) at state (DC) using the behavioral test case T_{CD1}: w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0. The mitigation of f_1 is to keep the system running even when there is no connectivity by going to the Fail Safe state s_g. This is an error page describing the defect and asking to contact system administration. The mitigation model is $MM1$ as shown in Table 6.20. The failure mitigation test path (***fmt***) is given by: w_0,n_0,VC,VC,n_e,n_{p3},DC,s_g. Similarly, the

Table 6.25: FMT for last generation

#	(p,e)	Failure	Node	MM used	FMT	BT used
1	(33,3)	f_3	SI	MM3	T_{CD3}	T_{CD3}
2	(60,3)	f_3	DC	MM3	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},DC,DC,DC,n_{p3},n_e,n_0,w_0	T_{CD5}
3	(75,4)	f_4	DC	MM4	w_0,n_0,SP,AS,SP,AS, SR,n_e,n_{p3},DC,DC,DC,n_{p3},n_e,n_0,w_0	T_{CD6}
4	(165,4)	f_4	CI	MM4	T_{CD12}	T_{CD12}
5	(89,5)	f_5	CI	MM5	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,n_{p3}	T_{CD7}
6	(133,5)	f_5	n_e	MM5	T_{CD10}	T_{CD10}
7	(56,6)	f_6	AS	MM6	T_{CD5}	T_{CD5}
8	(135,6)	f_6	CI	MM6	T_{CD10}	T_{CD10}
9	(7,6)	f_6	DC	MM6	T_{CD1}	T_{CD1}
10	(105,6)	f_6	SI	MM6	T_{CD8}	T_{CD8}
11	(46,7)	f_7	DC	MM7	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,mti,DC,n_{p3},n_e,n_0,w_0	T_{CD4}
12	(90,7)	f_7	SI	MM7	T_{CD7}	T_{CD7}
13	(120,8)	f_8	CI	MM8	w_0,n_0,SP,AS,SP,AS, SR,n_e,n_{p3},CI,CI,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD9}
14	(19,9)	f_9	CI	MM9	w_0,n_0,VC,VC,n_e,n_{p3},CI,n_{p3}	T_{CD2}
15	(129,9)	f_9	SP	MM9	T_{CD10}	T_{CD10}
16	(148,9)	f_9	n_{p3}	MM9	T_{CD11}	T_{CD11}
17	(81,10)	f_{10}	w_0	MM10	T_{CD7}	T_{CD7}
18	(32,10)	f_{10}	SI	MM10	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,n_1, n_2,CI,n_{p3},n_e,n_0,w_0	T_{CD3}
19	(32,10)	f_{10}	SI	MM10	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,n_1, n_3,CI,n_{p3},n_e,n_0,w_0	T_{CD3}
20	(159,1)	f_1	SP	MM1	w_0,n_0,SP,AS,SP,s_g	T_{CD12}
21	(47,1)	f_1	DC	MM1	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,DC,s_g	T_{CD4}
22	(56,1)	f_1	AS	MM1	w_0,n_0,SP,SP,AS,s_g	T_{CD5}
23	(107,1)	f_1	n_{p3}	MM1	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},s_g	T_{CD8}
24	(15,1)	f_1	VC	MM1	w_0,n_0,VC,s_g	T_{CD2}
25	(7,1)	f_1	DC	MM1	w_0,n_0,VC,VC,n_e,n_{p3},DC,s_g	T_{CD1}
26	(5,1)	f_1	n_e	MM1	w_0,n_0,VC,VC,n_e,s_g	T_{CD1}
27	(38,1)	f_1	w_0	MM1	w_0,s_g	T_{CD4}
28	(85,1)	f_1	EE	MM1	w_0,n_0,SP,SR,EE,s_g	T_{CD7}
29	(130,1)	f_1	SR	MM1	w_0,n_0,SP,SR,s_g	T_{CD10}
30	(119,1)	f_1	n_{p3}	MM1	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},s_g	T_{CD9}
31	(153,1)	f_1	n_0	MM1	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,s_g	T_{CD11}
32	(136,2)	f_1	CI	MM2	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,s_g	T_{CD10}
33	(158,2)	f_1	AS	MM2	w_0,n_0,SP,AS,s_g	T_{CD12}
34	(6,2)	f_1	n_{p3}	MM2	w_0,n_0,VC,VC,n_e,n_{p3},s_g	T_{CD1}
35	(7,2)	f_2	DC	MM2	w_0,n_0,VC,VC,n_e,n_{p3},DC,w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3}, n_e,n_0,w_0	T_{CD1}
36	(32,2)	f_2	SI	MM2	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,CI,n_{p3}, n_e,n_0,w_0	T_{CD3}
37	(157,2)	f_2	SP	MM2	w_0,n_0,SP,w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	T_{CD12}
38	(101,2)	f_2	SR	MM2	w_0,n_0,SP,SP,AS,SR,w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3}, n_e,n_0,w_0	T_{CD8}
39	(131,2)	f_2	EE	MM2	w_0,n_0,SP,SR,EE,w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,n_{p3},n_e, n_0,w_0	T_{CD10}
40	(152,2)	f_2	n_e	MM2	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,w_0,n_0,SP,SP,AS,SR, n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	T_{CD11}
41	(94,2)	f_2	n_0	MM2	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0,n_0,SP,SR, EE,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD7}

mitigation defect for f_2 is found by using pair (32,2) (state SI in test path T_{CD3}) and constructing the *fmt* using the weaving rule "End All" and starting over. Lastly, the mitigation defect for f_3 is found using pair (60,3) (state DC in test path T_{CD5}). The mitigation is constructed by repeating the edge that showed the failure and then proceeding. In our case, it is the edge (DC,n_{p3}).

Table 6.26: fmt_i that found defects

FMT	Failure	State	BT used	GA pairs	MM used	Explanation
fmt_1	f_1	DC	T_{CD1}	(7,1)	MM1	w_0,n_0,VC,VC,n_e,n_{p3},DC, s_g
fmt_2	f_2	SI	T_{CD3}	(32,2)	MM2	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,w_0,n_0,VC,VC,n_e,n_{p3},CI,SI, CI,n_{p3},n_e,n_0,w_0
fmt_3	f_3	DC	T_{CD5}	(60,3)	MM3	w_0,n_0,SP,SP,AS,SR, n_e ,n_{p3},DC,DC, DC,n_{p3},n_e,n_0,w_0

This study applied our test generation method to the CD sub system, a subset of the full mortgage handling system. We apply the GA to the whole system. We seed the same percentage of defects. This results in 25 defects. Table 6.27 shows the total number of generations needed and the cumulative number of pairs generated to expose all mitigation defects. The GA needs 42 generations to find all defects. There are 485 pairs resulting in 504 failure mitigation tests (*FMTs*) for the whole system. As it did for the Closing Documents (DC) subsystem, the GA finds all mitigation defects.

Table 6.27: The results of run GA on the whole system

# of Generation	# of pairs	Defect Found %
initial population	170	0%
2	174	0%
3	175	0%
4	181	0%
5	184	4%
6	184	4%
7	186	4%
8	192	4%
9	195	4%
10	198	4%
11	201	4%
12	210	8%
13	216	8%
14	237	8%
15	238	8%
16	240	8%
17	269	16%
18	314	16%
19	324	27%
20	325	33%
21	361	33%
22	370	33%
23	373	42%
24	378	42%
25	391	46%
26	407	53%
27	410	53%
28	418	58%
29	430	67%
30	442	72%
31	453	72%
32	457	75%
33	457	77%
34	463	77%
35	465	77%
36	469	80%
37	472	85%
38	476	93%
39	479	93%
40	482	93%
41	484	95%
42	485	100%

6.5 Comparison of Effort GA vs. Exhaustive Search

At several points, we claimed that exhaustive search, i.e converting all feasible pairs to executable tests, executing, and validating them is prohibitive. To investigate this claim, we measured the time it took to translate a set of test requirements into executable tests, executing and validating the results. We then computed average effort per node in the test path. We computed length of failure mitigation test suite for exhaustive search for both the CD subsystem and the whole system and multiplied with the average effort per node to a arrive at an effort estimate for exhaustive search.

Table 6.28 shows the results. For the CD subsystem, the test requirements using GA requires 352 pairs, the total length of all failure mitigation test 3565. Estimated average test effort is about 6 work days (note work day=8 hours) while using exhaustive search requires more than 11 work days of testing. However, for the whole system, the differences are much more drastic: more than 155 work days for GA vs. about 525 work days for exhaustive search more than three times as long given that both find all mitigation defects, the choice is obvious.

Table 6.28: Time Budget Comparison between GA vs. Exhaustive search

The Sub system (CD)					
Approach	Test Requirements (# of pairs)	*length(FMT)*	Time Estimation (min)	Total Hours	Work Days
GA	352	3565	2674	45	5.6
Exhaustive search	638	7195	5396	90	11.25
The Whole system					
GA	8276	99312	74484	1241	155.23
Exhaustive search	27986	335920	251940	4199	524.88

Table 6.29 shows the number of nodes, transitions, the total number of behavioral test paths BT, and the the total length of concatenated I. Note that the effort estimates reported in Table 6.28 refer to mitigation testing only and do not include testing primary functionality.

Table 6.29: The size of BT for CD vs. Mortgage system

	# of nodes	# of transitions	size of BT	Length (I)
CD subsystem	12	9	12	169
Mortgage System	127	224	266	3998

Chapter 7

Regression Testing Process

Based on the changes to the artifact used in the test generation approach in chapter 3, we classify tests as retestable, reusable, or obsolete. Changes to our various models (behavioral model BM, mitigation models MM, weaving rules WR, type of failures F, or state-event matrix SE) having varying impact on which tests in the failure mitigation test suite FMT are retestable, reusable or obsolete. To make generating a regression test suite more efficient, we provide an approach for partial regeneration of obsolete test cases that takes advantage of the fact that our test generation approach has phases. We determine at which point in the generation process a test path becomes obsolete. Only the steps from that point on need to be repeated. Figure 7.1 shows the regression testing process and references the sections that describe the changes. Table 7.1 lists the variables and a short explanation how they are used in the regression testing process.

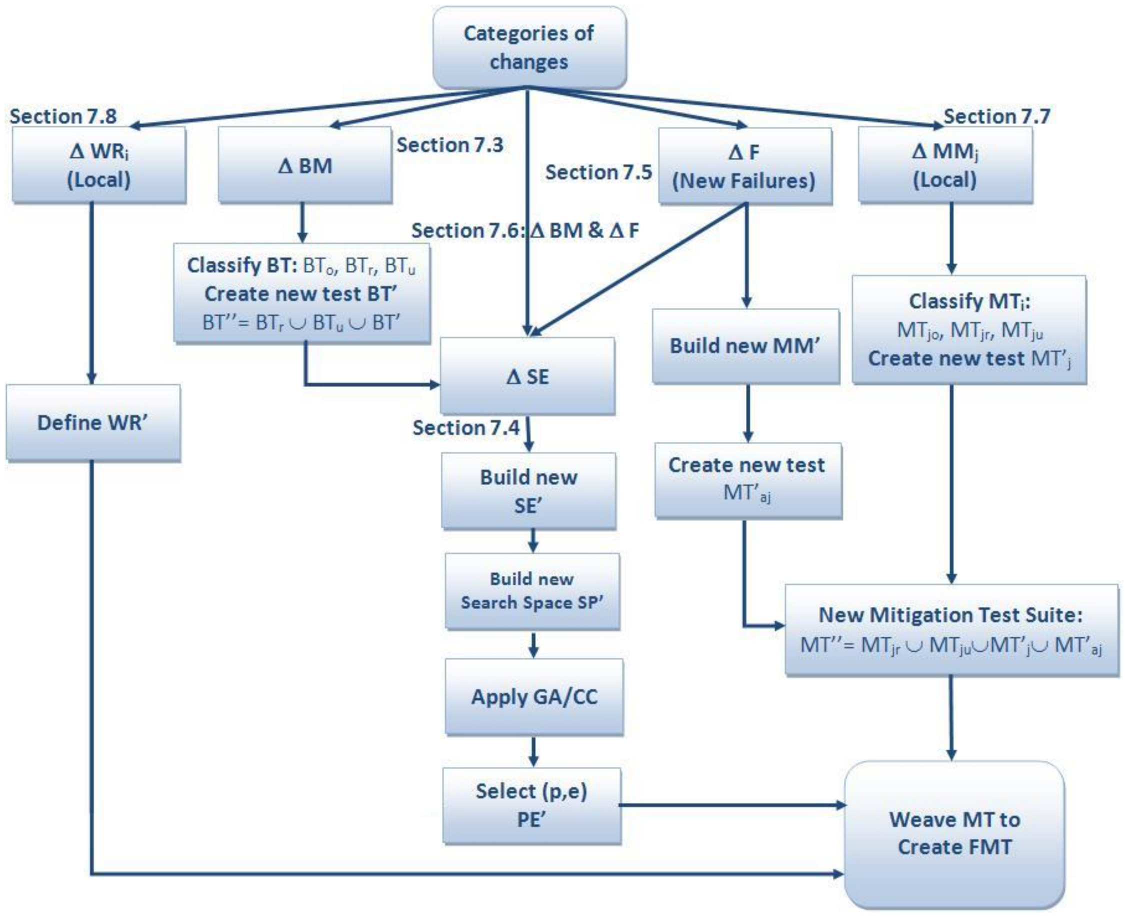


Figure 7.1: Regression Testing Process

Table 7.1: Variables description used in regression testing process

Variable	Explanation
BM	behavioral model
BM'	modified behavioral model
BT	behavioral test suite

Continued on next page

Table 7.1 – *Continued from previous page*

Variable	Explanation
BT_o	obsolete behavioral tests
BT_r	retestable behavioral tests
BT_u	reusable behavioral tests
BT_{se}	behavioral tests that visits states in S_{se}
BT'	additional behavioral tests needed to satisfy coverage requirements for BM'
BT''	behavioral test suite for BM' where $BT'' = BT_r \cup BT'$
BT_a	behavioral tests where $BT_a = BT' \cup BT_r \cup BT_u$
S_{se}	the subset of states whose applicability changed due to the changes to SE
N	set of behavioral nodes
E	set of behavioral edges
N_o	set of deleted or modified nodes
O_N	set of obsolete test paths due to node changes

Continued on next page

Table 7.1 – *Continued from previous page*

Variable	Explanation
E_o	set of deleted or modified edges
O_E	set of obsolete test paths due to edge changes
N_r	set of retestable nodes
SE	state-event matrix before change
SE'	modified state-event matrix due to changes to BM, F
SE_a	the new state-event matrix due to added failure types F_a
I	concatenation of behavioral tests BT
I'	concatenation of behavioral tests BT''
I_{se}	concatenation of of impacted behavioral tests due to the changes to SE
$Length(I')$	is the length of the concatenation of new behavioral tests BT'.
$Length(I_{se})$	is the length of the concatenation of impacted tests due to changes to SE.
SP	search space before change

Continued on next page

Table 7.1 – *Continued from previous page*

Variable	Explanation
SP'	modified search space
SP_a	the new search space due to added failure types F_a
F	set of Failure types
p	Position of Failure
e	Failure Type
(p, e)	where ($1 \leq p \leq \|I\|$) and ($1 \leq e \leq \|F\|$)
PE	set of selected (p, e) pairs before change
PE'	new set of selected (p, e) pairs
F_d	set of deleted failures $F_d = \{f_{d1}, \cdots, f_{dm}\}$ when m is the number of deleted failure types
F_a	set of added failures $F_a = \{f_{a1}, \cdots, f_{an}\}$ when n is the number of new failure types
F_{inf}	set of failure types that become infeasible due to changes to SE
F'	the set of impacted failures due to deleted failures

Continued on next page

Table 7.1 – *Continued from previous page*

Variable	Explanation
F_{se}	the set of failures whose applicability changed due to the changes to SE
MM_j	mitigation model before change where $1 \leq j \leq k$ and $k = \|F\|$ is the number of failure types
MM'_j	modified mitigation model for failure type j
MM_{aj}	added mitigation model for added failures F_a
MM_d	the set of mitigation model for deleted failures F_d
MT_j	mitigation test cases before change
MT'_j	new mitigation tests for MM'_j
MT_{j_o}	obsolete mitigation tests of failure type j
MT_{j_r}	retestable mitigation tests of failure type j
MT_{j_u}	reusable mitigation tests of failure type j
MT_{aj}	mitigation tests for MM_{aj}
MT''_j	full mitigation test suite for MM'_j where $MT''_j = MT_{j_r} \cup MT'_j$

Continued on next page

Table 7.1 – *Continued from previous page*

Variable	Explanation
mod_{MM}	is the changed mitigation model where $j \in mod = \{j \mid 1 \leq j \leq \lvert F \rvert \wedge MM'_j \neq MM_j\}$
$PE_{MM'}$	set of selected (p, e) pairs where $e \in mod_{MM}$
FMT	failure mitigation tests before change
FMT_r	failure mitigation tests based on the retestable test cases BT_r
FMT'	failure mitigation test derived from BT'
FMT_{F_d}	the removable failure mitigation tests of deleted failure types F_d
FMT_{F_a}	the failure mitigation tests based on added new failure types F_a
FMT''	the full new failure mitigation test suite
FMT'_{MM}	the new failure mitigation tests derived from the changes to MM'_j.
FMT_{rt}	the failure mitigation tests that are derived from MT_{j_r}.
FMT_{MMu}	the failure mitigation tests that are derived from MT_{j_u}.
FMT_o	the failure mitigation tests that are derived from MT_{j_o}.
E_I	the set of failure types that are impacted by MM'_j.

Continued on next page

Table 7.1 – *Continued from previous page*

Variable	Explanation
PE_I	the set of pairs that are impacted by MM'_j.
FMT_I	the set of failure mitigation tests that are impacted by MM'_j.
FMT_c	the failure mitigation tests impacted due to added failure types F_a.
FMT_o	the obsolete failure mitigation tests due to the changes to SE.
WR	weaving rules for MM_j where $j = 1, .., \|F\|$
WR'	modified weaving rules for MM_j where $j = 1, \cdots, \|F\|$
WR_a	the new weaving rules for new mitigation model MM_{aj}
WR_d	the set of weaving rules for deleted failures F_d
mod_{WR}	are the failure types for which weaving rules changed
PE_{WR}	the set of pairs that are impacted by WR'.

The process steps to build a regression test suite outlined in Figure 7.1 is based on the types of changes to various artifacts:

1. Changes in the behavioral model (BM):

   We classify the behavioral tests (BT) as obsolete (BT_o), retestable (BT_r), or reusable (BT_u). We determine if any parts of the new behavioral model (BM') have not been tested and generate new tests (BT') for them. We construct the new behavioral test suite (BT'') resulting from (BT_r), and (BT'). If there are no changes to F, we build a new state-event matrix (SE') and a modified search space (SP'), then select (p, e) pairs (PE')[1]. In case of added failure types, we add the new failure types to the new state-event matrix (SE') and to the new search space (SP'). If mitigation models are not changed, we use the mitigation tests (MT) from the existing mitigation models (MM) that are built for each failure type. Then we generate the new failure mitigation tests (FMT'') by weaving mitigation tests (MT) into (BT) using weaving rules (WR). In case the mitigation models are changed (MM'), we classify the mitigation tests (MT) as obsolete, retestable, or reusable. We generate new mitigation tests (MT') if there are any parts of new mitigation models (MM') that have not been tested. Next, we weave the new mitigation test suite (MT'') to generate new failure mitigation tests (FMT''). In case of the changes to the weaving rules (WR), we apply the new weaving rules to the mitigation tests (MT) if there are no changes to mitigation models (MM); otherwise, we use the new rules (WR') with the new mitigation test suite (MT'') to generate new failure mitigation tests (FMT'').

2. Changes in the state-event matrix (SE):

   If there are no changes to BM or F, the changes to SE requires building a new search space (SP') as the changes in the SE matrix affect the applicability of the node-failure relation. We select (p, e) pairs. If mitigation models are not changed, we then weave the mitigation tests (MT) to create the new failure mitigation tests (FMT'') using weaving rules (WR).

   If there are changes to BM as well we follow the same steps that are described

[1] If the new search space is sufficiently large, we use GA [5] to construct PE, otherwise coverage criterion (CC)[6] are the best choice to construct PE.

in step one for classifying (BT), and generate the new behavioral test suite (BT''). We add the new nodes if they exist as new columns in the new state-event matrix (SE'). We build the new search space (SP') based on the changes to SE and BM.

If there are added failure types, we simply extend the SE to include the new failure types and build the new search space (SP') based on all changes. Next, we apply GA to select (p, e) pairs and weave them to generate new failure mitigation tests (FMT''). If there are any changes to the mitigation models (MM) or weaving rules (WR), we also apply step 4 and 5 below.

3. Changes in failure types (F):

   Adding new failures requires building a new state-event matrix (SE') by adding a new row to the matrix for each new failure type. Then, we construct a new search space (SP') and select (p, e) pairs for the new failure types only. We build new mitigation models (MM') for the new failure types, and create new mitigation tests (MT_a).

   If there are no changes to BM and WR, we generate the new failure mitigation tests (FMT'') by weaving the new mitigation tests (MT_a) into behavioral tests (BT). Otherwise, we use the new behavioral test suite BT'' resulting from the new behavioral model BM'. Next, we weave BT'' using both the new mitigation tests (MT_a) for the new failure types and the existing (MT) for the old failure types to generate new failure mitigation tests (FMT''). In case of changes to the mitigation models (MM) or weaving rules (WR), we follow steps 4 and 5 below. If failure types are deleted, we simply delete all the associated mitigation models, weaving rules and failure mitigation tests for the deleted failure types.

4. Changes in mitigation models (MM):

   We only need to consider changes to individual mitigation models since all other changes to various artifacts such as changes to BM, SE, or F are already covered. Assume changes are to mitigation model MM_j resulting in MM'_j. Similar to the changes in BM, we classify the mitigation tests MT_j for the mitigation model MM_j into obsolete (MT_{j_o}), retestable (MT_{j_r}), or reusable (MT_{j_u}). We generate new mitigation tests (MT'_j) for edges that are not covered in the modified mitigation model (MM'_j), so the new mitigation test suite is $MT''_j = MT_{j_r} \cup MT'_j$. This process applies to each mitigation model that has changed. Next, we weave MT''_e into position (p, e) to build the new failure mitigation tests (FMT''). If there are added failure types, we include the new mitigation tests (MT_a) derived from the new mitigation models (MM_a) that have been built for the new failure types. As a result, the new mitigation test suite is $MT'' = MT_{j_r} \cup MT'_j \cup MT'_a$. In case of changes to (WR), we apply the new weaving rules to the new mitigation test suite (MT'') to generate new failure mitigation tests (FMT'').

5. Changes in weaving rules (WR):

   Since all other changes to BM, SE, or F are already dealt with, we need only consider changes to WR. Let WR'_e be the modified weaving rule for failure type f_e. We reweave all mitigation tests (MT_e) for failure type (e) at all positions p ($1 \leq p \leq |I|, (p, e) \in PE$) using the new weaving rule (WR'_e) for failure type e to generate the new failure mitigation tests (FMT''). As before, if there are any changes to (BM) and (MM), they have been dealt with in priors steps.

6. We execute FMT''.

We use a simple example to illustrate the process in each step of our regression testing framework. It is introduced in subsection 7.1. The remaining subsections explain this process of building a regression test suite in more detail and use the example to illustrate each step.

7.1 Example

To illustrate our approach, we extend the example from section 3.2.1. Figure 7.2 shows three FSMs and two levels of hierarchy. This is our behavioral model BM. It also shows $FSM1$ and $FSM2$ before and after changes resulting in $FSM1'$ and $FSM2'$. Table 3.3 and Table 3.4 in section 3.2.1 shows paths through each FSM that achieve edge coverage, including derivation rules and the test path lengths.

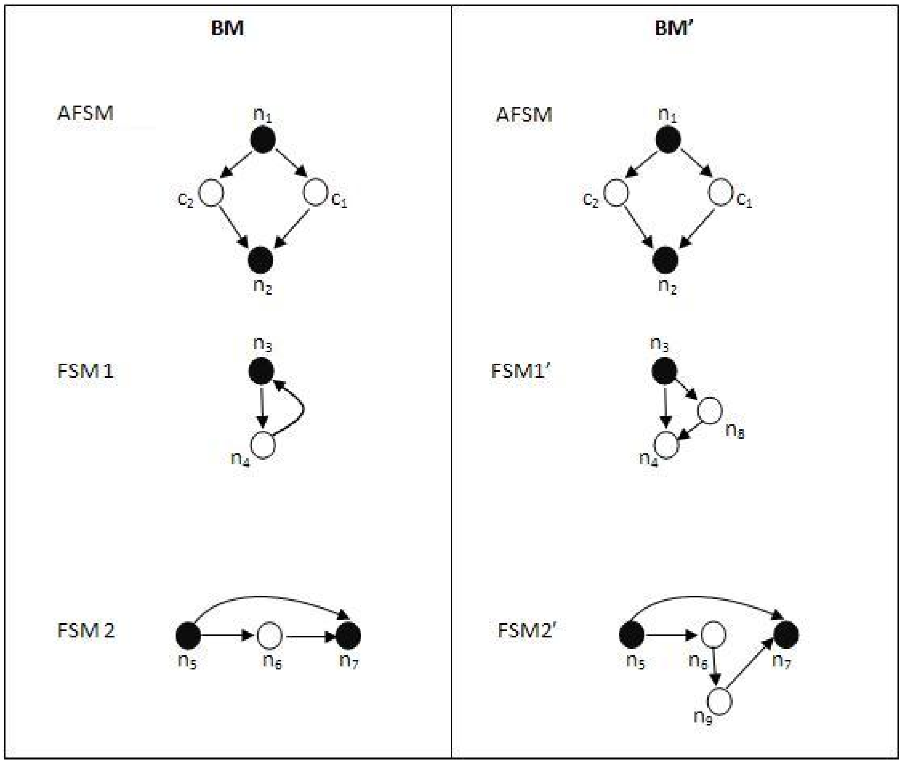


Figure 7.2: Behavioral Models BM, BM'

In section 3.3, The SE is defined as shown in Table 3.5. In section 3.5.1, Table 3.7 shows the potential search space of $length(I) \times F$. In section 3.6, Table 3.6 shows the corresponding mitigation models and associated weaving rules for each failure type and Table 3.9 shows the selected pairs PE and resulting FMT.

The next section describes the regression test generation in detail. We define changes to the models in the example as we formalize the algorithms in each step to illustrate how they work.

7.2 Changes to the Behavioral Model (BM)

The types of changes in the FSMWeb behavioral model are as follows [7]:

1. node change: delete, modify (i. e. change the node type from LWP to cluster or vice versa). Node addition is covered by edge changes.
2. edge change: modify in/out edge (i. e. modify the edge's input-action constraint)[2], delete in/out edge, add in/out edge with or without new node

Based on existing work by Andrews et al. [7], The behavioral test suite (BT) will be classified as obsolete (BT_o), retestable (BT_r), or reusable (BT_u)

$$\text{where } BT = BT_o \cup BT_u \cup BT_r$$

Figure 7.2 showed the changes to the behavioral model. They are:

- For FSM1: we delete edge (n_4, n_3), add node n_8 and edges (n_3, n_8),(n_8, n_4).
- For FSM2: we delete edge (n_6, n_7), add node n_9 and edges (n_6, n_9),(n_9, n_7).

The following subsections describe the steps based on changes to BM.

[2]Modifying the source or target node of an edge, that is, redirecting the edge, is covered by edge deletion and subsequent edge addition

7.2.1 Classify BT into obsolete, reusable and retestable behavioral tests

The test classification rules are as follows:

- **Obsolete Tests Paths** (BT_o)

  Andrews et al. [7] define a set of rules for defining obsolete test paths based on type of change: Node deletion affects all paths that include the deleted node, rendering it obsolete. Node modifications change the type of node from LWP to a cluster node or vice versa. This type of modification affects the paths as the node needs to be replaced by another node or a sequence of nodes. Thus, both node deletions and node modifications render test paths obsolete that tour these nodes: let the set of behavioral test paths be $BT = \{bt_1, \dots , bt_l\}$. Let N_o ={n| n $\in$ N; n is deleted or modified } where N is the set of behavioral nodes, then the set of obsolete test paths due to node changes is given by O_N =$\{bt_i|$ $\exists$ n $\in$ N_o : bt_i visits n$\}$, where bt_i is a behavioral test path. For our example, $N_o = \phi$, and $O_N = \phi$.

  Edge deletion makes any test paths that tour the edge obsolete. Any edge modification that requires changes in the inputs, guards, actions, outputs, and messages associated with it, will also make test paths obsolete that visit the modified edge.

  Let E_o ={e| e $\in$ E; e is deleted or modified} where E is the set of behavioral edges, then the set of obsolete test paths due to edge changes is given by O_E =$\{bt_i \mid t_a$ tours e $\in E_o$ }. For our example, $E_o = \{(n_4, n_3), (n_6, n_7)\}$, and $O_E = \{bt_2, bt_3\}$. Thus, the set of obsolete behavioral test paths is given by: $BT_O = O_N \cup O_E$. Hence $BT_o = \phi \cup O_E = \{bt_2, bt_3\}$.

- **Retestable Test Paths (BT_r)**

  In [7], retestable tests are defined as those that are still valid and test portions of the application and visit part of the FSMWeb model that are affected by the changes. This can be determined in different ways. For example, [7] considers any node n that is one edge away from a modified or deleted node as impacted by the change, except for the source and sink nodes of the AFSM. Using this definition:

  $N_{r_{node}}$ $=\{n|\ \exists$ e : $(n, \hat{n})$ or $(\hat{n}, n)$; $\hat{n} \in N_o$;$n \neq n_{source}$; $n \neq n_{sink}$ $\}$. Since in our example $N_o = \phi$, $N_{r_{node}} = \phi$ as well. When edges are changed, the set of retestable edges depends on the type of change. When edges are deleted or modified, we assume that the starting and ending nodes of the changed edges are potentially impacted and hence nonobsolete tests that visit these nodes are retestable (except for the source and sink nodes of the model):

  $N_{r_{edm}} = \{n|\hat{e} \in E_o; \hat{e} = (n_i, n)\ or\ \ \hat{e} = (n, n_i); n \neq n_{source}; n \neq n_{sink}\}$. In the example $N_{r_{edm}} = \{n_3, n_4, n_6, n_7\}$.

  Similarly, when we add new edges, existing nodes at which these new edges start or end are considered potentially affected by the modification and hence non-obsolete tests that visit these nodes are retestable (Except for the source and sink nodes of the model):

  Let E be the set of edges in BM. Let E' is the set of added edges in BM' Then $E' \setminus E$ is the set of added edges.

  $N_{r_{ea}} = \{n|n \in N_j : \hat{e} = (n, n_i)$ or $\hat{e} = (n_i, n); \hat{e} \in E' \setminus E, n_i \in N'; n \neq n_{source}; n \neq n_{sink}\}$

  In the example, $E' \setminus E = \{(n_3, n_8), (n_8, n_4), (n_6, n_9), (n_9, n_7)\}$

  $\implies N_{r_{ea}} = \{n_3, n_4, n_6, n_7\}$. This happens to be the same as $N_{r_{edm}}$.

  The set of retestable nodes is then given by $N_r = N_{r_{node}} \cup N_{r_{edm}} \cup N_{r_{ea}}$ and the set of retestable abstract test paths is

$BT_r = \{bt_i \mid bt_i \text{ visits } n \in N_r, \, bt_i \in BT\} \setminus BT_O$

In the example, $N_r = \{n_3, n_4, n_6, n_7\}$

$\implies BT_r = \{bt_1, bt_2, bt_3\} \setminus \{bt_2, bt_3\} = \{bt_1\}$ In our example, the only non-obsolete test path bt_1 is also retestable.

- **Reusable Test Paths (BT_u)**

  Those tests are neither obsolete nor retestable. $BT_u = BT \setminus (BT_o \cup BT_r)$. The example has no reusable test paths $BT_u = \phi$.

- **New Test Paths (BT')**

  New test cases need to be designed whenever current test cases do not meet coverage requirements for BM'. This happens when edges or nodes are added. Obsolete test cases can also cause gaps in coverage that need to be addressed with new tests. In the example, the following edges in the modified model BM': $\{(n_3, n_8), (n_8, n_4), (n_6, n_9), (n_9, n_7)\}$ are not covered. Thus, new test paths are generated: $bt'_1 = \{n_1, n_3, n_4, n_2\}$, $bt'_2 = \{n_1, n_3, n_8, n_4, n_2\}$, and $bt'_3 = \{n_1, n_5, n_6, n_9, n_7, n_2\}$. As a result, the new test paths $BT' = \{bt'_1, bt'_2, bt'_3\}$.

The new test suite will be as follows: $BT'' = BT_r \cup BT'$. In the example, Table 7.2 shows the classification of behavioral test paths BT and the new test path BT' for the modified model. So, the new test suite will be as follows:
$BT'' = BT_r \cup BT' = \{bt_1, bt'_1, bt'_2, bt'_3\}$.

7.2.2 Build New SE' matrix

Any change in the BM such as adding, modifying, or deleting node requires to rebuild the state event(SE) matrix. When adding a new node, we add a new column to SE, and similarly when deleting a state, we delete a column from SE. We also

Table 7.2: Classification of BT after the changes

BT	Path	Classification
bt_1	n_1, n_5, n_7, n_2	Retestable
bt_2	n_1, n_5, n_6, n_7, n_2	Obsolete
bt_3	n_1, n_3, n_4, n_3, n_2	Obsolete
BT'	Path	Classification
bt'_1	n_1, n_3, n_4, n_2	New
bt'_2	n_1, n_3, n_8, n_4, n_2	New
bt'_3	$n_1, n_5, n_6, n_9, n_7, n_2$	New

need to recalculate dpe and dps.

The new state-event matrix SE' is shown in Table 7.3. It is the same as the original SE except for the added column for the new states n_8, and n_9.

Table 7.3: New State-Event Matrix SE'

Behavioral States/ Failure Type (f)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	**dpe**
1	1	0	0	1	1	0	1	0	0	**0.44**
2	0	0	1	0	1	1	0	0	1	**0.44**
3	0	1	0	1	1	1	1	1	0	**0.67**
4	0	1	0	0	0	0	1	0	0	**0.22**
dps	**0.25**	**0.50**	**0.25**	**0.50**	**0.75**	**0.50**	**0.75**	**0.25**	**0.25**	

7.2.3 Build New Search Space SP' and FMT"

The new search space used only consider the new tests BT'. This is because the failure mitigation tests derived from retestable behavioral tests BT_r are still valid

and can be reused. Let FMT_r be the failure mitigation tests that were built from tests in BT_r. Thus, $FMT_r \subseteq FMT$.
Additionally, we need to determine failure mitigation test for the new tests BT'. This requires constructing a new search space SP' for BT'. Let $BT' = \{bt'_1, bt'_2, ..., bt'_z\}$ where z is number new of test cases. Concatenating the new test suite such that:

$$I' = (bt'_1 \circ bt'_2 \circ ... \circ bt'_z)$$

The second dimension in the search space is given by the number of failure types $E = |F|$. The new search space SP' is defined as:

$$SP' = \{(p, e)|1 \leq p \leq length(I'), 1 \leq e \leq E, se'_{(node(p),e)} = 1\}$$

We use *SP'* to select new pairs PE' where PE'=$\{(p, e)|(p, e) \in SP' \wedge$ GA or CC selected $(p, e)\}$. We assume that any changes in mitigation models have been dealt with and the new mitigation test suite MT' has been built or that $MT' = MT$ if no changes occurred. We generate failure mitigation tests FMT' with PE'. As a result, the failure mitigation regression test suite consists of the failure mitigation tests FMT_r based on the retestable tests BT_r plus the tests generated via PE' (i.e FMT'). The failure mitigation regression test suite is:

$$FMT'' = FMT_r \cup FMT'$$

In the example, only bt_1 is retestable, and according to Table 3.9, only number fmt_1 and fmt_9 are constructed based on bt_1, thus $FMT_r = \{fmt_1, fmt_9\}$. Table 7.2 shows the new test paths $BT' = \{bt'_1, bt'_2, bt'_3\}$.

We concatenate the tests in BT': $I' = (bt'_1 \circ bt'_2 \circ bt'_3)$
Then, $I' = n_1, n_3, n_5, n_2, n_1, n_3, n_6, n_5, n_2, n_1, n_5, n_6, n_8, n_7, n_2$ where $length(I') = 15$. The new search space: SP'=$\{(p, e)|1 \leq p \leq 15, 1 \leq e \leq 4, se'_{(node(p),e)} = 1\}$ (see table 7.4).

Table 7.4: New Search Space SP'

	bt'_1				bt'_2					bt'_3					
I'/F	n_1	n_3	n_4	n_2	n_1	n_3	n_8	n_4	n_2	n_1	n_5	n_6	n_9	n_7	n_2
f_1	1	0	1	0	1	0	0	1	0	1	1	0	0	1	0
f_2	0	1	0	0	0	1	0	0	0	0	1	1	1	0	0
f_3	0	0	1	1	0	0	1	1	1	0	1	1	0	1	1
f_4	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1

We use *SP'* to select new pairs PE'. Next, we generate new failure mitigation tests FMT'' using the existing weaving rules that are defined in Table 3.6. Table 7.5 shows the selected (p, e) pairs and resulting FMT'' for the modified model BM'. The first column in Table 7.5 numbers each failure mitigation test ($fmt_1 - fmt_{12}$). The second column lists each (p, e) pair in PE'. The third column refers to the failure type whose mitigation is tested. The fourth column states the node at position p. The fifth column identifies the new behavioral test used in constructing fmt_i ($i = 1, \cdots, 12$). The sixth column identifies which mitigation model is used as described in Table 3.6. The seventh column lists which mt_{ij} is used as described in Table 3.6. The last column shows the failure mitigation tests. Table 7.5 shows the new failure mitigation regression tests FMT'' that includes FMT_r pulse FMT'. The first two rows for tests number 1 and 2 are behavioral retestable ones and the tests number 3 to 12 are the new ones. The number of mitigation tests is incremented as a result of building a new failure regression test suite.

Table 7.5: FMT'' for modified model BM'.

#	pairs PE'	Failure	Node	BT" used	MM used	mt_{ij} used	FMT"
1	(2,1)	f_1	n_5	bt_1	MM1	mt_{11}	n_1, n_5, s_g
2	(4,3)	f_3	n_2	bt_1	MM3	mt_{31}	n_1, n_5, n_7, n_2, n_2
3	(6,2)	f_2	n_3	bt'_2	MM2	mt_{21}	$n_1, n_3, n_1, n_3, n_8, n_4, n_2$
4	(7,3)	f_3	n_8	bt'_2	MM3	mt_{31}	$n_1, n_3, n_8, n_8, n_4, n_2$
5	(14,4)	f_4	n_7	bt'_3	MM4	mt_{41}	$n_1, n_5, n_6, n_9, n_7, s_1, s_2, n_2$
6	(14,4)	f_4	n_7	bt'_3	MM4	mt_{42}	$n_1, n_5, n_6, n_9, n_7, s_1, s_3, n_2$
7	(3,1)	f_1	n_4	bt'_1	MM1	mt_{11}	n_1, n_3, n_4, s_g
8	(4,4)	f_4	n_2	bt'_1	MM4	mt_{41}	$n_1, n_3, n_4, n_2, s_1, s_2, n_2$
9	(4,4)	f_4	n_2	bt'_1	MM4	mt_{42}	$n_1, n_3, n_4, n_2, s_1, s_3, n_2$
10	(12,2)	f_2	n_6	bt'_3	MM2	mt_{21}	$n_1, n_5, n_6, n_1, n_5, n_6, n_9, n_7, n_2$
11	(8,1)	f_1	n_4	bt'_2	MM1	mt_{11}	n_1, n_3, n_8, n_4, s_g
12	(11,3)	f_3	n_5	bt'_3	MM3	mt_{31}	$n_1, n_5, n_5, n_6, n_9, n_7, n_2$

7.3 Changes to State-Event Matrix (SE)

Two types of changes can occur in SE': (1) some failures become applicable in some states (changes from 0 to 1) or (2) not applicable (changes from 1 to 0). In many cases, the system requirements are changed for some states that impact some failure types to be feasible or infeasible for those states. For example, when typed input (which can be incorrect) is replaced with button selection, an incorrectly typed input no longer occurs. Similarly, if power backup is provided, loss of power no longer is applicable. On the other hand, when new requirements are added that require mitigations where none were required before, this changes entries in the SE matrix from 0 to 1.

- Case A: Feasible to infeasible (changes from 1 to 0)

  Let F_{inf} be the failure types that have became infeasible. Then $E_{inf} = \{e|f_e \in F_{inf}\}$. Any failure mitigation tests that were derived from a pair

(p, e) where the node in position p is a node for which the failure e is no longer applicable is now obsolete. These obsolete tests are given by $FMT_o = \{fmt | fmt \in FMT \wedge fmt$ based on pair $(p, e) : se_{(node(p),e)} = 1, se'_{(node(p),e)} = 0$ and $1 \leq e \leq |F|, node(p) \in S\}$. Note that failure mitigation test suite $FMT \setminus FMT_o$ is reusable, not retestable, since we have run these tests already and they are not affected by the change.

Back to our example in section 7.1, assume failure type f_3 is no longer applicable in state n_6. From Table 3.7 and Table 3.9, we have three tests associated with f_3 but only one position using n_6. The node in position 7 is n_6. Hence, only the pair (7,3) is obsolete. It was used to create fmt_3 which is now obsolete, and position 7 is the only position using n_6. Thus, $FMT_o = \{fmt_3\}$. Note that we do not have to rerun the remaining tests in Table 3.9, since they have already been executed.

- Case B: Infeasible to feasible (changes from 0 to 1)

  This requires building a new search space for failures that have became applicable. Let F_{se} be the subset of failure types that become now feasible such that $F_{se} \subset F$. Let S_{se} be the subset of states that become applicable for any failure f_j where $f_j \in F_{se}$ and $S_{se} \subset S$.

  Using our example, let failure type f_1 become applicable in state n_6 and f_4 become applicable in state n_5. Thus, $F_{se} = \{f_1, f_4\}$, and $S_{se} = \{n_5, n_6\}$. The new SE' is defined in Table 7.6. It includes both making f_3 inapplicable in state n_6. as well as making failures f_1 and f_4 applicable in states n_5 and n_6 respectively.

The new search space SP' and new selected pairs are based on the following:

Table 7.6: New State-Event Matrix SE'

Behavioral States/ Failure Type (f)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	**dpe**
1	1	0	0	1	1	1	1	**0.71**
2	1	0	1	0	1	1	0	**0.43**
3	0	1	0	1	1	0	1	**0.58**
4	0	1	0	0	1	0	1	**0.43**
dps	**0.25**	**0.50**	**0.25**	**0.50**	**1.00**	**0.50**	**1**	

– Remove tests from BT that do not visit states in S_{se} (states for which certain failure types have become applicable). The remaining behavioral tests BT_{se} will be: $BT_{se} = \{bt_i | \exists s_i \in S_{se} : bt_i \text{ visits } s_i\}$.

  Next, we concatenate BT_{se}. Let I_{se} be the concatenation of BT_{se}.

  In the example, the affected node n_5 exists only in bt_1 and bt_2, and n_6 exists only in bt_2. Thus, $BT_{se} = \{bt_1, bt_2\}$, and the concatenation $I_{se} = \{bt_1 \circ bt_2\}$.

– Remove all failures that are not in F_{se}. In our example, we exclude f_2 and f_3.

– Rebuild the new search space such as $SP' = \{(p,e) | 1 \leq p \leq length(I_{se}), 1 \leq e \leq |F_{se}|, se'_{(node(p),e)} = 1\}$. Table 7.7 shows the new search space SP' marking each feasible entry as a "1". The new search space is: $SP' = \{(p,e) | 1 \leq p \leq 9, 1 \leq e \leq 2, se'_{(node(p),e)} = 1 \wedge node(p) \in S_{se}\}$.

Table 7.7: The New Search Space SP'

I	bt_1				bt_2				
Position (p)	1	2	3	4	5	6	7	8	9
F/N	n_1	n_5	n_7	n_2	n_1	n_5	n_6	n_7	n_2
f_1	1	1	1	0	1	1	1	1	0
f_4	0	1	0	1	0	1	0	1	1

- Select a new set of pairs PE' where PE'=$\{(p,e)|(p,e) \in SP'\}$. Table 7.7 marks where the two nodes in $S_{se} = \{n_5, n_6\}$ occur, resulting in PE'. $PE' = \{(7,1),(2,4),(6,4)\}$.

  Note, that the definition of PE' requires to select as test requirements all occurrences of nodes $s \in S_{se}$. If such nodes occur many times, it may be useful to restrict the number of times that position p are selected where $node(p) \in S_{se}$. We can use coverage criteria C2 or C3 for this. We usually expect that the search space is not large enough to use the GA.

- Generate new failure mitigation tests FMT'' as in chapter 3. Table 7.8 shows the selected pairs and new failure mitigation regression tests.

  Section 7.5 describes the case when there are multiple changes to the artifacts (BM, SE, F).

Table 7.8: Selected pairs and constructing FMT'' for F_{se}.

#	Selected pairs (PE')	Failure	Node	BT used	MM used	mt_{ij} used	FMT''
9	(7,1)	f_1	n_6	bt_2	MM1	mt_{11}	n_1, n_5, n_6, s_g
10	(2,4)	f_4	n_5	bt_1	MM4	mt_{41}	$n_1, n_5, s_1, s_2, n_7, n_2$
11	(2,4)	f_4	n_5	bt_1	MM4	mt_{42}	$n_1, n_5, s_1, s_3, n_7, n_2$

7.4 Changes to Failure Types (F)

First, we assume that the changes to failure types are the only changes. Later, we will discuss the situation when changes to multiple artifacts occur. We describe the change in failure type as follows:

- Delete failure types $F_d = \{f_{d1}, \cdots, f_{dm}\}$ where m is the number of deleted failure types: $F' = F \setminus F_d$

  An example of deleting a failure type: a faulty network can no longer cause a network connection error by using a backup router to quickly swap out the faulty network. Suppose, we delete the failure types (f_2 and f_3) from our example. Thus, $F_d = \{f_2, f_3\}$.

  Next, we remove the mitigation models $\{MM_{d_1}, \cdots MM_{d_m}\}$ and weaving rules $\{WR_{d_1}, \cdots WR_{d_m}\}$. Any failure mitigation tests that test proper mitigation for a failure $f \in F_d$ is removed as well. Let $FMT_{F_d} \subseteq FMT$ such that each $t \in FMT_{F_d}$ was constructed to test a failure type $f \in F_d$. $FMT_{F_d} = \{fmt | fmt \in FMT \wedge fmt$ based on pair $(p, e') \in PE$ where $f_{e'} \in F_d$ and $1 \leq p \leq Length(I)\}$. Note that $FMT \setminus FMT_{F_d}$ is reusable, not retestable, since we have executed these tests already.

  Back to our example, we remove the mitigation models $MM_{F_d} = \{MM_2, MM_3\}$ and weaving rules $WR_{F_d} = \{WR_2, WR_3\}$. From Table 3.9, the deleted failure mitigation tests are $FMT_{F_d} = \{fmt_2, fmt_3, fmt_6, fmt_8, fmt_9\}$. Table 7.9 shows the reusable tests after deleting failures f_2 and f_3.

Table 7.9: Reusable tests after deleting failures f_2 and f_3

#	Selected pairs (PE)	Failure	Node	BT used	MM used	mt_{ij} used	FMT
1	(2,1)	f_1	n_5	bt_1	MM1	mt_{11}	n_1, n_5, s_g
2	(8,4)	f_4	n_7	bt_2	MM4	mt_{41}	$n_1, n_5, n_6, n_7, s_1, s_2, n_7, n_2$
3	(8,4)	f_4	n_7	bt_2	MM4	mt_{42}	$n_1, n_5, n_6, n_7, s_1, s_3, n_2$
4	(8,1)	f_1	n_7	bt_2	MM1	mt_{11}	n_1, n_5, n_6, n_7, s_g

- Add new failure type $F_a = \{f_{a1}, \cdots, f_{an}\}$ where n is the number of new failure types:

  In this case, we need to build a state-event matrix for the new failure, construct a search space for the concatenated test suite and the new failures, and generate (p,e) pairs. We also must define mitigation models and weaving rules for the new failures and create failure mitigation tests based on the (p,e) pairs selected. Note that we do not need to rerun FMT, nor include existing failure types in the construction of the search space. Thus, a new state-event matrix SE_a is an $n \times |S|$ matrix where

$$SE_a(i,j) = \begin{cases} 1 & \text{if failure type j applies in node } n_i \in S \\ 0 & otherwise \end{cases}$$

  Note that number of states $i = 1, \cdots, |S|$ and number of failures types $j = 1, \cdots, n$. Using our example, we add two new failure types (f_5 and f_6), so $F_a = \{f_5, f_6\}$. Next, we have to build a new state-event matrix (SE_a) that includes the new failure types F_a as shown in Table 7.10.

  Next, we create new mitigation models $MM_a = \{MM_{a1}, \cdots, MM_{an}\}$ for each failure type in F_a. Let $WR_a = \{WR_{a1} \cdots WR_{an}\}$ be the weaving rules for the new mitigation models. Next, we generate the new mitigation test

Table 7.10: New State-Event Matrix SE_a

Behavioral States/ Failure Type (f)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	**dpe**
5	0	0	0	0	0	1	0	**0.14**
6	0	1	0	0	0	0	1	**0.29**
dps	**0.00**	**0.50**	**0.00**	**0.00**	**0.00**	**0.50**	**0.50**	

suites $MT_a = \{MT_{a1}, \cdots MT_{an}\}$. In the example, we add new mitigation models $MM_a = \{MM_5, MM_6\}$ and new weaving rules (WR_5 and WR_6) for the new failure types. They are shown in Table 7.11. The first mitigation model returns to state n_6 after a database error, the second returns to the start node n_1 and ends the test.

Next, we create new mitigation test paths MTs_a using the new mitigation models. In our case there are no paths required, since the only task is specified by the weaving rule.

Table 7.11: New Mitigation Requirements

MM	Explanation	Model	WR#
MM5	Retry: database server error	$MT_5 = \phi$ where $node(p)^r$ is the state in which we are trying to save data $node(p) = n_6$ and $r = 1$ (retry once)	2
MM6	End Activity: misunderstood behaviour such as try to access node without having specific user role	$MT_6 = \phi$ where $s_f = n_1$,and s_f is the start node and stop	4

Then, we build the new search space. The new search space is defined with the concatenated behavioral test suite BT and the new failure types as follows: $SP_a = \{(p,e)|1 \leq p \leq length(I), 1 \leq e \leq n, se_{a_{(node(p),e)}} = 1\}$. We select a new set of pairs $(p,e) \in PE_a$ where PE_a=$\{(p,e)|(p,e) \in SP_a$ (p,e) selected$\}$. How we select the (p,e) pairs depends on the size of the new search space. If it is large enough, we use the GA, otherwise coverage criteria are more appropriate.

In the example, the new search space $SP_a = \{(p,e)|1 \leq p \leq 14, 1 \leq e \leq 2, se_{a_{(node(p),e)}} = 1\}$ is shown in Table 7.12. Because the search space is too small for using GA, we use coverage criteria ($C2$) for selecting pairs (i.e all unique nodes, all applicable failures). Using coverage certeria $C2$ results in selecting three pairs as shown in Table 7.13.

Table 7.12: New Search Space SP_a due to the added failure F_a

	bt_1				bt_2					bt_3				
	n_1	n_5	n_7	n_2	n_1	n_5	n_6	n_7	n_2	n_1	n_3	n_4	n_3	n_2
f_5	0	0	0	0	0	0	1	0	0	0	0	0	0	0
f_6	0	0	1	1	0	0	0	1	1	0	0	0	0	1

Finally, new failure mitigation tests FMT_{F_a} are generated from pairs in PE_a. Hence, $FMT'' = FMT_{F_a}$

In the example, new failure mitigation test FMT_{F_a} are generated based on new weaving rules WR_5 and WR_6 for the new failure types in $F_a = \{f_5, f_6\}$. Table 7.13 shows the selected pairs and FMT''.

Table 7.13: Selected pairs and constructing FMT_{F_a} from them.

#	Selected pairs (PE)	Failure	Node	BT used	MM used	mt_{ij} used	$FMT_{F_a} = FMT''$
1	(7,5)	f_5	n_6	bt_2	MM5	mt_{51}	$n_1, n_5, n_6, n_6, n_7, n_2$
2	(3,6)	f_6	n_7	bt_1	MM6	mt_{61}	n_1, n_5, n_7, n_1
3	(4,6)	f_6	n_2	bt_1	MM6	mt_{61}	n_1, n_5, n_7, n_2, n_1

Note, we assume that the addition of new failures does not impact existing failure mitigations. Otherwise, we would need to include mitigation tests for failures that have been affected by mitigation of new failures. This can happen when the mitigations for two different failure types share some of the mitigation code. Let impacted failures types $F_c = \{f_1, \cdots, f_q\}$ be the failures impacted by new mitigation requirements, where q is the number of affected failures. Let the impacted failure mitigation tests $FMT_c = \{fmt|fmt_j$ built from pair $(p, e) \in PE$ where $f_e \in F_c\}$. The new failure mitigation tests derived from F_c will be as follows: $FMT'' = FMT_{F_a} \cup FMT_c$.
Note: FMT does not have to be rerun and FMT_c is a selection not a creation of new tests.

In our example, suppose that mitigation of new failures impacts existing failure mitigation. Let failure type f_1 be impacted by new mitigation requirements. As a result, $F_c = \{f_1\}$, and from Table 7.9 (after deleting failure mitigation tests based on deleted failures FMT_{F_d}), the impacted failure mitigation tests are $FMT_c = \{fmt_1, fmt_4\}$. The new failure mitigation tests FMT'' are shown in Table 7.14. The first two test requirements address impacted failure f_1, the others are identical to those in Table 7.13.

Table 7.14: Selected pairs and FMT''

#	Selected pairs (PE)	Failure	Node	BT used	MM used	mt_{ij} used	FMT''
1	(2,1)	f_1	n_5	bt_1	MM1	mt_{11}	n_1, n_5, s_g
2	(8,1)	f_1	n_7	bt_2	MM1	mt_{11}	n_1, n_5, n_6, n_7, s_g
3	(7,5)	f_5	n_6	bt_2	MM5	mt_{51}	$n_1, n_5, n_6, n_6, n_7, n_2$
4	(3,6)	f_6	n_7	bt_1	MM6	mt_{61}	n_1, n_5, n_7, n_1
5	(4,6)	f_6	n_2	bt_1	MM6	mt_{61}	n_1, n_5, n_7, n_2, n_1

7.5 Changes to BM , Failure Types (F) and State-Event Matrix (SE)

When *BM*, *F* and *SE* have been changed at the same time, we need to perform the following steps:

- Classify BT into BT_r, BT_u, and BT_o as in section 7.2.1.
- Generate new behavioral test cases BT' due to the changes to BM.
- Create new mitigation models MM_{aj}; add the weaving rules for the new failures F_a, and generate new mitigation tests MT_{aj} for new failure types F_a.
- Delete the mitigation models and weaving rules, and failure mitigation tests FMT_{F_d} for all deleted failure in F_d.
- Remove any obsolete mitigation tests FMT_o due to the changes to SE for any inapplicable failure types.
- Define the subset of failure types F_{se} and subset of states S_{se} that are affected and become now feasible due to the changes to SE.
- Build a new state-event matrix SE' that includes any new states due to the changes to BM and the added failure types due to the changes to F. Also, because of the changes to SE, the new state-event matrix SE' includes any failure types in F_{se}.
- Construct three new search spaces as follows (see Figure 7.3):
  - The first search space (SP_1') is for existing failure types F and new test paths BT'.
  - The second search space (SP_2') is for new failure types F_a and for all tests

(new and non-obsolete tests) such that $BT_a = BT' \cup BT_r \cup BT_u$. Since new failure types could be applicable for nodes that exist in BT_u, we need to test the mitigation requirements for F_a among reusable behavioral tests that visit these nodes.

– The third search space (SP'_3) is due to the changes to SE for newly feasible failure types in F_{se} and affected behavioral tests BT_{se}. Note that if $F_{se} \simeq F$ and $BT_{se} \simeq BT_a$ then it means basically to regenerate tests from scratch for the modified model.

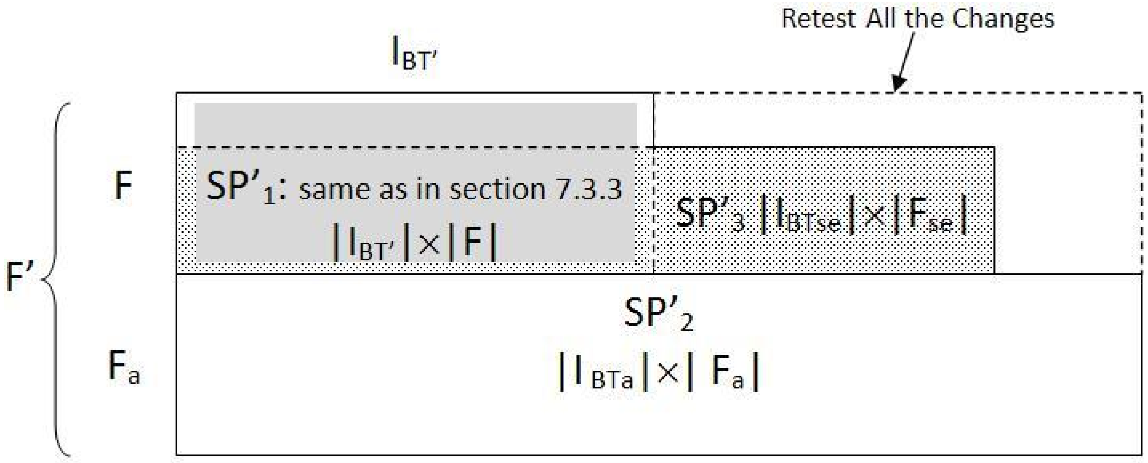


Figure 7.3: The new search space SP' due to the Changes to BM, SE, and F

The new state-event matrix SE' will be constructed as in section 7.2.2 based on the changes to BM, section 7.3 based on the changes to SE, and section 7.4 based on the changes to F. Using the example, we will use the same changes to BM as in section 7.2, and the changes to the SE are the same in section 7.3. Finally, deleted and added failure types will be similar as in section 7.4.

The search space SP' will be three different matrices. One matrix SP'_1 represents the length of BT' concatenation times the number of old failure types F. Note that we remove deleted failure F_d such as $F \setminus F_d$. The second matrix SP'_2 is the

concatenation of $BT_a = BT' \cup BT_r \cup BT_u$ in one dimension and the added failure types F_a. The third matrix SP'_3 represents the length of concatenation of affected tests BT_{se} due to the changes to SE times the feasible failure types in F_{se}. Let $I_{BT'}$ be the concatenation of BT', and I_{BT_a} be the concatenation of BT_a. Let I_{se} be the concatenation of BT_{se}. Thus, new search space will be as follows:
$SP'_1 = \{(p,e)|1 \leq p \leq length(I_{BT'}), 1 \leq e \leq |F|, se'_{(node(p),e)} = 1\}$
$SP'_2 = \{(p,e)|1 \leq p \leq length(I_{BT_a}), 1 \leq e \leq |F_a|, se'_{(node(p),e)} = 1\}$
$SP'_3 = \{(p,e)|1 \leq p \leq length(I_{se}), 1 \leq e \leq |F_{se}|, se'_{(node(p),e)} = 1\}$

From our example and based on the changes to BM (see Figure 7.2), we delete edge (n_4, n_3), and add n_8 and edges $(n_3, n_8), (n_8, n_4)$ to $FSM1$. Also, we delete edge (n_6, n_7), and add n_9 and edges $(n_6, n_9), (n_9, n_7)$ to $FSM2$. The changes to BM result into classifying BT as follows: $BT_o = \{bt_2, bt_3\}$, $BT_r = \{bt_1\}$, $BT_u = \phi$, and new test paths $BT' = \{bt'_1, bt'_2, bt'_3\}$.
Because of the changes to SE, f_1 becomes applicable in state n_6 and f_4 becomes applicable in state n_5. Thus, $F_{se} = \{f_1, f_4\}$, and $S_{se} = \{n_5, n_6\}$. We exclude any test that does not visit n_5 or n_6. Since bt_2 and bt_3 become obsolete and only bt_1 is reusable and visits n_5, $BT_{se} = \{bt_1, bt'_3\}$. Note that f_3 is already deleted due to the changes to F regardless of the changes to SE.
In the example, we delete the failure types (f_2 and f_3), and add new failure types (f_5 and f_6). Thus, $F_d = \{f_2, f_3\}$, and $F_a = \{f_5, f_6\}$.
Next, we build the new (SE') as shown in Table 7.15.

Tables 7.16-7.18 show the three search spaces. Note that each one has different dimensions. Because each search space is small, we use coverage criteria $C2$ for SP'_1, SP'_2, and SP'_3.

Table 7.15: New State-Event Matrix SE'

Behavioral States/ Failure Type (f)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	**dpe**
1	1	0	0	1	1	1	1	0	0	**0.56**
4	0	1	0	0	1	0	1	0	0	**0.33**
5	0	0	0	0	0	1	0	1	0	**0.22**
6	0	1	0	0	0	0	1	0	1	**0.33**
dps	**0.25**	**0.50**	**0.00**	**0.50**	**0.25**	**0.50**	**0.75**	**0.25**	**0.25**	

Table 7.16: New Search Space SP'_1 (BM change and f_2, f_3 deleted)

	bt'_1				bt'_2					bt'_3					
$I_{BT'}/F$	n_1	n_3	n_4	n_2	n_1	n_3	n_8	n_4	n_2	n_1	n_5	n_6	n_9	n_7	n_2
f_1	1	0	1	0	1	0	0	1	0	1	1	1	0	1	0
f_4	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1

The next step is to select a new set of pairs from each search space. Based on the size of the search space, we use GA if it is large enough, otherwise a coverage criterion. Because there are three different search spaces, we have to select three different set of pairs. Tables 7.16-7.18 mark the pairs selected by $C2$. Finally , new failure mitigation tests are generated using PE'_1, PE'_2, and PE'_3 respectively. Table 7.19, Table 7.20, and Table 7.21 show the selected pairs and failure mitigation regression tests for FMT'_1 using PE'_1, FMT'_2 using PE'_2, and FMT'_3 using PE'_3. Note that failure f_4 has multiple mitigation paths, hence $|FMT'_1| > |PE'_1|$ and

Table 7.17: New Search Space SP'_2 (new failures)

	bt_1				bt'_1				bt'_2					bt'_3					
I_{BT_a}/F_a	n_1	n_5	n_7	n_2	n_1	n_3	n_4	n_2	n_1	n_3	n_8	n_4	n_2	n_1	n_5	n_6	n_9	n_7	n_2
f_5	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
f_6	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0	1	1	1

Table 7.18: New Search Space SP'_3 (applicability change)

	bt_1				bt'_3					
I_{se}/F_{se}	n_1	n_5	n_7	n_2	n_1	n_5	n_6	n_9	n_7	n_2
f_1	1	1	1	0	1	1	1	0	1	0
f_4	0	1	1	1	0	1	0	0	1	1

$|FMT'_3| > |PE'_3|$.

Table 7.19: Constructing FMT'_1 with PE'_1.

#	pairs PE'_1	Failure	Node	BT' used	MM used	mt_{ij} used	FMT'_1
1	(1,1)	f_1	n_1	bt'_1	MM1	mt_{11}	n_1, s_g
2	(3,1)	f_1	n_4	bt'_1	MM1	mt_{11}	n_1, n_3, n_4, s_g
3	(11,1)	f_1	n_5	bt'_3	MM1	mt_{11}	n_1, n_5, s_g
4	(12,1)	f_1	n_6	bt'_3	MM1	mt_{11}	n_1, n_5, n_6, s_g
5	(14,1)	f_1	n_7	bt'_3	MM1	mt_{11}	$n_1, n_5, n_6, n_9, n_7, s_g$
6	(4,4)	f_4	n_2	bt'_1	MM4	mt_{41}	$n_1, n_3, n_4, n_2, s_1, s_2, n_2$
7	(4,4)	f_4	n_2	bt'_1	MM4	mt_{42}	$n_1, n_3, n_4, n_2, s_1, s_3, n_2$
8	(11,4)	f_4	n_5	bt'_3	MM4	mt_{41}	$n_1, n_5, s_1, s_2, n_5, n_6, n_9, n_7, n_2$
9	(11,4)	f_4	n_5	bt'_3	MM4	mt_{42}	$n_1, n_5, s_1, s_3, n_5, n_6, n_9, n_7, n_2$
10	(14,4)	f_4	n_7	bt'_3	MM4	mt_{41}	$n_1, n_5, n_6, n_9, n_7, s_1, s_2, n_7, n_2$
11	(14,4)	f_4	n_7	bt'_3	MM4	mt_{42}	$n_1, n_5, n_6, n_9, n_7, s_1, s_3, n_7, n_2$

Table 7.20: Constructing FMT'_2 with PE'_2

#	Selected pairs (PE'_2)	Failure	Node	BT_a used	MM used	mt_{ij} used	FMT'_2
1	(11,5)	f_5	n_8	bt'_2	MM5	mt_{51}	$n_1, n_3, n_8, n_8, n_4, n_2$
2	(16,5)	f_5	n_6	bt'_3	MM5	mt_{51}	$n_1, n_5, n_6, n_6, n_9, n_7, n_2$
3	(3,6)	f_6	n_7	bt_1	MM6	mt_{61}	n_1, n_5, n_7, n_1
4	(4,6)	f_6	n_2	bt_1	MM6	mt_{61}	n_1, n_5, n_7, n_2, n_1
5	(17,6)	f_6	n_9	bt'_3	MM6	mt_{61}	n_1, n_5, n_6, n_9, n_1
6	(18,6)	f_6	n_7	bt'_3	MM6	mt_{61}	$n_1, n_5, n_6, n_9, n_7, n_1$

Table 7.21: Constructing FMT'_3 with PE'_3.

#	Selected pairs (PE'_3)	Failure	Node	BT_{se} used	MM used	mt_{ij} used	FMT'_3
1	(1,1)	f_1	n_1	bt_1	MM1	mt_{11}	n_1, s_g
2	(2,1)	f_1	n_5	bt_1	MM1	mt_{11}	n_1, n_5, s_g
3	(3,1)	f_1	n_7	bt_1	MM1	mt_{11}	n_1, n_5, n_7, s_g
4	(7,1)	f_1	n_6	bt'_3	MM1	mt_{11}	n_1, n_5, n_6, s_g
5	(2,4)	f_4	n_5	bt_1	MM4	mt_{41}	$n_1, n_5, s_1, s_2, n_5, n_7, n_2$
6	(2,4)	f_4	n_5	bt_1	MM4	mt_{42}	$n_1, n_5, s_1, s_3, n_5, n_7, n_2$
7	(3,4)	f_4	n_7	bt_1	MM4	mt_{41}	$n_1, n_5, n_7, s_1, s_2, n_7, n_2$
8	(3,4)	f_4	n_7	bt_1	MM4	mt_{42}	$n_1, n_5, n_7, s_1, s_3, n_7, n_2$
9	(3,4)	f_4	n_2	bt_1	MM4	mt_{41}	$n_1, n_5, n_7, n_2, s_1, s_2, n_2$
10	(3,4)	f_4	n_2	bt_1	MM4	mt_{42}	$n_1, n_5, n_7, n_2, s_1, s_3, n_2$

7.6 Changes to Mitigation Models (MM)

We assume that BT', F', SE', and PE' have been constructed based on changes to corresponding artifacts and algorithms in sections 7.2,7.3,7.4, and 7.5. We also assume that MT'_j have been constructed for any changed model: $j \in mod_{MM} = \{j | 1 \leq j \leq |F| \wedge MM'_j \neq MM_j\}$. When changes to mitigation models occur, we need to follow:

- determine a mitigation test paths for the changed mitigation model(s).
- determine retestable FMT and pairs upon which they are based that use mitigation tests from the changed model(s).

Note that we do not have to consider PE'_2 or PE'_3 since they are based on new mitigation models, not changed ones, or failure types that used to be inapplicable, hence no retestable failure mitigation tests exist. Note that SE'_3 only specifies feasible pairs that did not exist in the prior version. Note also that PE'_1 describes new test requirements from which new failure mitigation tests are created (regardless of changes to the mitigation models).
Hence we only need to consider two cases:

- failure mitigation tests that are based on retestable mitigation tests. These need to be rerun.
- failure mitigation tests that were built based on failure types for the modified mitigation model. These need to be rebuilt with the new mitigation tests set.

Note that we need to exclude any obsolete failure mitigation tests FMT_o.

7.6.1 Determine Mitigation Test Paths for Changed Mitigation Models

Since the mitigation model is similar to the behavioral model (FSMWeb), the same concept is applied to the mitigation test paths using [7]. The same classification will be used in terms of obsolete, reusable, or retestable test paths. We classify mitigation tests (MT_j) of failure type f_j as obsolete (MT_{j_o}), retestable (MT_{j_r}), and reusable (MT_{j_u}). We determine new mitigation test cases (MT'_j) for uncovered edges in the mitigation model of failure type f_j.

Back to our example, the mitigation model MM_4 for failure type f_4 is modified as shown in Figure 7.4. We modify the model from export to Excel to be exported to Word format. We delete edges: $(s_1, s_2), (s_2, s_f)$. The deleted edges make mitigation test mt_{41} obsolete. Thus, $MT_{4_o} = \{mt_{41}\}$. We also add node (s_4) and edges: $(s_1, s_4), (s_4, s_f)$. As a result, a new mitigation test path is needed: $mt'_{41} = \{s_i, s_1, s_4, s_f\}$. Thus, $MT'_4 = \{mt'_{41}\}$. Since mitigation test $mt_{42} = \{s_i, s_1, s_3, s_f\}$ visits s_i, this makes mt_{42} retestable. Consequently, $MT_{4r} = \{mt_{42}\}$. Since only one mitigation model has been changed, $mod_{MM} = \{4\}$. The new mitigation test for MM'_4 is $MT''_4 = MT_{4r} \cup MT'_4 = \{mt_{42}, mt'_{41}\}$.

7.6.2 Failure Mitigation Tests Based Retestable Mitigation Tests

As stated before these tests need to be rerun. They constitute the retestable failure mitigation tests FMT_{rt}. They are defined as follows: $FMT_{rt} = \{fmt | fmt \in FMT$ based on: $(p, j) \in PE, j \in mod_{MM}, mt_j \in MT_{j_r}\}$. We need to rerun FMT_{rt}.

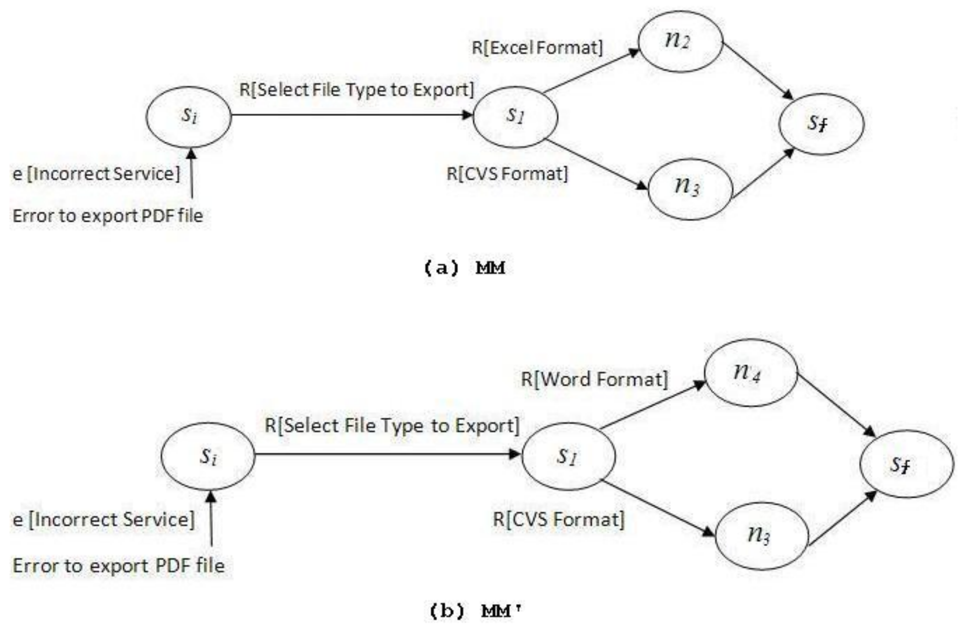


Figure 7.4: Modified Mitigation Model $MM4$.

From Table 3.9, the only failure mitigation tests used for f_4 are fmt_4 and fmt_5. However, fmt_4 is obsolete because of using mt_{41}, and fmt_5 becomes retestable because of using mt_{42}. Hence $FMT_{rt} = \{fmt_5\}$.

7.6.3 Build New Failure Mitigation Tests

Here we need to make sure that all $mt_j \in MT'_j$ ($j \in mod_{MM}$) are used to build new failure mitigation tests. This requires identifying all pairs: $PE_{MM'} = \{(p, j) | j \in mod_{MM}\}$ and then using $mt_j \in MT'_j$ to build the new failure mitigation tests FMT'_{MM}. $FMT'_{MM} = \{fmt' | \textit{ based on } (p, e) \in PE_{MM'}$ using $mt_j \in MT'_j, j \in mod_{MM}\}$. These new tests need to be run.

In the example and from Table 3.9, $PE_{MM'} = \{(8, 4)\}$ and $MT'_4 = \{mt'_{41}\}$. We generate new failure mitigation test using mt'_{41}. Thus, $fmt'_4 = \{n_1, n_5, n_6, n_7, s_1, s_4, n_7, n_2\}$.

7.6.4 Potential Impact on Other Failure Mitigations

Changes to failure mitigations can impact mitigations of other failures whose models have not changed. This can happen when they share portions of the mitigation code, for example. In such a case, the failure mitigation tests for these failures need to be rerun. Let E_I be the failure types impacted. Then all failure mitigation tests based on $PE_I = \{(p, e) | e \in E_I\}$ need to be rerun. The failure mitigation tests $FMT_I = \{fmt | fmt \in FMT$, based on $(p, e) \in PE_I\}$ are the impacted set of tests. From the example, we assume the changes to MM'_4 has affected the failure mitigation associated with failure type f_1. Using Table 3.9, $E_I = \{1\}$, $PE_I = \{(2, 1), (8, 1)\}$ and $FMT_I = \{fmt_1, fmt_7\}$. These need to be rerun.

In summary, the regression test suite FMT'' to execute consists of retestable tests FMT_{rt}, new failure mitigation tests FMT'_{MM} and tests for failures that are impacted by mitigation model changes in other failures FMT_I. Hence $FMT'' = FMT_{rt} \cup FMT'_{MM} \cup FMT_I$. Table 7.22 shows the new failure mitigation regression tests as follows:

$FMT'' = \{fmt_5\} \cup \{fmt'_4\} \cup \{fmt_1, fmt_7\} = \{fmt_1, fmt'_4, fmt_5, fmt_7\}$

Table 7.22: PE and resulting FMT''.

#	pairs (PE)	Failure	Node	BT used	MM used	mt_{ij} used	FMT”
1	(2,1)	f_1	n_5	bt_1	MM1	mt_{11}	n_1, n_5, s_g
2	(8,4)	f_4	n_7	bt_2	MM4	mt'_{41}	$n_1, n_5, n_6, n_7, s_1, s_4, n_7, n_2$
3	(8,4)	f_4	n_7	bt_2	MM4	mt_{42}	$n_1, n_5, n_6, n_7, s_1, s_3, n_2$
4	(8,1)	f_1	n_7	bt_2	MM1	mt_{11}	n_1, n_5, n_6, n_7, s_g

7.7 Changes to Weaving Rules (WR)

When a weaving rule is modified, we need to identify which (p,e) pairs are affected and we need to regenerate tests that were created using the old weaving rule, as these failure mitigation tests are now obsolete. Let mod_{WR} = {failure types for which weaving rule changed}. Let $PE_{WR} = \{(p,e)|(p,e) \in PE \wedge e \in mod_{WR}\}$. We generate new failure mitigation tests for these (p,e) pairs. Then $FMT' = \{fmt'|(p,e) \in PE_{WR}$ used to construct $fmt'\}$. We simply regenerate failure mitigation tests using appropriate coverage criteria and rerun these tests.

In the example, we update the weaving rule "End All" for failure f_2 to be "Fix and proceed". Hence $PE_{WR} = \{(11,2),(6,2)\}$ (*cf* Table 3.9). Tests fmt_2 and fmt_8 are obsolete and need to be regenerated. Table 7.23 shows the result.Using Table 3.9, we use the new weaving rule wr'_{f_2} and generate new failure mitigation tests FMT'' for them using the new weaving rule for pairs in PE_{WR}. New failure mitigation test for f_2 need to be constructed for pairs (11,2) and (6,2). The new failure mitigation tests are shown in Table 7.23.

In case of multiple changes to artifacts, we do not have to consider PE'_1 or PE'_3, since they are based on new mitigation models (and weaving rules), not changed ones or failures that used to be inapplicable, hence no retestable failure mitigation tests exist. Since PE'_1 describes new test requirements, we would have already constructed failure mitigation tests with the new weaving rules as described in section 7.5. FMT' refers to reconstruction of existing tests.

Table 7.23: PE_{WR} and resulting FMT''.

#	pairs (PE)	Failure	Node	BT used	MM used	mt_{ij} used	FMT”
2	(11,2)	f_2	n_3	bt_3	MM2	mt_{21}	$n_1, n_3, n_3, n_4, n_3, n_2$
8	(6,2)	f_2	n_5	bt_2	MM2	mt_{21}	$n_1, n_5, n_5, n_6, n_7, n_2$

7.8 Discussion

We build the framework for selective regression testing of fail-safe testing. We provide a systematic method by showing the formalization steps for each type of change to the various models (BM, MM, WR, F, SE) and build a regression test suite based on each type of change, allowing for multiple changes to artifacts. If the new search space is sufficiently large, we use GA [5] to construct the new test requirements, otherwise coverage criterion (CC)[6] are the best choice. In case changes to the behavioral model are the only change and the changes did not add many new states/edges or delete many states/edges, there will be few new mitigation tests. When failure types no longer apply or when some failure types become not applicable in some nodes (entries in SE matrix change from 1 to 0), the number of failure mitigation tests becomes smaller. However, when new failure types are required to be mitigated, or some failure types become applicable in states where they were not applicable before, the number of failure mitigation tests increases. Therefore, the changes to behavioral models BM, the changes to SE, or adding new failure types have higher impact as more work is involved such as building a SE', search space $SP_1 - SP_3$, and generating more failure mitigation tests. On the other hand, changes to mitigation models or weaving rules have less impact because they are local changes. Multiple changes to the artifacts can become very expensive and may require to create a full new test suite. Also in this case, we are also more likely to use a GA.[3]

[3]This is a trade-off situation, that is beyond the scope of this dissertation and will be considered in future work research.

7.9 Case Study

We use the behavioral model of the Closing Documents (CD) sub system in section 6.1 to illustrate our regression testing framework. The next subsections describe the changes to artifacts(BM,F,SE,MM,WR) and how we build the search space and generate the regression failure mitigation tests for all the changes.

7.9.1 Changes to BM

Figure 7.5 shows the changes to the behavioral model of the CD subsystem. We have added a new node (FF), and edges p3_n1, p3_n2, and deleted a node (SI) and the edges related to it.

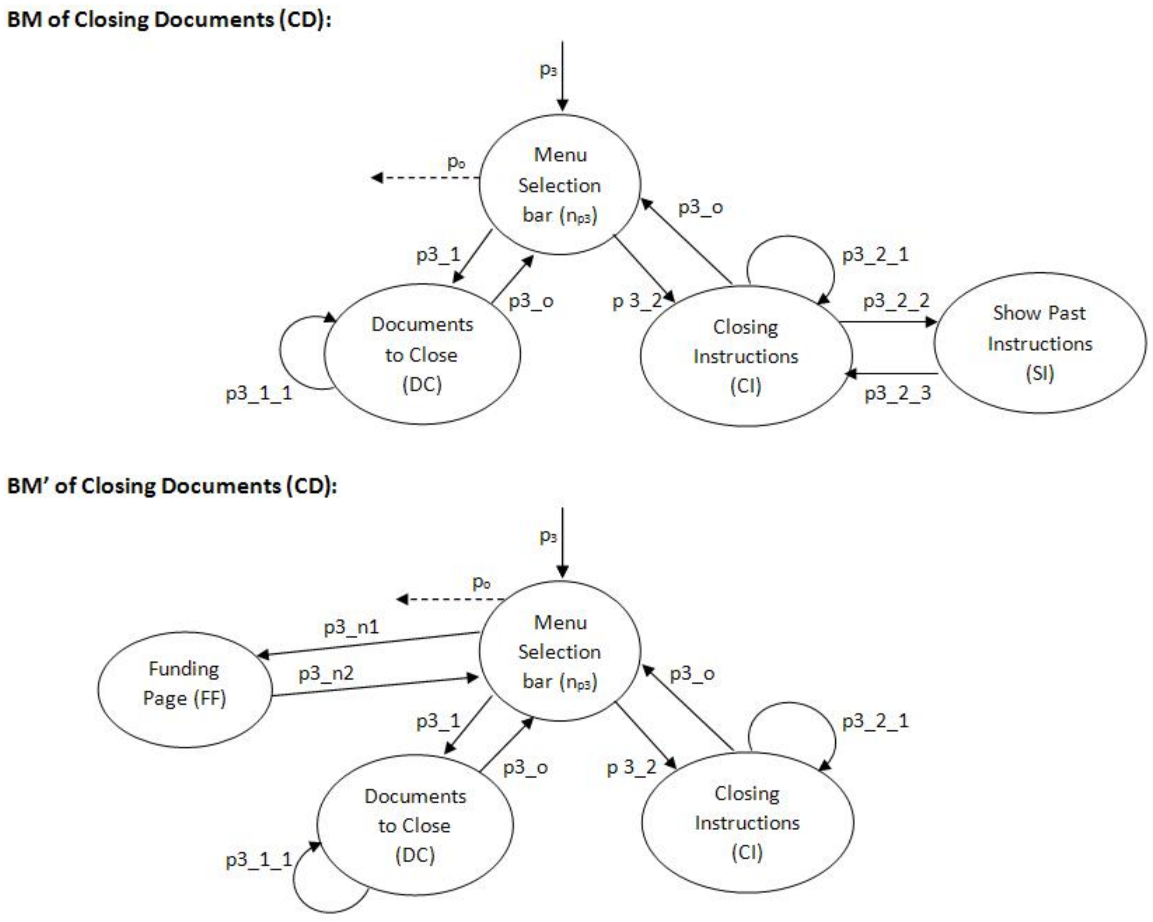


Figure 7.5: Changes to BM of CD sub-system.

7.9.1.1 Classify BT and Build BT'

The following steps describe the changes to BM:

- Classify BT int BT_o, BT_r, and BT_u:
  - The set of deleted nodes $N_o = \{SI\}$, then the set of obsolete test paths due to node changes $O_N = \{T_{CD3}, T_{CD7}, T_{CD8}, T_{CD9}\}$. The set of deleted edges $E_o = \{(CI, SI), (SI, CI)\}$, then the set of obsolete test paths due to edge changes $O_E = \{T_{CD3}, T_{CD7}, T_{CD8}, T_{CD9}\}$. Thus, obsolete tests $BT_o = O_N \cup O_E = \{T_{CD3}, T_{CD7}, T_{CD8}, T_{CD9}\}$.

- The set of added edges $E' \setminus E = \{(n_{p3}, FF), (FF, n_{p3})\}$, and the set of retestable nodes $N_r = \{n_{p3}, CI\}$.

  Thus, retestable tests $BT_r = \{T_{CD2}, T_{CD10}, T_{CD11}, T_{CD12}\}$.
- Reusable tests $BT_u = BT \setminus (BT_o \cup BT_r) = \{T_{CD1}, T_{CD4}, T_{CD5}, T_{CD6}\}$.

- The following edges in the BM':$\{(n_{p3}, FF), (FF, n_{p3})\}$ are not covered. Thus, the new tests $BT' = \{T_{new1}, T_{new2}, T_{new3}, T_{new4}\}$ as shown in Table 7.24.

Table 7.24 shows the classification of behavioral tests BT and the new tests BT'. The first column shows the clusters of aggregation sequence for CD subsystem. The second column shows the test number, third column the test path and the last column the classification of test path based on the changes to BM.

Table 7.24: Classification of BT

Clusters	**Test No.**	**Test Path**	Classification
Home-LPD-CD	T_{CD1}	w_0,n_0,VC,VC,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	Reusable
Home-LPD-CD	T_{CD2}	w_0,n_0,VC,VC,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	Retestable
Home-LPD-CD	T_{CD3}	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	Obsolete
SE-LPD-CD	T_{CD4}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	Reusable
SE-LPD-CD	T_{CD5}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	Reusable
SE-LPD-CD	T_{CD6}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},DC,DC,n_{p3},n_e,n_0,w_0	Reusable
SE-LPD-CD	T_{CD7}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	Obsolete
SE-LPD-CD	T_{CD8}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	Obsolete
SE-LPD-CD	T_{CD9}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,SI,CI,n_{p3},n_e,n_0,w_0	Obsolete
SE-LPD-CD	T_{CD10}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	Retestable
SE-LPD-CD	T_{CD11}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	Retestable
SE-LPD-CD	T_{CD12}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	Retestable
Home-LPD-CD	T_{new1}	w_0,n_0,VC,VC,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	New
SE-LPD-CD	T_{new2}	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	New
SE-LPD-CD	T_{new3}	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	New
SE-LPD-CD	T_{new4}	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	New

7.9.1.2 Build SE' matrix

The new state-event matrix SE' is shown in Table 7.25. It is the same as the original SE in Table 6.21 except for the added column for the new state FF.

Table 7.25: New State-Event Matrix SE' based on the changes to BM

Behavioral States/ Failure Type (f)	w_0	n_0	VC	n_e	n_{p3}	SP	SR	AS	EE	DC	CI	FF	**dpe**
1	1	1	1	1	1	1	1	1	1	1	1	1	**1.0**
2	0	1	1	1	1	1	1	1	1	1	1	1	**0.9**
3	1	0	0	0	0	1	0	0	0	1	0	1	**0.3**
4	0	0	0	0	0	1	0	1	0	1	0	1	**0.3**
5	0	0	0	0	0	0	0	0	0	1	1	1	**0.3**
6	0	1	1	1	1	1	1	1	1	1	1	1	**0.9**
7	0	0	0	0	0	0	0	0	0	1	0	1	**0.1**
8	0	0	0	0	0	0	0	0	0	1	1	0	**0.2**
9	0	0	0	0	0	0	0	0	0	1	1	0	**0.2**
10	0	0	0	0	0	0	1	0	0	0	1	0	**0.2**
dps	**0.2**	**0.3**	**0.3**	**0.3**	**0.3**	**0.5**	**0.4**	**0.4**	**0.3**	**0.9**	**0.7**	**0.7**	

7.9.1.3 Build new search space SP'and FMT''

We consider only the new BT' in building SP'. Concatenating BT' such as: $I' = (T_{new1} \circ T_{new2} \circ T_{new3} \circ T_{new4})$. The length of I' is 51, and the number of failure types is $E = |F| = 10$. Thus, the new search space $SP' = \{(p,e)|1 \leq p \leq 51, 1 \leq e \leq 10, se'_{(node(p),e)} = 1\}$. We select new pairs PE' using SP'. Since the search space size is reasonable to use GA, we apply GA for selecting new pairs PE'. Next, we generate new failure mitigation tests FMT'' using the existing weaving rules that are defined in Table 6.20.

Table 7.26 shows the selected (p, e) pairs and resulting FMT'' for the modified model BM'. The first column in Table 7.26 numbers each failure mitigation test $(fmt_1 - fmt_{29})$. The second column lists each (p, e) pair in PE'. The third column refers to the failure type whose mitigation is tested. The fourth column states the node at position p. The fifth column identifies the new behavioral test used in constructing fmt_i $(i = 1, \cdots, 29)$. The sixth column identifies which mitigation model is used as described in Table 3.6. The seventh column lists which mt_{ij} is used as described in Table 3.6. The last column shows the failure mitigation tests. Table 7.26 shows the new failure mitigation regression tests FMT'' that includes FMT_r pulse FMT'. The $(fmt_1 - fmt_{15})$ are behavioral retestable ones and the tests number 9 to 20 are the new ones. The number of mitigation tests is incremented as a result of building a new failure regression test suite.

Table 7.26: Constructing FMT'' for modified model BM'

#	(p,e)	Failure	Node	MM used	FMT'	BT used
1	(165,4)	f_4	CI	MM4	T_{CD12}	T_{CD12}
2	(133,5)	f_5	n_e	MM5	T_{CD10}	T_{CD10}
3	(135,6)	f_6	CI	MM6	T_{CD10}	T_{CD10}
4	(19,6)	f_6	CI	MM6	T_{CD2}	T_{CD2}
5	(19,9)	f_9	CI	MM9	w_0,n_0,VC,VC,n_e,n_{p3},CI,n_{p3}	T_{CD2}
6	(129,9)	f_9	SP	MM9	T_{CD10}	T_{CD10}
7	(148,9)	f_9	n_{p3}	MM9	T_{CD11}	T_{CD11}
8	(15,1)	f_1	VC	MM1	w_0,n_0,VC,sg	T_{CD2}
9	(130,1)	f_1	SR	MM1	w_0,n_0,SP,SR,sg	T_{CD10}
10	(136,2)	f_1	CI	MM2	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,sg	T_{CD10}
11	(158,2)	f_1	AS	MM2	w_0,n_0,SP,AS,sg	T_{CD12}
12	(157,2)	f_2	SP	MM2	w_0,n_0,SP,w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	T_{CD12}
13	(131,2)	f_2	EE	MM2	w_0,n_0,SP,SR,EE,w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,n_{p3},n_e, n_0,w_0	T_{CD10}
14	(152,2)	f_2	n_e	MM2	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,w_0,n_0,SP,SP,AS,SR, n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	T_{CD11}
15	(19,3)	f_3	CI	MM3	T_{CD2}	T_{CD2}
16	(47,3)	f_3	FF	MM3	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},FF,FF, n_{p3},n_e,n_0,w_0	T_{new4}
17	(30,1)	f_1	SR	MM1	w_0,n_0,SP,SP,AS,SR,s_g	T_{new3}
18	(4,2)	f_2	VC	MM2	w_0,n_0,VC,VC,w_0,n_0,VC,VC,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new1}
19	(7,5)	f_5	FF	MM5	w_0,n_0,VC,VC,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new1}
20	(40,3)	f_3	SP	MM3	w_0,n_0,SP,AS,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new4}
21	(22,6)	f_6	n_e	MM6	T_{new2}	T_{new2}
22	(16,1)	f_1	EE	MM1	w_0,n_0,SP,SR,EE,sg	T_{new2}
23	(32,2)	f_2	n_{p3}	MM2	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3}, n_e,n_0,w_0	T_{new3}
24	(12,10)	f_{10}	n_0	MM10	T_{new2}	T_{new2}
25	(28,4)	f_4	SP	MM4	w_0,n_0,SP,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new3}
26	(43,6)	f_6	AS	MM10	T_{new4}	T_{new4}
27	(35,1)	f_1	n_e	MM1	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,sg	T_{new3}
28	(1,3)	f_3	w_0	MM3	w_0,w_0,n_0,VC,VC,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new1}
29	(39,6)	f_6	n_0	MM2	w_0,n_0,w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new4}

7.9.2 Changes to SE

We assume failure f_7 becomes inapplicable in state DC and f_7 becomes applicable in state AS. From Table 6.25, we have one two tests associated with f_7 but only one position using DC. The node in position 46 is DC. Only the pair (46,7) is obsolete. It was used to create fmt_{16} which is now obsolete hence $FMT_o = \{fmt_{16}\}$. Hence $F_{se} = \{f_7\}$, and $S_{se} = \{AS\}$. The new SE' is defined in Table (7.27). It includes both making f_7 inapplicable in state DC as well as making f_7 applicable in state AS respectively. From Table 7.24, behavioral tests that visits states in S_{se} are $BT_{se} = \{T_{CD5}, T_{CD6}, T_{CD8}, T_{CD9}T_{CD11}, T_{CD12}\}$.

Table 7.27: New State-Event Matrix SE' based on the changes to SE

Behavioral States/ Failure Type (f)	w_0	n_0	VC	n_e	n_{p3}	SP	SR	AS	EE	DC	CI	SI	**dpe**
1	1	1	1	1	1	1	1	1	1	1	1	1	**1.0**
2	0	1	1	1	1	1	1	1	1	1	1	1	**0.9**
3	1	0	0	0	0	1	0	0	0	1	0	0	**0.3**
4	0	0	0	0	0	1	0	1	0	1	0	0	**0.3**
5	0	0	0	0	0	0	0	0	0	1	1	1	**0.3**
6	0	1	1	1	1	1	1	1	1	1	1	1	**0.9**
7	0	0	0	0	0	0	0	1	0	0	0	0	**0.1**
8	0	0	0	0	0	0	0	0	0	1	1	0	**0.2**
9	0	0	0	0	0	0	0	0	0	1	1	0	**0.2**
10	0	0	0	0	0	0	1	0	0	0	1	1	**0.3**
dps	**0.2**	**0.3**	**0.3**	**0.3**	**0.3**	**0.5**	**0.4**	**0.4**	**0.3**	**0.9**	**0.7**	**0.5**	

We rebuild the new search space $SP' = \{(p,e)|1 \leq p \leq 116, 1 \leq e \leq 1, se'_{(node(p),e)} = 1 \wedge node(p) \in S_{se}\}$. We select a new pairs PE' using coverage criteria $C2$. Table 7.28 shows the selected pairs and new failure mitigation tests.

Table 7.28: Selected pairs and Constructing FMT'' for F_{se}

#	(p,e)	Failure	Node	MM used	FMT'	BT used
26	(78,7)	f_7	AS	MM7	w_0,n_0,SP,AS,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	T_{CD12}

7.9.3 Changes to F

7.9.3.1 Delete Failure types

We delete f_1 as a faulty network can no longer cause a network connection error by using a backup router to quickly swap out the faulty network, and f_2 as session expiration failure is no longer valid. Hence the deleted failure types are $F_d = \{f_1, f_2\}$, we remove the mitigation models $\{MM_1, MM_2\}$ and weaving rules $\{WR_1, WR_2\}$ for the failures $f \in F_d$. From Table 6.25, the deleted failure mitigation tests are $FMT_{F_d} = \{fmt_{25}, fmt_{26}, fmt_{27}, fmt_{28}, fmt_{29}, fmt_{30}, fmt_{31}, fmt_{32}, fmt_{33}, fmt_{34}, fmt_{35}, fmt_{36}, fmt_{37}, fmt_{38}, fmt_{39}, fmt_{40}, fmt_{41}, fmt_{42}, fmt_{43}, fmt_{44}, fmt_{45}, fmt_{46}\}$. Table 7.29 shows reusable failure mitigation tests after deleting f_1 and f_2.

7.9.3.2 Add Failure types

We add a mitigation requirement for power outage as explained (failure f_{11}). Hence $F_a = \{f_{11}\}$. We have to build a new state-event matrix (SE_a) that includes the new failure types F_a as shown in Table 7.30. Then, we create a new mitigation model $MM_a = \{MM_{11}\}$ and weaving rule WR_{11} for the new mitigation model (see Table 7.31).

The new search space is defined with the concatenated behavioral test suite BT and the new failure types as follows: $SP_a = \{(p, e) | 1 \leq p \leq 169, 1 \leq e \leq 1, se_{a_{(node(p),e)}} = 1\}$. Because the search space is too small for using GA, we use coverage criteria ($C2$) for selecting pairs as shown in Table 7.32.

Table 7.29: Reusable tests after deleting failures f_1 and f_2

#	(p,e)	Failure	Node	MM used	FMT	BT used
1	(33,3)	f_3	SI	MM3	T_{CD3}	T_{CD3}
2	(2,3)	f_3	n_0	MM3	T_{CD1}	T_{CD1}
3	(60,3)	f_3	DC	MM3	w_0,n_0,SP,SP,AS,SR,n_e,n_{p3},DC,DC,DC,n_{p3},n_e,n_0,w_0	T_{CD5}
4	(75,4)	f_4	DC	MM4	w_0,n_0,SP,AS,SP,AS, SR,n_e,n_{p3},DC,DC,DC,n_{p3},n_e,n_0,w_0	T_{CD6}
5	(165,4)	f_4	CI	MM4	T_{CD12}	T_{CD12}
6	(5,5)	f_5	VC	MM5	T_{CD1}	T_{CD1}
7	(89,5)	f_5	CI	MM5	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,n_{p3}	T_{CD7}
8	(133,5)	f_5	n_e	MM5	T_{CD10}	T_{CD10}
9	(41,6)	f_6	SR	MM6	T_{CD4}	T_{CD4}
10	(42,6)	f_6	EE	MM6	T_{CD4}	T_{CD4}
11	(56,6)	f_6	AS	MM6	T_{CD5}	T_{CD5}
12	(135,6)	f_6	CI	MM6	T_{CD10}	T_{CD10}
13	(7,6)	f_6	DC	MM6	T_{CD1}	T_{CD1}
14	(19,6)	f_6	CI	MM6	T_{CD2}	T_{CD2}
15	(105,6)	f_6	SI	MM6	T_{CD8}	T_{CD8}
16	(46,7)	f_7	DC	MM7	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},DC,mti,DC,n_{p3},n_e,n_0,w_0	T_{CD4}
17	(90,7)	f_7	SI	MM7	T_{CD7}	T_{CD7}
18	(93,8)	f_8	n_e	MM8	T_{CD7}	T_{CD7}
19	(120,8)	f_8	CI	MM8	w_0,n_0,SP,AS,SP,AS, SR,n_e,n_{p3},CI,CI,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD9}
20	(19,9)	f_9	CI	MM9	w_0,n_0,VC,VC,n_e,n_{p3},CI,n_{p3}	T_{CD2}
21	(129,9)	f_9	SP	MM9	T_{CD10}	T_{CD10}
22	(148,9)	f_9	n_{p3}	MM9	T_{CD11}	T_{CD11}
23	(81,10)	f_{10}	w_0	MM10	T_{CD7}	T_{CD7}
24	(32,10)	f_{10}	CI	MM10	w_0,n_0,VC,VC,n_e,n_{p3},CI,n_1, n_2,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD3}
25	(32,10)	f_{10}	CI	MM10	w_0,n_0,VC,VC,n_e,n_{p3},CI,n_1, n_3,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD3}
26	(19,3)	f_3	CI	MM3	T_{CD2}	T_{CD2}

Table 7.30: New state-event matrix SE_a

Behavioral States/ Failure Type (f)	w_0	n_0	VC	n_e	n_{p3}	SP	SR	AS	EE	DC	CI	FF
11	1	1	1	1	1	1	1	1	1	1	1	1

Table 7.31: New Mitigation Requirements of f_{11}

MM	Explanation	Model	WR#
MM11	End Activity: power outage	$MT_{11} = \phi$ where $s_f = w_0$,and s_f is the start node and stop	4

Table 7.32: Selected pairs and Constructing FMT_{F_a}

#	(p,e)	Failure	Node	MM used	FMT'	BT used
14	(1,11)	f_{11}	w_0	MM11	w_0,w_0	T_{CD1}
15	(2,11)	f_{11}	n_0	MM11	w_0,n_0,w_0	T_{CD1}
16	(3,11)	f_{11}	VC	MM11	w_0,n_0,VC,w_0	T_{CD1}
17	(5,11)	f_{11}	n_e	MM11	w_0,n_0,VC,VC, n_e, w_0	T_{CD1}
18	(6,11)	f_{11}	n_{p3}	MM11	w_0,n_0,VC,VC, n_e,n_{p3}, w_0	T_{CD1}
19	(7,11)	f_{11}	DC	MM11	w_0,n_0,VC,VC, n_e,n_{p3},DC, w_0	T_{CD1}
20	(19,11)	f_{11}	CI	MM11	w_0,n_0,VC,VC,n_e,n_{p3},CI,w_0	T_{CD2}
21	(32,11)	f_{11}	SI	MM11	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,w_0	T_{CD3}
22	(40,11)	f_{11}	SP	MM11	w_0,n_0,SP,w_0	T_{CD4}
23	(41,11)	f_{11}	SR	MM11	w_0,n_0,SP,SR,w_0	T_{CD4}
24	(42,11)	f_{11}	EE	MM11	w_0,n_0,SP,SR,EE,w_0	T_{CD4}
25	(56,11)	f_{11}	AS	MM11	w_0,n_0,SP,SP,AS,w_0	T_{CD5}

7.9.4 Changes to BM, F, and SE

We apply the same changes as described in previous subsections at the same time. The new SE' is defined in Table (7.33). It includes both making f_7 inapplicable in state DC as well as making f_7 applicable in state AS respectively. Also, it includes the changes to BM by showing new state FF.

Table 7.33: State-Event Matrix for CD Cluster

Behavioral States/ Failure Type (f)	w_0	n_0	VC	n_e	n_{p3}	SP	SR	AS	EE	DC	CI	FF	**dpe**
3	1	0	0	0	0	1	0	0	0	1	0	1	**0.33**
4	0	0	0	0	0	1	0	1	0	1	0	1	**0.33**
5	0	0	0	0	0	0	0	0	0	1	1	1	**0.25**
6	0	1	1	1	1	1	1	1	1	1	1	1	**0.92**
7	0	0	0	0	0	0	0	1	0	0	0	1	**0.17**
8	0	0	0	0	0	0	0	0	0	1	1	0	**0.17**
9	0	0	0	0	0	0	0	0	0	1	1	0	**0.17**
10	0	0	0	0	0	0	1	0	0	0	0	0	**0.08**
11	1	1	1	1	1	1	1	1	1	1	1	1	**1.0**
dps	**0.22**	**0.22**	**0.22**	**0.22**	**0.22**	**0.44**	**0.33**	**0.44**	**0.22**	**0.78**	**0.56**	**0.67**	

We build the search space is based on all the changes to BM, F, and SE. Since we already classify BT into BT_r, BT_u, BT_o and generate new behavioral test paths BT' as described in subsection 7.9.1, we build the first search space SP'_1 for existing failure type F and new test paths BT'. Based on the changes to F as describe in subsection 7.9.3, we remove F_d from F such as $F \setminus F_d$. As a result of concatenating the new tests $I' = (T_{new1} \circ T_{new2} \circ T_{new3} \circ T_{new4})$, the $length(I_{BT'}) = 51$. Thus, $SP'_1 = \{(p, e) | 1 \leq p \leq 51, 1 \leq e \leq 9, se'_{(node(p),e)} = 1\}$.

From subsection 7.9.3, we define the new mitigation model, weaving rules and mitigation tests for the new failure F_a. Also, we delete the mitigation models, weaving rules and failure mitigation tests FMT_{fd} for all deleted failure in F_d. Then we build

the second search space SP'_2 for new failure types F_a and for all tests (new and non-obsolete tests) BT_a. Thus, $BT_a = BT_r \cup BT_u \cup BT' =$
$\{T_{CD2}, T_{CD10}, T_{CD11}, T_{CD12}, T_{CD1}, T_{CD4}, T_{CD5}, T_{CD6}, T_{new1}, T_{new2}, T_{new3}, T_{new4}\}$. The length of $|BT_a| = 110$. $SP'_2 = \{(p,e)|1 \leq p \leq 110, 1 \leq e \leq 1, se'_{(node(p),e)} = 1\}$. Based on the changes to SE as described in subsection 7.9.2, we remove any obsolete tests FMT_o for any inapplicable failure types, and we define the subset of failure types F_{se} and subset of states S_{se} that are affected and become feasible. We build the third search space SP'_3 due to the changes to SE for feasible failure types in F_{se} and affected behavioral tests BT_{se}. From Table 7.24, behavioral tests that visits states in S_{se} are $BT_{se} = \{T_{CD5}, T_{CD6}, T_{CD11}, T_{CD12}, T_{new3}, T_{new4}\}$. Note we exclude obsolete tests due to the BM changes. Thus, $SP'_3 = \{(p,e)|1 \leq p \leq 85, 1 \leq e \leq 1, se'_{(node(p),e)} = 1\}$.

We remove all failure mitigation tests for any failure $f \in F_d$. From Table 6.25, the deleted failure mitigation tests $FMT_{F_d} = \{fmt_{25}, fmt_{26}, ...to..., fmt_{46}\}$ and retestable failure mitigation tests:
$FMT_r = \{fmt_5, fmt_8, fmt_{12}, fmt_{14}, fmt_{20}, fmt_{22}, fmt_{47}\}$.

We need to determine failure mitigation test based on:

- Selected pairs PE'_1 using SP'_1 which is the new tests BT' and $F \setminus F_d$.
- Selected pairs PE'_2 using SP'_2 which is BT_a and new failure types F_a.
- Selected pairs PE'_3 using SP'_3 which is BT_{se} and F_{se}.

Table 7.34 shows the regression failure mitigation tests for FMT'_1 using PE'_1. Note that failure mitigation tests ($fmt_1 - fmt_8$) are derived from retestable behavioral tests BT_r and the rest from new tests BT'. The first column in Table 7.34 numbers each failure mitigation test. The second column lists each (p, e) pair in PE'. The third column refers to the failure type whose mitigation is tested. The forth column states the node at position p. The fifth column identifies the mitigation model used. The sixth column shows the construction of failure mitigation test. The last column refers to the behavioral test used.

Since SP'_2 and SP'_3 have a small search space, we use coverage criteria $C2$. Table 7.35 and Table 7.36 shows the regression failure mitigation tests for FMT'_3 using PE'_3 and for FMT'_3 using PE'_3.

Table 7.34: Constructing FMT'_1 with PE'_1

#	(p,e)	Failure	Node	MM used	FMT'	BT used
1	(165,4)	f_4	CI	MM4	T_{CD12}	T_{CD12}
2	(133,5)	f_5	n_e	MM5	T_{CD10}	T_{CD10}
3	(135,6)	f_6	CI	MM6	T_{CD10}	T_{CD10}
4	(19,6)	f_6	CI	MM6	T_{CD2}	T_{CD2}
5	(19,9)	f_9	CI	MM9	w_0,n_0,VC,VC,n_e,n_{p3},CI,n_{p3}	T_{CD2}
6	(129,9)	f_9	SP	MM9	T_{CD10}	T_{CD10}
7	(148,9)	f_9	n_{p3}	MM9	T_{CD11}	T_{CD11}
8	(19,3)	f_3	CI	MM3	T_{CD2}	T_{CD2}
9	(47,3)	f_3	FF	MM3	w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},FF,FF, n_{p3},n_e,n_0,w_0	T_{new4}
10	(7,5)	f_5	FF	MM5	w_0,n_0,VC,VC,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new1}
11	(40,3)	f_3	SP	MM3	w_0,n_0,SP,AS,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new4}
12	(22,6)	f_6	n_e	MM6	T_{new2}	T_{new2}
13	(12,10)	f_{10}	n_0	MM10	T_{new2}	T_{new2}
14	(28,4)	f_4	SP	MM4	w_0,n_0,SP,SP,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new3}
15	(43,6)	f_6	AS	MM10	T_{new4}	T_{new4}
16	(1,3)	f_3	w_0	MM3	w_0,w_0,n_0,VC,VC,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new1}
17	(39,6)	f_6	n_0	MM2	w_0,n_0,w_0,n_0,SP,AS,SP,AS,SR,n_e,n_{p3},FF,n_{p3},n_e,n_0,w_0	T_{new4}

7.9.5 Changes to MM

The mitigation model $MM10$ for failure type f_{10} has been modified as shown in Figure 7.6. We modify the model from export to Excel to be exported to Word format. We delete edges: $(n_1, n_2), (n_2, s_f)$. The deleted edges make mitigation test mt_{10_1} obsolete. Thus, $MT_{4o} = \{mt_{10_1}\}$. We also add node (n_4) and edges: $(n_1, n_4), (n_4, s_f)$. As a result, a new mitigation test path is needed: $mt'_{10_1} =$

Table 7.35: Constructing FMT'_2 with PE'_2

#	(p,e)	Failure	Node	MM used	FMT'	BT used
14	(1,11)	f_{11}	w_0	MM11	w_0,w_0	T_{CD1}
15	(2,11)	f_{11}	n_0	MM11	w_0,n_0,w_0	T_{CD1}
16	(3,11)	f_{11}	VC	MM11	w_0,n_0,VC,w_0	T_{CD1}
17	(5,11)	f_{11}	n_e	MM11	w_0,n_0,VC,VC, n_e, w_0	T_{CD1}
18	(6,11)	f_{11}	n_{p3}	MM11	w_0,n_0,VC,VC, n_e,n_{p3}, w_0	T_{CD1}
19	(7,11)	f_{11}	DC	MM11	w_0,n_0,VC,VC, n_e,n_{p3},DC, w_0	T_{CD1}
20	(19,11)	f_{11}	CI	MM11	w_0,n_0,VC,VC,n_e,n_{p3},CI,w_0	T_{CD2}
21	(27,11)	f_{11}	SP	MM11	w_0,n_0,SP,w_0	T_{CD4}
22	(28,11)	f_{11}	SR	MM11	w_0,n_0,SP,SR,w_0	T_{CD4}
23	(29,11)	f_{11}	EE	MM11	w_0,n_0,SP,SR,EE,w_0	T_{CD4}
24	(43,11)	f_{11}	AS	MM11	w_0,n_0,SP,SP,AS,w_0	T_{CD5}
25	(117,11)	f_{11}	FF	MM11	w_0,n_0,VC,VC,n_e,n_{p3},FF,w_0	T_{new1}

Table 7.36: Constructing FMT'_3 with PE'_3

#	(p,e)	Failure	Node	MM used	FMT'	BT used
26	(47,7)	f_7	AS	MM7	w_0,n_0,SP,AS,AS,SP,AS,SR,n_e,n_{p3},CI,CI,n_{p3},n_e,n_0,w_0	T_{CD12}

$\{s_i, n_1, n_4, s_f\}$. Thus, $MT'_{10} = \{mt'_{10_1}\}$. Since mitigation test $mt_{10_2} = \{s_i, n_1, n_3, s_f\}$ visits s_i, this makes mt_{10_2} retestable. Consequently, $MT_{10r} = \{mt_{10_2}\}$. Since only one mitigation model has been changed, $mod = \{10\}$. The new mitigation test for MM'_{10} is $MT''_{10} = MT_{10r} \cup MT'_{10} = \{mt_{10_2}, mt'_{10_1}\}$. From table 6.25, we identify all affected pairs $PE_{MM'} = \{(32, 10)\}$ then using MT'_{10} to build the new failure mitigation test FMT'. Table 7.37 shows the new failure mitigation test; however, if we consider the changes to BM, this failure mitigation tests becomes obsolete as it was built based on obsolete behavioral test T_{CD3}.

Table 7.37: FMT' for modified mitigation model $MM10$

#	(p,e)	Failure	Node	MM used	FMT	BT used
24	(32,10)	f_{10}	SI	MM10	w_0,n_0,VC,VC,n_e,n_{p3},CI,SI,n_1, n_4,CI,n_{p3},n_e,n_0,w_0	T_{CD3}

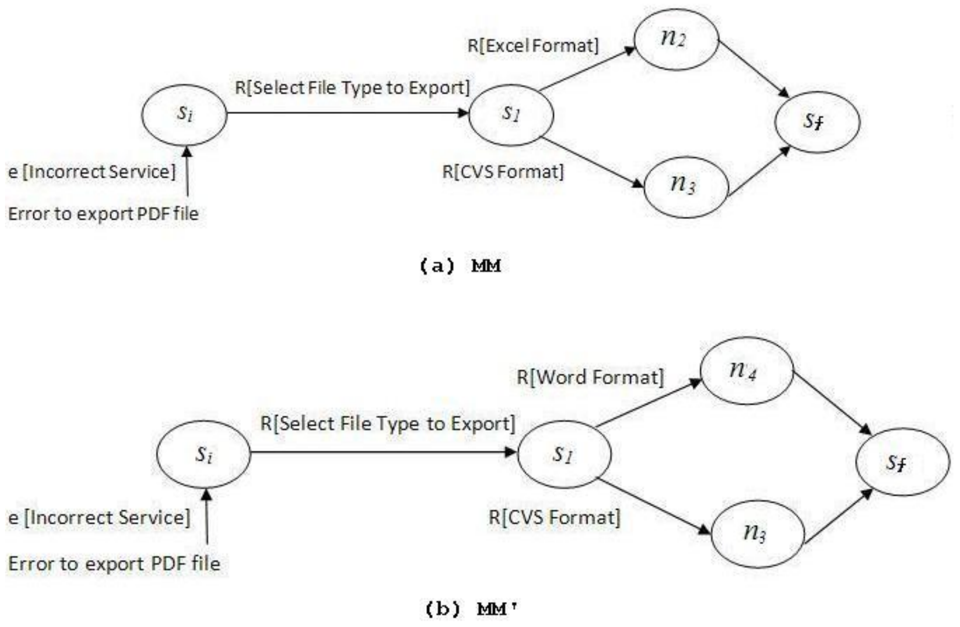


Figure 7.6: Modified Mitigation Model $MM10$.

7.9.6 Changes to WR

We update the weaving rule "End Activity" for failure type f_5 to be "Fix and proceed". Next from Table 6.25, we identify affected pairs $PE_{WR} = \{(89,5)\}$ then using new weaving rule to build the new failure mitigation test FMT'. Table 7.38 shows the new failure mitigation test; however, if we consider the changes to BM, this failure mitigation tests becomes obsolete as it was built based on obsolete behavioral test T_{CD7}.

Table 7.38: FMT' for modified weaving rule WR

#	(p,e)	Failure	Node	MM used	FMT	BT used
7	(89,5)	f_5	CI	MM5	w_0,n_0,SP,SR,EE,SR,n_e,n_{p3},CI,CI,SI,CI,n_{p3},n_e,n_0,w_0	T_{CD7}

Chapter 8

Future Work

In future work, the dissertation could be extended in many ways:

- **New system domains:**

  This dissertation is focused on the web applications domain. We plan to apply the technique in different system domains such as medical systems, robotic devices, flight control systems or other safety critical systems, so we could apply and compare the mitigation patterns of safety critical systems versus the mitigation patterns of web applications. By involving new system domains, we investigate different failure types and how they might occur in different system domains. This would also require using behavioral models other than FSMWeb.

- **New Behavioral models:**

  We will use different behavior models for our approach such as UML activity and sequence diagrams, Petri Nets, EFSMs or CEFSMs. These models have been used in different application domains. We will show the generalizability of our approach by investigating other behavioral models that have the ability to describe communicating processes.

- **More Simulation Experiments:**

  We will investigate robustness of our current results experimenting with a wider range of values for simulation parameters such as applicability level, defect density, and duplication factor. We noticed during our experiments that a low defect density with a low applicability level in a large search space has been a very tough problem since the feasible region is not big enough and this affects the GA's performance. For example, some of the safety critical systems have a very large number of failure types but they are only applicable in particular phases which means a sparse (SE) matrix. Also, the duplication factor has an enormous effect on defect detection. Thus, exploring high and low values for different combinations of simulation parameters could increase understanding of our approach. Hence, future work will take us into two different directions:

  - investigate the compound impact of varying more individual parameters.
  - investigate further the types of parameter values occurring in case studies.

- **Improve Fitness Function:**

  We plan to add cost of failure as an additional dimension to the approach when selecting test requirements. Failure Mode and Effect Criticality Analysis (FMECA) is one analysis method that is used to calculate the anticipated risk and criticality of failures in a system. We incorporate the ideas of cost and FMECA into black-box testing using GA that would enhance system reliability and performance. We could add failure criticality as a third dimension to the fitness function, or we can incorporate FMECA into the Applicability Matrix.

- **Regression Testing Case study:**

  We need to apply our frame work for selective regression testing with case studies, so we can show applicability and generalizability of our technique. Also, by using different case studies, we show how our formalization steps for each type of change to the various models (BM, MM, WR, F, SE) are applicable. Moreover, we could illustrate how our regression testing framework in generating a regression test suite is efficient.

- **Trade-Off Analysis: Full Retest vs. Selective Regression Testing:**

  In regression testing, multiple changes to the artifacts can become very expensive and may require to create a full new test suite; however, this is a trade-off situation. We plan to build a test cost model that identifies the conditions under which our selective regression testing is less expensive than the retest-all strategy. Trade-off analysis could help to assess the cost-benefits of our technique. For example, comparing various regression testing approaches could help in selecting a regression testing approach that achieves a favorable reduced cost.

- **Trade-Off Analysis: Cost of Testing vs. Cost of Failure**

  One of the motivations for testing proper failure mitigation was that defective mitigations can be costly. On the other hand, testing also carries its own cost. This is another trade-off situation that can be investigated.

- **Building Tools:**

  We will build tools to generate executable tests. This would decrease the cost of testing. At the moment only a tool for defining FSMWeb models and generating test paths exists.

Chapter 9

Conclusion

Testing proper mitigation of failures in web applications is important since defects in mitigation code can cause expensive outages, such as compromising proper debiting of credit cards and other financial harm. We presented a systematic MBT approach to derive fail-safe tests for web applications. It leverages an existing functional test suite. A fault model and mitigation model are used to define mitigation tests in a rigorous manner. Weaving rules specify how to weave mitigations into the functional tests at selected points of failure in the functional test suite. This requires determining which type of failure is to be injected at which position in the test suite ((p,e)pairs). A genetic algorithm is used to determine points of failure and type of failure that needs to be tested.

First, we build a simulation that helps in investigating and understating our approach. We address the threats that are related to the choice of simulation parameters used in the GA as follows:

- The weights w_r and w_s are based on tuning experiments performed in our work [5]. We select the values based on the simulation results. The tuning relies on the use of published mitigation defect rates (around 20%) (i.e Sawaelpong *et al.* [67]). We also experiment with a much lower defect rate (5%). This supports our use of a common defect rate of 20%, contrasting it with a low one (5%) as well. While it may be possible to tune these weights for higher efficiency, this would expose to potential overfitting. We are hence willing to accept that the GA is not always optimally tuned.

- As for the choice of mutation rate and crossover, we use values that have been suggested in the literature, as pointed out in section 5.2.1. Similarly, the number of runs the GA makes is commonly used as well [2].
- We next turn to the choice of parameter values for each experiment. Test suite size *length(I)* ranges from 5-1000 while $|F|$ ranges from 2-10. This falls within the range described for the case studies in Table 5.15, except for the large Mortgage system (length(I)=3998). That means that if the trends observed in our simulation experiments do not hold for larger search spaces, validity is limited to the sizes we experimented with. Similarly, our choice of failure types spans the ranges reported for the case studies, hence is realistic.
- The applicability level was set close to the highest found in the case studies. Higher applicability levels increases the search space and make the problem more difficult, i.e. our results are conservative. To the degree future case studies follow the same pattern, the simulation results should carry over. If a future application has very different characteristics, careful consideration of the differences and their impact becomes important, as generalizability is not assured.
- As for the duplication factor values, we derived them from the case studies presented in section 5.3. The low DF=2 is used for the small search space while DF=20 is used for the large ones. To be conservative (i.e. defining a

more difficult search problem), we used the smallest DF found in the case studies (DF=2) and a smaller DF=20 than the one found in the largest case study (DF=31). In the case studies, DF ranges from 2-31.

Clearly, when applications deviate by a large amount from this, the experimentation results are affected. Specifically, a higher duplication factor results in fewer pairs needed to find mitigation defects as the probability of selecting a position with a node that is associated with a mitigation defect increases (it occurs more frequently in I). In summary, we selected simulation parameter values guided by either literature or case studies. If a future case study has very different characteristics, our results may not hold.

Second, we compare GA versus Random in generating fail-safe scenarios. We showed how to tune the fitness function and compared the performance of the GA to random selection of fail-safe test mitigation requirements. We showed how to tune the fitness function and compared the performance of the GA to random selection of fail-safe scenarios. We show that the GA performance was better than random selection and that the approach was robust when the search space increased.

Third, we compare efficiency and effectiveness of using a GA vs. coverage criteria when testing proper mitigation of failures in safety-critical systems and web applications. The goal was to evaluate and compare the use of GA [5] versus the use of coverage criteria (C1-C4) advocated in [6], neither of which provided such a comparison.

We designed and implemented a simulator so as to be able to vary problem size (search space) mitigation defect density, and type of approach used (GA versus C1-C4). The simulator also assumes that history (i.e. position of a state s_i in I) makes no difference. If it does, C2 and C3 will show differences in detecting defects.

Next, we consider choice of measures for the dependent variables. Effectiveness is measured by the proportion of mitigation defects found. This is common for many experiments as a measure of effectiveness, see for example [76]. Notice that both GA and coverage criteria generate test requirements rather than executable test cases. If a human tester were to take these requirements, they would have to turn them into tests, execute them, and validate them. There is potential for error in each of these steps that could affect effectiveness results. Hence our simulation results need to be interpreted (like many other GA simulation, e.g. [29]) as a best case scenario.

Efficiency is measured in terms of (p,e) pairs (i.e. number of test requirements). Some test requirements may take more effort to turn into test cases than others, hence one can argue that it might not always reflect testing effort spent. On the other hand, test requirements have been used in the past to predict test effort [17].

Our comparisons show that for large search spaces the GA is more efficient (more so when compared to C1 than to C2/C3) while GA, and C1-C3 are equally effective. C4 was rather ineffective and cannot be recommended.

We also showed that dropping the defect rate from a more common 20% as reported in [67] to a much lower 5% did not result in a large increase in test requirements, i.e. the GA is relatively robust in this range (although an extremely low mitigation defect rate might change that. However, empirical data does not support such a low mitigation defect rate occurring in practice).

The simulation results are more in favor of C1-C3 when search spaces are small. Additionally, dropping the mitigation defect rate had an effect on how big the search space had to be for the GA to be effective.

We also presented results of case studies where we investigated and compared performance of GA and C1-C4 for several reasons:

1. They provided guidance on what ranges of parameter values to use in the simulation study.

2. They showed applicability of results to a number of behavioral models, application domains and model sizes.

3. While the actual number of test requirements that GA vs. C1-C4 produce may vary from the simulation results, the nature of the recommendation stays the same. We thus were able to investigate on a snapshot basis duplication factors between 4 and 31 and applicability levels between 40%-81%, failure types ranging between 2-14, and *length(I)* ranging from 24-3998.

Fourth, we explored the use of GA to test proper mitigation of failures in an actual web application. We reported summary results for a large web application, a mortgage system, as well as detailed results of its Closing Documents subsystem. We showed that the GA found seeded mitigation defects. We also showed scalability of our approach to a large commercial web application. Since a GA is able to deal with large search spaces, it was a good choice for the case study presented.

We also used the case study to evaluate how good the initial population is by comparing the initial population selected via defect potential (Algorithm 3) against multiple runs of randomly selected initial populations (10 runs). The results clearly show that using defect potential to generate test requirements outperforms random selection.

Multiple runs are possible when the use of a GA is explored with a simulator as in [58][13]. However, when actual test cases need to be generated, executed, and validated to determine a test requirements fitness, this GA evaluation cost becomes prohibitive for multiple runs. We accept a local rather than global optimum as long as the mitigation defects are found, For quantitative results on evaluation cost see section 6.5 in our case study. Note also the global minimum in terms of number of test requirements is equal to the number of mitigation defects that exist. Additionally, Cantú-Paz [19] explore whether multiple runs of GA can reach solutions of higher quality or reach acceptable solutions faster. Their results suggest that with a fixed evaluation budget a single run reaches a better solution than multiple independent runs.

Finally, we built a framework for selective regression testing of fail-safe testing. We provide a systematic method by showing the formalization steps for each type of change to the various models (BM, MM, WR, F, SE) and build a regression test suite based on each type of change, allowing for multiple changes to artifacts. If the new search space is sufficiently large, we use GA [5] to construct the new test requirements, otherwise coverage criterion (CC)[6] are the best choice. We show full formalization based on type of changes to the behavioral model (BM), failure types (F), state-event matrix (SE), and mitigation model (MM). We show how to build a new search space (I') and how to build the new failure mitigation tests (FMT'). The changes to behavioral models BM, changes to SE, or adding new failure types have higher impact as more work is involved.

References

[1] Enrique Alba and Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Comput. Oper. Res.*, 35:3161–3183, October 2008.

[2] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742 –762, nov.-dec. 2010.

[3] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013, USA, first edition, 2008.

[4] A. Andrews and Hyunsook Do. Trade-off analysis for selective versus brute-force regression testing in FSMWeb. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 184–192, Jan 2014.

[5] Anneliese Andrews, Salah Boukhris, and Salwa Elakeili. Fail-safe testing of web applications. In *Software Engineering Conference (ASWEC), 2014 23rd Australian*, pages 200–209, April 2014.

[6] Anneliese Andrews, Salwa Elakeili, and Salah Boukhris. Fail safe test generation in safety critical systems. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 49–56, 2014.

[7] Anneliese A. Andrews, S. Azghandi, and O. Pilskalns. Regression testing of web applications using FSMWeb. pages 8–10, CA, 725-033, November 2010. Marina del Rey.

[8] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery,

Kshamta Jerath, and Roger Alexander. Scalability issues with using FSMWeb to test web applications. *Inf. Softw. Technol.*, 52(1):52–66, January 2010.

[9] Anneliese Amschler Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Software and System Modeling*, pages 326–345, 2005.

[10] D. Ardagna, C. Cappiello, M.G. Fugini, E. Mussi, B. Pernici, and P. Plebani. Faults and recovery actions for self-healing web services. 15th international World Wide Web conference, 2006.

[11] Thomas Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In *Parallel Problem Solving from Nature 2, PPSN-II, Brussels, Belgium, September 28-30, 1992*, pages 87–96, 1992.

[12] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, and A. Watkins. Breeding software test cases with genetic algorithms. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*, HICSS '03, pages 338.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[13] D. J. Berndt and A. Watkins. High volume software testing using genetic algorithms. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Volume 09*, pages 318–326, Washington, DC, USA, 2005. IEEE Computer Society.

[14] P. Bourque and R. Dupuis. Guide to the software engineering body of knowledge 2004 version. *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*, pages –, 2004.

[15] Marco Brambilla, Stefano Ceri, Sara Comai, and Christina Tziviskou. Exception handling in workflow-driven web applications. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 170–179, New

York, NY, USA, 2005. ACM.

[16] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 43–50, New York, NY, USA, 2002. ACM.

[17] Ilene Burnstein. *Practical software testing: a process-oriented approach.* Springer Science & Business Media, 2003.

[18] Bruno Cabral and Paulo Marques. Exception handling: a field study in Java and .NET. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 151–175, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Erick Cantú-Paz and DavidE. Goldberg. Are multiple runs of genetic algorithms better than one? In Erick Cantú-Paz, JamesA. Foster, Kalyanmoy Deb, LawrenceDavid Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell Standish, Graham Kendall, Stewart Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, MitchA. Potter, AlanC. Schultz, KathrynA. Dowsland, Natasha Jonoska, and Julian Miller, editors, *Genetic and Evolutionary Computation — GECCO 2003*, volume 2723 of *Lecture Notes in Computer Science*, pages 801–812. Springer Berlin Heidelberg, 2003.

[20] K.S.May Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. A fault taxonomy for web service composition. In Elisabetta Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 363–375. Springer Berlin Heidelberg, 2009.

[21] T. Chow. Testing software designs modeled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[22] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975.

[23] Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1081–1082, New York, NY, USA, 2005. ACM.

[24] Kinga Dobolyi and Westley Weimer. Modeling consumer-perceived web application fault severities for testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 97–106, New York, NY, USA, 2010. ACM.

[25] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* SpringerVerlag, 2003.

[26] Salwa Elakeili. *Fail safe test generation of safety critical systems.* PhD dissertation, Computer Science Department, University of Denver, November 2014.

[27] Joseph G. D'Ambrosio Christopher L. Denlinger Deron Littlejohn Eldon G. Leaphart, Barbara J. Czerny. Survey of software failsafe techniques for safety-critical automotive applications. Technical report, SAE, 2005.

[28] Marisa A. Sánchez Andmiguela. Felder, Bahía Blanca, Pragma Consultores, and Buenos Aires. A systematic approach to generate test cases based on faults. In *In ASSE2003, ISSN 1666 1087, Buenos Aires*, 2003.

[29] Erik M Fredericks, Byron DeVries, and Betty HC Cheng. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 17–26. ACM, 2014.

[30] Xiaocheng Ge, R.F. Paige, and J.A. McDermid. An iterative approach for de-

velopment of safety-critical software and safety arguments. In *Agile Conference (AGILE), 2010*, pages 35–43, Aug.

[31] Mati Golani and Avigdor Gal. Flexible business process management using forward stepping and alternative paths. In WilM.P. Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin Heidelberg, 2005.

[32] Anatoliy Gorbenko, Iraj Elyasi-Komari, Vyacheslav S. Kharchenko, and Alexey Mikhaylichenko. Exception analysis in service-oriented architecture. In Heinrich C. Mayr and Dimitris Karagiannis, editors, *ISTA*, volume 107 of *LNI*, pages 228–233. GI, 2007.

[33] Joesph Gradecki and Nicholas Lesiecki. *Mastering AspectJ.* Wiley, 2003.

[34] Yuepu Guo and Sreedevi Sampath. Web application fault classification - an exploratory study. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 303–305, New York, NY, USA, 2008. ACM.

[35] C. Hagen and G. Alonso. Exception handling in workflow management systems. *Software Engineering, IEEE Transactions on*, 26(10):943–958, Oct.

[36] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: a comprehensive analysis and review of trends techniques and applications. Department of Computer Science, King's College London, King's College London, 2009. Department of Computer Science-Technical Report.

[37] W. Howden. A methodology for generating program test data. *IEEE Transactions on Computers*, 24(5):554–560, 1975.

[38] J. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.

[39] Shujuan Jiang and Yuanpeng Jiang. An analysis approach for testing exception handling programs. *SIGPLAN Not.*, 42(4):3–8, April 2007.

[40] AbdulSalam Kalaji, Robert M. Hierons, and Stephen Swift. Automatic generation of test sequences form EFSM models using evolutionary algorithms. Brunel University, UK, 2008. Brunel University-School of Information Systems, Computing and Mathematics.

[41] HyeonJeong Kim, W. Eric Wong, Vidroha Debroy, and DooHwan Bae. Bridging the gap between fault trees and UML state machine diagrams for safety snalysis. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, APSEC '10, pages 196–205, Washington, DC, USA, 2010. IEEE Computer Society.

[42] D.C. Kung, Chien-Hung Liu, and P. Hsia. An object-oriented web test model for testing web applications. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, pages 537–542.

[43] Mark Last, Shay Eyal, and Abraham Kandel. Effective black-box testing with genetic algorithms. In *Proceedings Haifa verification Conference*, pages 134–148, 2005.

[44] D. Lee and M. Yamakakis. Principles and methods of testing finite state machines. *Proceedings of the IEEE*, 84(4):1090–1123, August 1996.

[45] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines- a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[46] B.S. Lerner, S. Christov, L.J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise. Exception handling patterns for process modeling. *IEEE Transactions on Software Engineering*, 36(2):162 –183, March-April 2010.

[47] H.K.N. Leung and L. White. A cost model to compare regression test strategies. In *Conference on Software Maintenance, 1991., Proceedings.* , pages 201–208,

Oct 1991.

[48] C. Liu, D.C. Kung, P. Hsia, and C. Hsu. Structural testing of web applications. In *Proceedings of the 11th IEEE International Symposium on Software Reliability Engineering*, pages 84–96, October 2000.

[49] Qin Lu, Weishi Zhang, Bo Su, and Xiuguo Zhang. Exception handling policies for composite web services and their formal description. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, pages 793–798, Sept.

[50] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: the state of the art and future trends. *Information and Software Technology*, 48(12):1172 – 1186, 2006. Quality Assurance and Testing of Web-Based Applications.

[51] Li Ma and Jeff Tian. Analyzing errors and referral pairs to characterize common problems and improve web reliability. In JuanManuelCueva Lovelle, Bernardo-MartínGonzález Rodríguez, JoseEmilioLabra Gayo, María Puerto Paule Ruiz, and LuisJoyanes Aguilar, editors, *Web Engineering*, volume 2722 of *Lecture Notes in Computer Science*, pages 314–323. Springer Berlin Heidelberg, 2003.

[52] A.D. Shaligram Maha Alzabidi, Ajay Kumar. In *Automatic software structural testing by using evolutionary algorithms for test data generations*, volume 9, pages 390–395. IJCSNS International Journal of Computer Science and Network Security, April 2009.

[53] Samir W Mahfoud. *Niching methods for genetic algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1995.

[54] James McCart, Donald Berndt, and Alison Watkins. Using genetic algorithms for software testing: Performance improvement techniques.

In *Proceedings Americas Conference on Information Systems (AMCIS)*, http://aisel.aisnet.org/amcis2007/222, 2007.

[55] W. van der Aalst N. Russell and A. ter Hofstede. Exception handling patterns in process-aware information systems. PM Center Report BPM-06-04, 2006.

[56] Radu Oprisa. Error handling in software systems: modeling and testing with finite state machines. *ROMAI J.*, 2(1):175—180, 2006.

[57] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[58] Robert M. Patton, Annie S. Wu, and Gwendolyn H. Walton. A genetic algorithm approach to focused software usage testing. *Annals of Software Engineering*, 2003.

[59] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical report, Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-05-109, December 2005.

[60] S. Pimont and J. C. Rault. A software reliabiity assessment based on a structural and behavioral analysis of programs. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 486–491, San Francisco, CA, October 1976.

[61] Lihua Ran, Curtis Dyreson, Anneliese Andrews, Renée Bryce, and Christopher Mallery. Building test cases and oracles to automate the testing of web database applications. *Inf. Softw. Technol.*, 51(2):460–477, February 2009.

[62] N. M. Razali and J. Geraghty. Genetic algorithm performance with different selection strategies in solving TSP. *Proceedings of the World Congress on Engineering*, II, July 2011.

[63] Leo Rela. Evolutionary computing in searchbased software engineering. Mas-

ter's thesis, Lappeenranta University of Technology, Department of Information Technology, Kaohsiung, Finland, 2004.

[64] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 25–34, May.

[65] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[66] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A.S. Greenwald. Applying concept analysis to user-session-based testing of web applications. *Software Engineering, IEEE Transactions on*, 33(10):643–658, 2007.

[67] E. Sawadpong, P. Allen and B. Williams. Exception handling defects: an empirical study. In *2012 IEEE International Symposium on High-Assurance Systems Engineering*, pages 90–97, 2012.

[68] A. C. Schultz, J. J. Grefenstette, and K. A. De Jong. Adaptive testing of controllers for autonomous vehicles. In *Proceedings of the 1992 Symposium on autonomous underwater vehicle technology*, pages 158–164, 1992.

[69] S. Sinha and M.J. Harrold. Analysis of programs with exception-handling constructs. In *In Proceedings of the International Conference on Software Maintenance*, pages 348–357, Nov. 1998.

[70] Marisa A. Sánchez, Juan Carlos Augusto, and Miguel Felder. Fault-based testing of e-commerce applications. *Proceedings of 2nd Workshop on Verification and Validation of Enterprise Information Systems*, pages 66–71, 2004.

[71] Rafael Tolosana-Calasanz, José A. Bañares, Pedro Álvarez, Joaquín Ezpeleta, and Omer F. Rana. Exception handling patterns for hierarchical scientific workflows. In *Proceedings of the 6th international workshop on Middleware for grid computing*, MGC '08, pages 10:1–10:6, New York, NY, USA, 2008. ACM.

[72] A.C. Tribble and S.P. Miller. Software intensive systems safety analysis. *Aerospace and Electronic Systems Magazine, IEEE*, 19(10):21 – 26, Oct. 2004.

[73] Anneliese von Mayrhauser and Ning Zhang. Automated regression testing using DBT and Sleuth. *Journal of Software Maintenance*, 11(2):93–116, March 1999.

[74] A. Watkins, E. M. Hufnagel, D. Berndt, and L. Johnson. Using genetic algorithms and decision tree induction to classify software failures. *International Journal of Software Engineering & Knowledge Engineering*, 16(2):269 – 291, 2006.

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

[76] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[77] Liangzhao Zeng, Hui Lei, Jun-Jang Jeng, J. Y Chung, and B. Benatallah. Policy-driven exception-management for composite web services. In *E-Commerce Technology, 2005. CEC 2005. Seventh IEEE International Conference on*, pages 355–363, July.