A Complete & Practical Approach to Ensure the Legality of a Signal Transmitted by a Cognitive Radio

By

Patrick Carpenter Cowhig

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Dr. Michael S. Hsiao, Chair
Dr. Charles W. Bostian
Dr. Allen B. MacKenzie

September 5, 2006
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords:
Software Testing, Tabu Search,
Software Built-In Self-Test (BIST), Cognitive Radio

A Complete & Practical Approach to Ensure the Legality of a Signal Transmitted by a Cognitive Radio

Patrick Carpenter Cowhig

ABSTRACT

The computational power and algorithms needed to create a cognitive radio are quickly becoming available. There are many advantages to having a radio operated by cognitive engine, and so cognitive radios are likely to become very popular in the future. One of the main difficulties associated with the cognitive radio is ensuring the signal transmitted will follow all FCC rules. The work presented in this thesis provides a methodology to guarantee that all signals will be legal and valid. The first part to achieving this is a practical and easy to use software testing program based on the tabu search algorithm that tests the software off-line. The primary purpose of the software testing program is to find most of the errors, especially structural errors, while the radio is not in use so that it does not affect the performance of the system. The software testing program does not provide a complete assurance that no errors exist, so to supplement this deficit, a built-in self-test (BIST) is employed. The BIST is designed with two parts, one that is embedded into the cognitive engine and one that is placed into the radio's API. These two systems ensure that all signals transmitted by the cognitive radio will follow FCC rules while consuming a minimal amount of computational power.

The software testing approach based on the tabu search is shown to be a viable method to test software with improved results over previous methods. Also, the software BIST demonstrated its ability to find errors in the signal production and is dem to only require an insignificant amount of computational power. Overall, the methods presented in this paper provide a complete and practical approach to assure the FCC of the legality of all signals in order to obtain a license for the product.

FUNDING

This project is supported by Award No. 2005-IJ-CX-K017 awarded by the National Institute of Justice, Office of Justice Programs, US Department of Justice. The opinions, findings, and conclusions or recommendations expressed in this publication/program/exhibition are those of the author(s) and do not necessarily reflect the views of the Department of Justice.

This material is based upon work supported by the National Science Foundation under Grants No. 9983463, DGE-9987586, and CNS-0519959. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Michael Hsiao for his direction, support and motivation throughout this work. I would also like to thank Dr. Charles W. Bostian and Dr. Allen B. MacKenzie for graciously serving on my committee. Last, but not the least by any means, I would like to thank my mother, my father, and my sister, as well as my longtime girlfriend, Amber Davidson, who have helped me throughout my time in graduate school and made my stay enjoyable.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This chapter provides an introduction to the motivation and background of this project as well as the outline of the thesis.

1.1 MOTIVATION

Traditional transceivers are set to a certain specification by either the user turning physical knobs, the hardware being defined for an individual specification, or a combination of both. These methods have worked well, but do have their limitations. As processing power increases, especially for portable devices, and communication technology progresses, it is now possible to reduce some of the limitations of traditional transceivers by developing software defined radios (SDR) that are controlled by a cognitive engine.

Cognitive radios operate by having an embedded program decide upon the best signal and alter the radio accordingly. This has been compared to a radio operator turning the knobs on a radio and checking the meters, but in this case, it is done by a software program. This has many advantages over traditional transceivers as the cognitive engine can find and use "white spaces" with little or no operator intervention, can modify the radio to achieve the best performance, and can be set to meet certain objectives, such as maximizing battery life [1]. These tasks can be completed very quickly with little or no human intervention.

The complexity that enables cognitive radios a performance advantage over traditional radios also poses some very challenging testing problems. The testing problems are even further complicated as the radios have to be approved by the Federal Communications Commission (FCC) before they are allowed to be used. The FCC requires that any signal produced will not break any rules set for the product, and will not

interfere with any other signal. The main challenge of testing cognitive radios is that they can transmit any number of possible signals, and depend largely on what environment it is in. The environment may also be unexpected and therefore, difficult to test.

Thus, the work of this thesis is to build and apply a testing approach that will ensure that any signal produced will be completely compliant with FCC rules and regulations. The approach also needs to be minimally evasive as to not affect the operation or runtime of the cognitive radio.

1.2 APPROACH AND CONTRIBUTIONS

In order to ensure the legality of the signal being produced, this study proposes using both an off-line and an on-line test to catch errors. The errors targeted are unintended actions of the program, but of primary concern are those that result in a signal that does not follow FCC guidelines. The off-line test which is proposed is a dynamic test that is based on the tabu search, while the on-line test is a built-in self-test (BIST) for software that checks the signal production at various stages. Each method is used in order to counteract the weaknesses of the other. Dynamic tests are a very practical method of testing both software and hardware for errors, but the time needed to achieve a 100% complete test for large programs or circuits is far too large. The BIST, on the other hand, will be able to catch any possible error that was missed by the software testing program. The drawbacks with this method are that it will increase runtime, and thus energy consumption, as well as adding to the size and complexity of the program. Complemented with the software testing program, these drawbacks will be minimized to near negligible amounts.

The off-line test which is used is a simulation-based test that is based on the tabu search. Tabu search is an established optimization technique that can compete with all other known techniques, and because of its flexibility, often outperforms many classical procedures [2]. A description of the basic tabu search and how it is applied to this method will be discussed later. There has been one published attempt at using the tabu search for software testing [3], but this approach was hindered by a lack of learning

during the search and large increases in runtime with the addition of extra inputs to the program. The method proposed in this thesis will attempt to improve upon this work by reducing runtime and improving the learning during the test. In doing so, this will result in a practical and efficient testing program.

The online test, or BIST, has two parts to it. These two parts correspond to the two parts of the cognitive radio: the selection of which signal to use, and the production of the signal. For the selection of the signal, a mask has been designed whose primary purpose is to ensure that the signal chosen is a legal and valid signal. In addition to this purpose, the mask can improve a signal's fitness by guiding the searching process in a better direction. After the signal has been chosen, the radio then begins to produce the signal in software. Another test is used for this stage which ensures the signal production is done correctly. Here, the produced signal is checked to see if it is the same as the signal selected from the previous segment. If the produced signal is the same, which has been proven to be legal by the previous test, then it can be assured that the signal being sent to the hardware is legal and valid.

1.3 THESIS OUTLINE

An outline of the rest of the thesis is as follows:

- Chapter 2 outlines the basic definitions and terminology used. It provides an overview of the cognitive radio, as well as the cognitive engine and genetic algorithm which is used to control it, an introduction to software testing along with data generation and coverage metrics, and details of the basic operations of a tabu search.
- Chapter 3 proposes an automated software testing approach that is based on the tabu search. An in-depth description of the coverage metric, data generation, and error analysis will provide a complete analysis of how the process works. The testing program will be able to check for general errors within any software code. After the approach is presented, results and a comparison of the results to other attempts at software testing using a tabu search will be presented.

- Chapter 4 proposes a built-in self-test (BIST) for a cognitive radio. It provides the need and the details for the two part implementation. The first of these parts is a mask that is embedded into the cognitive engine to check that the signal being decided upon is legal. The second is a checker in the application program interface (API) that ensures that the production of the signal is correct. Finally, this chapter will prove that any signal developed by the cognitive radio is legal with respect to the rules it has been provided.
- Chapter 5 presents conclusions drawn from the work and recommendations for future work.

CHAPTER 2

BACKGROUND

This chapter presents the basics that are needed to understand the research in the later chapters. The cognitive radio that is to be tested will be described as well as the genetic algorithm used to choose the signal and the USRP individually. A brief background on software testing will be presented with different approaches that have been researched and compared. Lastly, a description of how the tabu search works and comparisons to other approaches will be examined.

2.1 COGNITIVE RADIOS

Cognitive radios are a very interesting research area because of their ability to efficiently use the radio frequency (RF) spectrum and optimize the transmission of a signal dependent on the users need, with little or no intervention by an operator. This has many advantages over current radio systems, from the ease of use to the ability to transmit a signal in "white spaces" where current systems cannot transmit. Due to these advantages, cognitive radios can have widespread appeal among users with many varied goals.

The term "cognitive radio" and the basic understanding of the subject was developed by Joseph Mitola who published his ideas in multiple publications [4][5][6][7], and eventually in a doctorial thesis [8]. His work has sparked a variety of research into the area of cognitive radios. Currently, there is no standard definition for cognitive radio, and because of this, there are a wide variety of definitions. Some definitions of a cognitive radio may just be another person's definition of an adaptive radio. The work done on this project defines a cognitive radio as a transceiver that is aware of the following [1]:

- The RF environment
- Its own power, frequency and waveform capabilities
- Its users requirements and operating abilities
- The regulations, etiquette, and protocols that governs its operation

This definition of a cognitive radio is very similar to how Mitola descried a cognitive radio in [4]. Here he called it a goal-driven framework where the radio autonomously observes the radio environment, infers context, assesses alternatives, generates plans, supervises services, and learns from its mistakes. He continued by pointing out that this observe-think-act cycle is radically different from today's handsets that either blast out on the frequency set by the user, or blindly take instructions from the network. Cognitive radio technology thus empowers radios to observe more flexible radio etiquettes than was possible in the past.

Cognitive radios operate through the use of "knobs" and "meters" inside a software defined radio (SDR) platform. The "meters" represent the ability for the radio to receive and measure the stimulus from the outside world. Based upon what the "meters" read, and the goals of the radio, the cognitive engine will then turn the "knobs" to obtain the optimal signal under those circumstances. This is one of the advantages of a cognitive radio, the ability to modify itself and deal with unexpected situations. Once the signal has been decided upon in the cognitive engine, it is sent to the SDR, where the radio is altered to the desired settings and sends, or prepares to receive, any possible signal.

2.1.1 SOFTWARE DEFINED RADIO (SDR)

Software defined radios are an integral part of a cognitive radio. SDRs allow a single radio to produce and receive a variety of signals with minimal hardware use, allowing a cognitive radio to automatically reconfigure a signal. The advantages of SDRs have resulted in widespread use by the military and for cell phone service, and, in

the long run, its proponents expect it to become the dominant technology in radio communications.

An SDR is a multi-band radio capable of supporting a wide range of air interfaces and protocols. It is, as the name would suggest, controlled by a software program that is used to modulate and demodulate a signal. Ideally, all functions of a radio would be processed by a general purpose processor, which is currently impractical for some signals for one reason or another (i.e. power consumption). Currently those signals are processed through application specific integrated circuits (ASICs), which are combined with field programmable gate arrays (FPGAs), digital signal processing (DSP) and general purpose processors to create the processing power of an SDR. As processor technology increases, the likelihood that all signals can be processed on general purpose processors is increasing [4]. This would make SDRs very feature-rich radios that are relatively inexpensive.

In a cognitive radio, the SDR is used to produce or receive the signal. The specification of the signal comes from an interface the SDR has with the cognitive engine.

2.1.2 COGNITIVE ENGINE

The cognitive engine is the brain of the cognitive radio. It observes the radio environment, plans on what actions to take, learns from past actions, and decides on the best action to take. It is in control of all aspects of the radio, from specifications to etiquette. It is designed to automatically produce the best signal dependent on the user's desires, and is sometimes built to emulate the actions an operator would take if he were in control of the radio.

There are many possible algorithms to implement a cognitive engine, partly due to the different definitions of cognitive radio, but also partly due to continuing research into the field. The cognitive engine in this project is based on biologically inspired genetic algorithms. How genetics algorithms work will be presented later in this chapter.

A basic state diagram of the cognitive cycle created by Mitola while at the Royal Institute of Technology is presented in Figure 2.1 [5]. Here it is shown that the first step in the process is observing the outside world. From that, a priority of the signal is established, which then, depending on that priority, is sent to either planning, deciding or acting. To improve upon the signal, learning is occurring during each step.

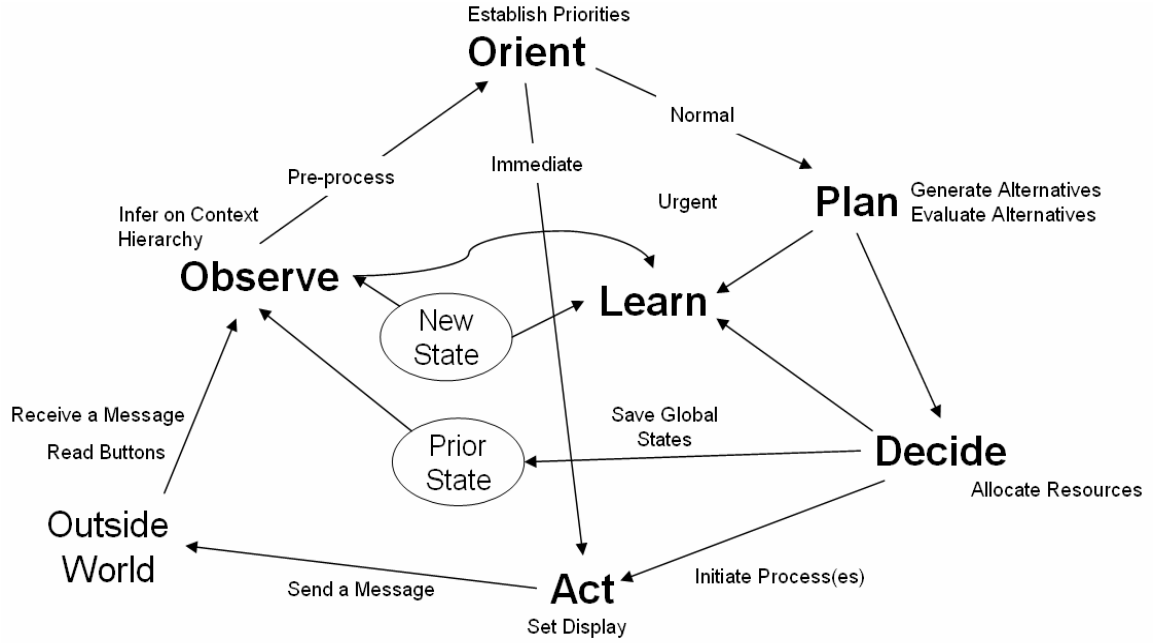

Figure 2.1: Cognitive Cycle

Again, the purpose of the cognitive engine is to provide the best quality of service (QoS) dependent on the user's guidelines. To do this, the radio must use sensors, or "meters", to receive information about the radio environment, decide upon the best signal, and alter the actuators to achieve the desired results. The actuators, or "knobs", determine the characteristics of the waveform. These can include, but not limited to those presented in Table 2.1 found in [1].

Symbol	Meaning
S	Signal Power
B	Bandwidth
R_s	Symbol Rate
Mod	Modulation Type
M	Modulation Order
PSF	Pulse Shape Filter Type
α, β	Roll-Off Factor for Root-Raised Cosine or Gaussian Filters

Table 2.1: List of Possible Knobs

On the other hand, meters allow the radio to understand its environment, and from such, can notice problems with signals and differences from the desired QoS. Some meters are present within the radio, but others can be calculated from an objective function using the values of the knobs as inputs. Some of the important meters displayed in [1] are presented in Table 2.2 along with the knobs that affect each meter. BER is the bit error rate, and the SNIR is the signal to noise-interference ratio.

Objective	Affecting Knobs
BER	$f_{BER}(S,N,I,B,R_s)$
SNIR	$f_{SNIR}(S,N,I)$
Data Rate	$f_{DR}(R_s,M)$
Occupied Bandwidth	$f_{OB}(R_s,M,\alpha)$
Spectral Efficiency	$f_{SE}(R_s,M,\alpha)$
Computational Complexity	$f_{CC}(f_{DR},f_{SE},f_{OB})$
Power	$f_P(f_{CC},f_{OB},f_{SNIR})$

Table 2.2: List of Possible Meters

It is noticeable from Table 2.2 that there are multiple meters that are affected by an individual knob. This demonstrates the difficulty of finding the best QoS, as altering one knob to improve upon the value of a meter will then alter the other meters that it also affects. This makes finding the exact QoS very challenging, which it is important to find the exact value because either a lower or a higher QoS is very undesirable. A lower QoS

is bad for obvious reasons, while the higher is unwanted because it uses resources that are unnecessary [1].

All cognitive engines need some tool that connects the meters to the knobs and implements learning into the engine. For this project, it is controlled by a genetic algorithm which reads the meters, finds new values for the knobs that would implement the best QoS, and turns the knobs accordingly.

2.1.3 GENETIC ALGORITHM

Genetic algorithms are a biologically inspired search technique intended to find an optimal solution to a problem [9]. They are considered a local search algorithm as it is generally an incomplete search. It has become a very popular search technique due to its flexibility allowing the algorithm to return a close to optimal solution in a minimal amount of time for a variety of problems. The algorithm has four main parts: initialization, selection, reproduction, and termination.

The genetic algorithm procedure begins by initializing a set of solutions, or individuals. This set is referred to as a population. The size of the population is generally dependent on the problem, (i.e. the amount of search space), but any size will work. The process of initialization can either be random or seeded with initial values that are more promising.

The next step is selection, where the best individuals from the current population are selected. To do this, each individual's fitness is evaluated in a fitness function. The fitness is a value rating how well the individual achieves a certain goal, or set of goals. The number of individuals selected in this step is dependent on the size of the population chosen earlier.

After selection occurs, the individuals selected previously are used to create a new generation of individuals in the reproduction step. The new generation is created through two processes called crossover and mutation. Crossover involves the individuals selected

before, now considered "parent" individuals, combining parts of the solution from one parent with parts of the solution from another parent to create a new solution, called the "child" individual. Each of the individuals are broken up into sectors, called chromosomes, where each chromosome is kept whole during the crossover, and combined with chromosomes from the other parent individual. All of the new child individuals now make up the current population. The new individuals typically will share many of the characteristics of their parents. Since the parents were some of the best individuals from the previous generation, the average fitness of the new generation will hopefully be higher.

In order to keep the individuals from being stagnant and maintain diversity, mutation can be added to the reproduction process. After a child individual has been created, a chance is taken, usually a small probability, that a small set of the child's data may be mutated into a new value. The amount of the child that is mutated is usually small, thus still keeping most of the parent's characteristics, but infuses the child with new information that may improve the results.

After the new generation is created, the process of selection and reproduction repeats until a terminating factor occurs. These terminating factors can be that an individual has been found that satisfactorily meets the criteria, that the average fitness of a population has become stagnant between generations, or just that it has run for a specified amount of time. Once termination has occurred, the best individual is then selected as the solution found.

The genetic algorithm is very similar to the theory of the evolution of species. It is a survival of the fittest practice, where the best of the best survive and pass on their traits to the next generation. This results in each new generation being more capable and better suited to solve the problem than the last.

For this project, the genetic algorithm is used to find the most optimal solution, or signal, dependent on the radio environment and the user's needs. The genetic algorithm

was chosen because it works very well with the structure of a radio. Just as genes are used to build traits, knobs are used in a radio to build the signal. Therefore, in this circumstance, the chromosomes are defined as the various knobs, and each of the chromosomes are set to the size that can represent all possible values of that knob. Figure 2.2 provides a sample chromosome where the different sizes of each chromosome is visible. This was demonstrated in [1].

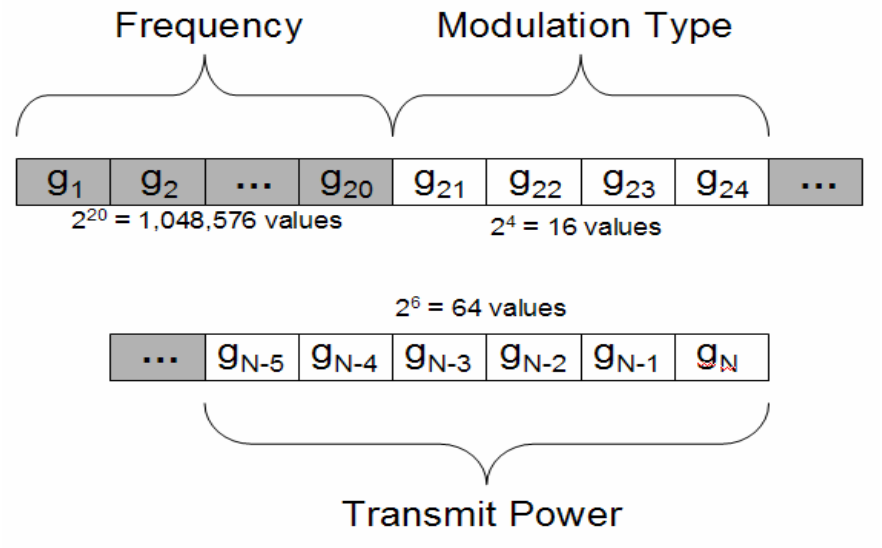

Figure 2.2: Sample Chromosome for Cognitive Radio

Genetic algorithms also provide the flexibility to allow for multi-objective optimization which is necessary in this situation, as the optimal signal needs to operate well among the entire range of a radio's dimensions, e.g. power, bit error rate, etc. Here, the genetic algorithm gives a weight to each dimension, which can change depending on the users needs. An example of this would be where a user needs a long battery life; the genetic algorithm would provide a higher weight to the power objective, to ensure the user received a longer battery life. The total fitness of an individual is the summation of all the fitness values of each dimension evaluated to its weight.

The use of genetic algorithms to obtain an optimal signal has shown to be very successful. It is able to efficiently find a signal that satisfies the user's needs within a given environment, and is able to deal with unexpected events. The challenge is that the

non-deterministic method of the algorithm means that the result can be different every time it is executed, thus making it difficult to test.

2.2 SOFTWARE TESTING

As software systems become increasingly larger and more complex, software testing has become more and more challenging, and thus more important. It is a very labor intensive process that can have a major effect on the bottom line of a company [10]. Currently, software testing accounts for approximately 50% of the cost in developing a software system [11]. Because of this, being able to automate, and quickly execute, testing algorithms has become very desirable as it can decrease testing times and development costs.

The purpose of software testing is to find any errors in the code through the detection of a fault or defect, or to demonstrate the absence of an error [12]. These errors can be created in each and every step in the software development process. Figure 2.3 demonstrates a development cycle for software showing where errors can occur and how the faults they produce can propagate to further sections of development. It is imperative that the code be tested during the development cycle, as it has been shown that programmers are only 25% efficient at finding their own errors [13].

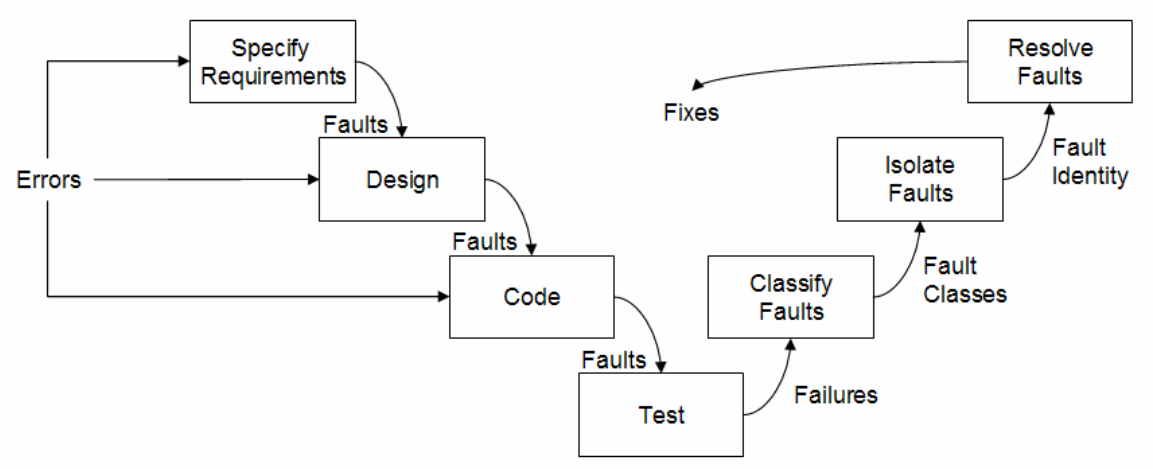

Figure 2.3: Software Development Cycle

There are many methods and algorithms used to test software. Many of these have come from hardware testing and verification, as these algorithms are already well known and well researched, and have been altered to better suite software programs. Software does add extra challenges that are not found in hardware systems, such as recursions, pointers, and dynamic memory allocations. They can also be much larger and much more complex than hardware systems.

All of the software testing approaches can be separated into two distinct groups: static and dynamic [14]. Static testing is any testing technique that does not involve executing the software. This can include anything from simply inspecting the code to symbolic execution and other formal methods. Dynamic testing, on the other hand, is any testing technique that executes the software. There are many types of dynamic testing, including path testing and data flow testing.

2.2.1 STATIC TESTING

The testing approach that will be presented in this paper is a dynamic execution of the software, so only a brief background to static methods will be presented.

Research in automated static testing began in the 1960's [15] from simple manual inspection. It has since developed into many types of tools with varying capabilities. Two of the more popular research areas in static testing are symbolic execution and model checking, a formal approach to verification.

Symbolic execution works by representing the program by some means. The representation is then traversed, not with values, but with symbols, until an output is obtained. Finally, the output is then presented in terms of the symbols placed at the input [16]. The program representation can be a type of flow-graph for the program. The flow-graph contains input statements, condition predicates, assignment statements, and outputs. The input symbols follow the flow-graph evaluating for each condition predicate and executing assignment statements. The results are outputs for every possible path in the program, and how the outputs relate to the inputs of the program. The output-

input relationship can be viewed to find if any outputs react in an unexpected way for all inputs.

Model checking is a formal approach where the program is abstracted into a finite state machine (FSM) [17]. Images or pre-images are then obtained about the FSM which allows a tester to prove whether or not a program, starting from a certain state, would ever be able to enter into an illegal state. The process of doing this has been well researched for hardware, but software characteristics make this process very different and much more challenging than hardware. The main positive characteristic of model checking is that it provides a complete 100% assurance that the error will not occur in the program.

2.2.2 DYNAMIC TESTING

Because of the challenges inherent in static testing, dynamic testing is a popular approach to testing software. The principle idea is that while running the program, a defect will be excited and then propagated to an output where the error will be observed. There are three parts to dynamic testing: data generation, coverage metric, and error evaluation [18]. Test data generation is the algorithm used to obtain the input data for the program, while the coverage metric is the measurement of how much of the program has been tested, and the error evaluation is how the detection of an error occurs.

The simplicity of dynamic testing has made it much easier to develop and execute, but its downsides are long test times and lack of a complete assurance the program is error free. For a dynamic test to provide a complete assurance, it would need to execute the program for all possible inputs. This can become very large very quickly, as the number of possible inputs to a program is $2^{(32*n)}$, where each input is a 32 bit integer and n is the number of inputs to the program, and that is only for programs with static inputs. Programs with a dynamic number of inputs will have an even greater selection. Because of the large number of possible inputs, it is impractical to execute all combinations, and thus the need to find a select number of inputs that best tests the entire program, which will reduce the testing times.

Two major classes of methods for dynamic testing exist. These are black-box testing and white-box testing [19]. The black-box approach, also called the functional approach, is where the implementation of the program is unknown to the testing program. The only information that the testing program has is the program specifications [20]. This method is used to check the functional correctness of a program. The white-box approach, also called the structural approach, is able to see the implementation of the program as well as know the program specifications [12]. This method is used to check the structural correctness of a program, and is used as the method for the approach presented in this paper.

2.2.2.1 Coverage Metric

One of the main focuses of software testing is to reduce the test times. This is done by reducing the number of test cases that evaluate the program. The number of test cases to run depends on the coverage metric. The coverage metric can range from all possible inputs, creating many test cases, to statement coverage, which says that every statement in the program has to be executed, needing a far smaller number of test cases.

The purpose of a coverage metric is to quantify how much of the program has been successfully tested, and when the test is complete. The types of coverage metrics possible depend on what type of test is to be run, whether it is black-box or white-box testing. In white-box testing, since the structure of the program under test is visible to the tester, a metric is used to measure how much of the structure has been executed. Popular metrics include path coverage, in which all paths are executed, branch coverage, in which all branches are evaluated, and statement coverage, in which all statements are executed. Black-box testing has different metrics, as it is not able to see the structure of the program, and focuses on executing the various functions of the program, rather than the structure of the program. An example of these metrics is state coverage, where the input domain is decomposed into sub-domains, and each sub-domain needs to be used to execute the program [21]. The testing algorithm proposed in this paper uses a form of branch coverage as the coverage criteria.

2.2.2.2 Data Generation

There are many different types of data generation algorithms, and much research has gone into this area of dynamic testing [20]. The sole purpose of data generation is to obtain 100% coverage of the program, which obviously depends on the coverage metric, in as few number of test cases as possible. This allows for a complete test in only a short period of time.

As stated before, there are many different methods of data generation, from a simple random data generation to very complex algorithms that search a program and deterministically create an input set. Which type of data generation method is best for a certain case can depend on many things. Of primary concern is which test is occurring, black-box or white-box testing [22]. Also, for white-box testing, simple programs, where all sections of the program are easily reached, can usually have simpler data generation methods. While more complex programs, where some sections of the code are unlikely to be reached, need more complex data generation methods that can better target these hard to reach sections.

2.2.2.3 Error Evaluation

After a test case has been created, the program under test is simulated with the values. The next step is to realize if a fault has been detected or not. This is a difficult task to automate, as being able to tell if values are not the "correct" values could mean implementing the same program that runs the software system for a comparison, but would thus possess the same faults. The location that the evaluation occurs depends on the type of test. A black-box testing method can only view the output of a program to determine an error, while white-box testing is able to view the values inside of a program to check for faults.

An approach to error evaluation that avoids the difficulty of automating the evaluation process is to manually check the values for correctness. This is a very simple approach, and while it does work, it is very tedious and errors can be missed through human error. A popular approach that does automate the process is the use of assertions.

Assertions are basically rules that are checked during or after a program's execution [19]. Once an assertion is violated, a flag is set to signal that an error has occurred.

Once an error has been found, the exact location of the fault in the program needs to be identified. This is easier in white-box testing, as where an assertion occurred can tip off the tester to where the fault is located. Otherwise, executing a stepping method of each line of the code again with the inputs that created the initial error should provide the location of the error.

2.3 TABU SEARCH

The tabu search is a metaheuristic optimization method that belongs to a class of local search techniques. It is generally considered to have been developed by Fred Glover, and has successfully been used in many fields, including telecommunications, molecular engineering, and financial analysis [23]. Its distinguishing feature over other local search methods is in the use of adaptive forms of memory.

Tabu search uses a neighborhood search procedure to move iteratively from one solution to another. In order to explore regions of the search space that would not be achievable from the initial solution, thus escaping local optimality, this method modifies the neighborhood structure as the search progresses. This continues until the stopping criterion has been reached. The final state produced is then the optimal solution selected for that particular problem.

2.3.1 SEARCH PROCEDURE

The basic procedure that constitutes a tabu search starts by initializing a solution to the problem. This solution usually is done randomly, but any seeding method will work as well. Every single alteration to the solution is considered a neighbor, and several neighbors are chosen. A quantitative measurement of how well each selected neighbor solves the problem is calculated in a fitness function. The one that is most fit among all

the neighbors is selected, and the process repeats until a stopping criterion has been reached.

An interesting change in the tabu search from classical search methods is that the transition from one solution to another is not always an improvement. This is demonstrated in Figure 2.4, which represents the weight, or fitness, of the solution during different iterations for an example that will be presented later in the memory structures. The purpose of allowing a new solution to be selected, even if it is less fit than the current solution is to prevent the solution from being trapped in a local minimum that may be far from the global minimum [2]. Thus the search is willing to take a step back, in order to take two steps forward and find the optimal solution.

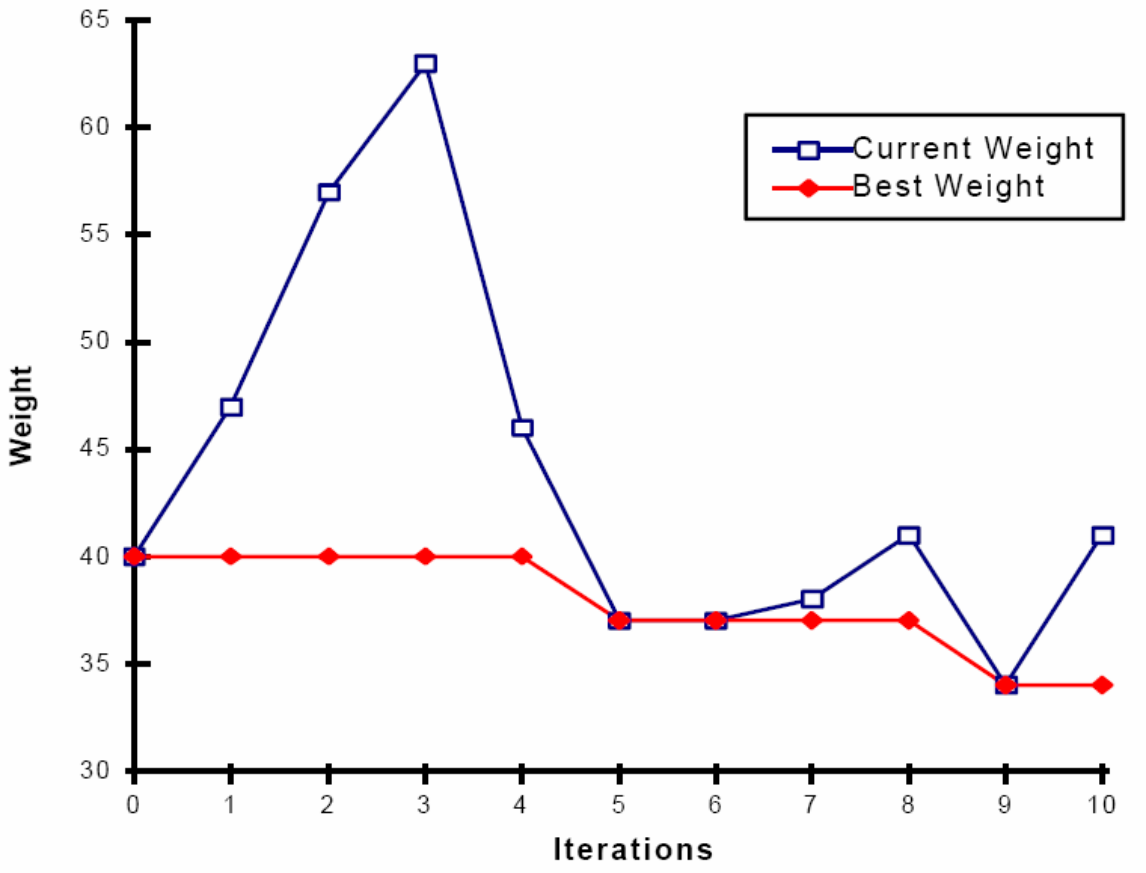

Figure 2.4: Fitness during Iterations [23]

To guide the search procedure into more desirable solutions quickly, tabu search implements methods of intensification and diversification [2]. Intensification strategies are used throughout the search procedure that modifies the rules to encourage moves that have been found to be historically good. Intensification strategies require a means for identifying optimal solution spaces [23]. Diversification, on the other hand, modifies the rules during the searching procedure so that areas, and especially exact solutions, that have already been looked at are not repeatedly used. This should prevent cycling, or at least keep it to a minimum which is a constant problem with various previous search techniques.

The ability to implement intensification and diversification into the tabu search is done by its use of memory structures.

2.3.2 MEMORY STRUCTURES

The memory structures used in the tabu search operate by reference to four principal dimensions, consisting of recency, frequency, quality, and influence [23]. The four dimensions can easily be applied to represent the rules of intensification and diversification, thus are stored in memory to be used throughout the search procedure to improve the intensification and diversification rules. The memory structures in which these are stored actually consist of two forms of memory, a short-term and a long-term memory, with each type of memory accompanied by it own special strategies.

2.3.2.1 Short-Term Memory

Short-term memory used in the tabu search stores information about recent past searches in order to exclude possible neighbor selections from occurring. This can have an effect on both the intensification and diversification of the search, by either eliminating neighbors that have already been searched or neighbors that appear to be bad solutions. The neighbors that have been excluded from the search are considered “tabu”, providing the namesake for this searching process. How a neighbor is eliminated as a search possibility depends on the problem being solved, but can be controlled by any of the four dimensions listed earlier.

A good example of this, from [23], can be seen in a minimum *k*-tree problem. Given the graph provided in Figure 2.5, create a tree with *k* edges, where the sum of the edges is minimum.

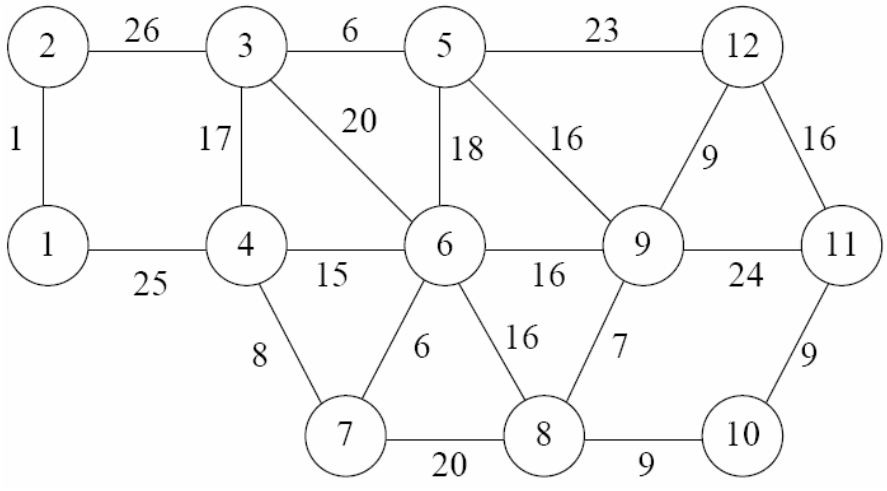

Figure 2.5: Graph Example

From a greedy algorithm, an initial *k*-tree solution, where *k* is four, is found consisting of the edges between nodes 2 - 1, 1 - 4, 4 -7, and 7 -6, which has a combined weight of 40. The iterative process that works for this problem is to remove one edge of the solution, and replace it with another. An example of how short-term memory could be used to exclude solutions from occurring is through the recency dimension. For this search problem, it is undesirable to change an edge of the tree right after it has already been altered, because this only provides for a minimal search of a tree with or without that edge. To alleviate the problem, edges that have been just removed can not be added back to the tree for two iterations, while nodes that have just been added to the tree can not be removed for one iteration. The process of placing and removing the edges, along with tabu valued for each edge during the search is presented in Figure 2.6.

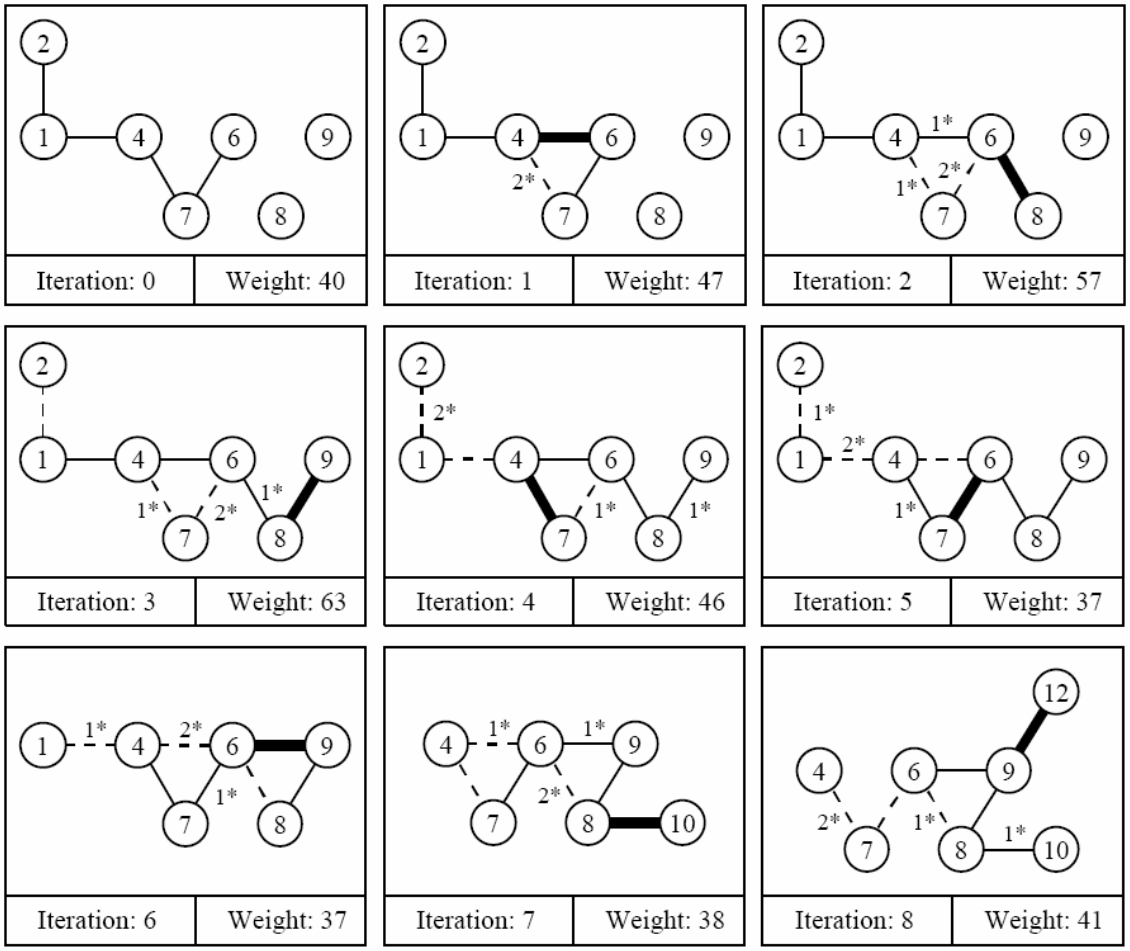

Figure 2.6: K-Tree Tabu Search Example [23]

In the example, dotted edges have been recently removed; the number next to the edge represents of the number of iterations that the edge is tabu and can not be used. Bold edges have been recently placed into the tree, and also have the tabu number next to it.

This is a very simple example, using only short term memory for a simple problem, but does provide a good description of how short-term memory is used to simplify the neighborhood representing solutions to be searched.

2.3.2.2 Long-Term Memory

Long-term memory is used in the tabu search to store local minimum solutions that have occurred during the search process. This is useful when the search gets stuck in certain areas, or there is a lack of improvement in fitness over an amount of time, and it is able to backtrack to previously good solutions. Figure 2.7 demonstrates the difference between what values are stored to short-term memory and what values are stored to long-term memory in a graph that provides the fitness of solutions for a section of iterations.

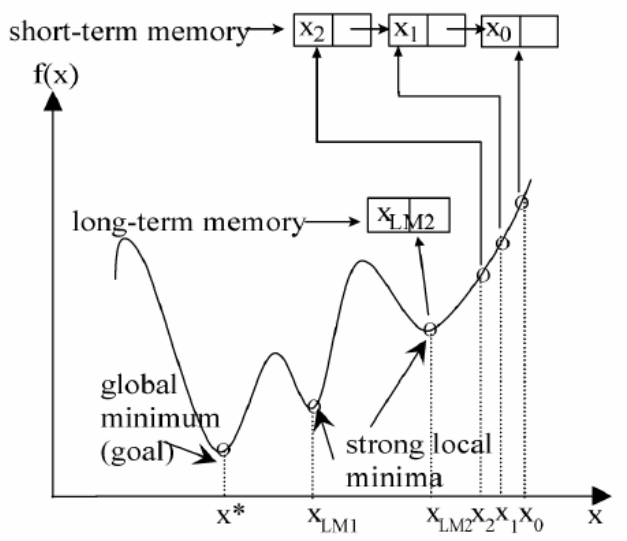

Figure 2.7: Short-Term & Long-Term Memory Usage [2]

Tabu search has been shown to be a very useful search technique to find an optimal solution to a problem. It primarily works best on problems where each individual part of the solution contributes to the overall effectiveness of the solution. This thesis presents the use of a tabu search to find inputs for a software testing program.

CHAPTER 3

AUTOMATED SOFTWARE TESTING PROGRAM BASED ON THE TABU SEARCH

This chapter proposes an automated software testing approach that is based on the tabu search. An in-depth description of the coverage metric, data generation, and error analysis will provide a complete look at how the process works. The testing program will be able to check for general errors within any software code. After the approach is presented, results and a comparison of the results to other attempts using this approach will be presented.

There are many factors to consider in building an effective automated software testing program, such as speed, quality, and ease of use. Dynamic testing methods have been around since the beginning of software testing, as this approach is the most obvious and simplistic way to test software. Currently, research is being focused on two separate areas of dynamic testing. The first is to represent the system by linear inequalities that are later solved [24], and then provides inputs to a program that will execute the code in the desired manor. The second is by using metaheuristic search techniques that view the testing process as a search or problem. The testing program then tries to find inputs that will satisfy the problem and obtain program coverage.

The algorithm presented in this thesis uses the tabu search, which is a metaheuristic search technique. Other, and maybe more well known, search techniques that have been applied to software testing are genetic algorithms [25][26] and simulated annealing [27]. Tabu search has been applied to software testing in only one published paper [3]. This paper, from the University of Oviedo, did a good job of showing the basic application to software testing and its potential, but the approach was hindered by a slow testing method, an inefficient search process, and by a lack of learning. The

approach presented here will increase the learning and efficiency of the search process from the previous paper, providing overall shorter test times.

3.1 COVERAGE METRIC

The selection of which coverage metric to use is a critical part of the software testing process. It selects how much of the program needs to be executed before the testing process is complete. For the tabu search, the coverage metric represents the goal that the search process is trying to reach.

Setting the coverage metric as the goal for the tabu search has some challenges in itself. The tabu search is designed to solve a single goal, and a coverage metric has many separate goals throughout the testing process. Furthermore, it is possible that fitness values for some of the goals would not be able to be evaluated during part of the search process. This would occur for sections of code that had yet to be executed, and thus, unable to be evaluated.

The authors of the previous approach [3] decided to set the coverage for the entire program as the single goal that possesses many sub-goals throughout the process. Figure 3.1 demonstrates how the goal is divided into sub-goals. It is easy to see this method being used for branch coverage, where each node is a branch and each edge provides a connection to branches that can only be executed within the section of code that is controlled by the previous branch. In this manner, the current goals of the process are nodes that have been reached, but not executed. The other nodes are viewed almost as if they did not exist, until they have been reached and can be evaluated. When every sub-goal has been achieved, the goal of the entire coverage metric has been completed.

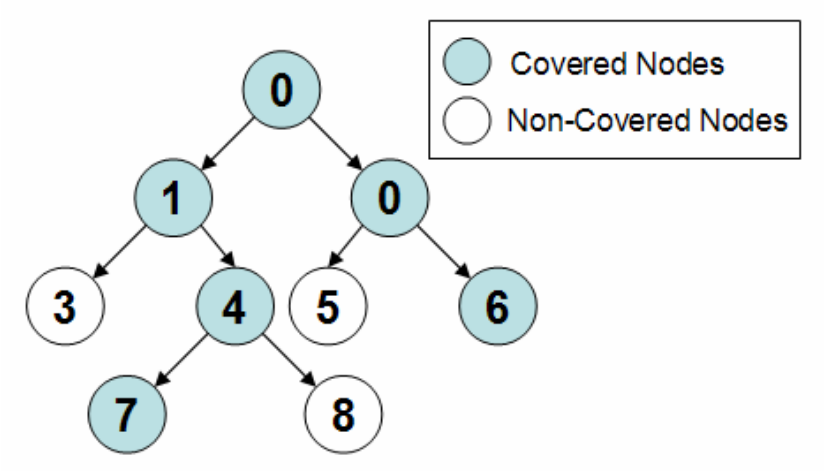

Figure 3.1: Sub-Goals of the Coverage Metric

Instead of using the sub-goal method as the authors from Oviedo did, each individual part of the coverage will be considered as a problem to solve for the purpose of this research. So, instead of running one tabu search to solve the entire coverage metric, many tabu searches are run to solve each individual item within the coverage metric. This change of what exactly is the goal of the tabu search has a small affect on covering the code, mostly due to the initialization structure being different. The goals set forth in this approach are based on the branch coverage metric.

3.1.1 Branch coverage

Branch coverage, in its basic form, checks to ensure that all branches have been executed for every condition in the code [24]. This is a very simplistic metric, but allows for strong coverage, as each branch carries out all possible results and thus every section of code is executed.

High-level languages, such as C/C++, allows for conditional statements with multiple conditions, e.g. (x AND y). When this is transformed into assembly language, the conditional statement is separated into two individual conditions, which is how it is modeled for this coverage metric. While the previous example, (x AND y), only provides two solutions, true and false, this metric checks for four solutions: false AND false, false AND true, true AND false, and true AND true. If there were three conditions,

instead of two, then the number of solutions checked for would be eight. This allows for the logical operator to be checked for correctness, e.g. if the logical AND, "&&", should have been a logical OR, "||". Once the conditional statements have been broken down, the individual conditions can be checked for correctness.

Conditions can be evaluated with either one variable, a Boolean expression, or two, a relational expression.

- !x; // (NOT x) Boolean expression
- x != y; //(x does not equal y) relational expression

In this approach, these two types of conditional statements are treated differently in order to improve the coverage. Boolean expressions are evaluated just as in the original branch coverage where the goals are to have the statement execute both the true and false branches. Accomplishing this will provide enough evidence about whether the statement is correct or should be revised.

Relational expressions are treated slightly different. The original branch coverage simply tries to execute both the true and false branches of the condition. This may skip some possible errors, as the example below demonstrates. To improve upon the coverage, a corner case test is added whose goal is to execute the conditional statement with both conditional variables being equal.

Applied (Erroneous) Expression:	(x < y)
Correct (Non-Erroneous) Expression:	(x <= y)

Branch Coverage:	Applied		Correct
x = 5; y = 10;	True	=	True
x = 10; y = 5;	False	=	False

No difference found between correct and incorrect expressions

BC w/ Corner Case:	Applied		Correct
x = 5; y = 10;	True	=	True
x = 10; y = 5;	False	=	False
x = 5; y = 5;	False	≠	True

A difference is found between the correct and incorrect expression, providing incorrect results. This method will provide different results for any incorrect expression being used.

With the application of the new goal, the entire coverage metric can be simplified by setting it to be independent of which condition is present. The three goals that are checked for are that the first variable is greater than the second variable, the first variable is less than the second, and the first variable is equal to the second. This new method is then capable of finding the difference between any relational expression, just as the previous example demonstrated the difference between two very similar, but different, conditional expressions.

3.1.2 FITNESS FUNCTION

A fitness function has to be created to evaluate the fitness of, or how close to the goal, a solution is. In this case, the goals of the program are to achieve a complete coverage of the program under test. To do this, the values of the variables at each condition are checked and the difference needed in those variables to alter the condition to the desired state is set as the fitness [3]. When the desired state is eventually reached, it possesses a needed change in value of zero, or even a negative value, which can be used to identify previously achieved goals. Every goal that has a positive fitness value has yet to be reached, with the lower the value, the closer the goal is to being reached. Even though a new test set may result in the same execution of a conditional statement, this approach is able to identify "better" results by measuring how close they are to reaching the goal.

The approach from Oviedo's only goal was to execute a conditional statement for both true and false [3]. Because of this, the fitness function was different for each conditional statement as presented in Table 3.1.

Element	TS Fitness Function
Boolean	0
x=y, x≠y	abs(x-y)
x<y, x≤y	y-x
x>y, x≥y	x-y
x∧y	Min(cost(x),cost(y))
x∨y	if x is TRUE and y is TRUE Min(cost(x),cost(y)) else Σ(cost(False c_i))
¬x	Negation is moved inwards and propagated over x

Table 3.1: Previous Fitness Function

Adding the corner case as another objective creates the same goal for every condition and thus simplifies the fitness function. The entire fitness function for each condition in this new method is presented in Table 3.2.

Element	TS Fitness Function		
	x < y	x = y	x > y
Boolean	0	0	0
Relational	y - x + 1	abs(x - y)	x - y + 1

Table 3.2: Tabu Search Fitness Function

To demonstrate that the new fitness function provides at least the same coverage as the basic branch coverage, the following example is provided in Table 3.3. This example shows some conditional statements and their results based on the new fitness function. It is noticeable that for every condition the new fitness function results in both true and false executions. This proves that the new fitness function will execute all the conditions with the same coverage as the branch coverage, but with the added support to find other errors that would not have been found, as shown in the previous section.

Condition	Less Than	Greater Than	Equal
x == y	FALSE	FALSE	TRUE
x != y	TRUE	TRUE	FALSE
x < y	TRUE	FALSE	FALSE
x <= y	TRUE	FALSE	TRUE
x > y	FALSE	TRUE	FALSE
x >= y	FALSE	TRUE	TRUE

Table 3.3: Fitness Function Example

The simplified fitness function does not specifically provide any advantage for the testing program with respect to run time, especially when considering the process is increasing the number of goals needed to be achieved. It does greatly improve the ease of applying the testing program to test sections of code, especially for larger sections, as well as increasing the likelihood of catching an error.

Once the coverage metric has been set and the fitness function created specifically for the coverage metric, the data generation part of the testing program is created in order to completely cover the program based on the terms of the coverage metric.

3.2 DATA GENERATION

The data generation section of the testing program follows the basic operation of the tabu search in order to create new input vectors. The process includes an initialization, neighborhood selection, learning, and finally a solution.

3.2.1 INITIALIZATION

The tabu search algorithm starts with an initial solution to the problem. This is the point where viewing the coverage metric as multiple problems, instead of one large problem, has an effect on the testing process. The tabu search initializes one solution for a problem, so having one input data set to initialize many conditions is very limiting. The initialized solution, while it may be good for one, or even a few, reachable conditions, may not be close to executing others. With highly controllable branches reachable at the beginning of the testing process, it is more effective to randomly initialize more test sets

than to try to execute these conditions from the tabu searching process. Thus, multiple test sets are initialized, just as if attempts were being made to solve multiple problems at the same time.

3.2.2 NEIGHBORHOOD SELECTION PROCESS

After initialization, it is time to begin the search by creating neighborhoods and finding optimal solutions to each problem. The process begins by selecting a suitable neighborhood, executing the neighborhood on the program under test, evaluating, choosing the new solution, and then repeating the process. This is done for each goal in the program.

Each goal throughout the program is evaluated separately, so only one goal is selected to be solved at a time. Every goal stores the current solution, which is based on the fitness function. The current solution stored for a particular goal, which was either decided upon through the initialization process or from a previous iteration, is the reference solution used to create the neighbors. The number of neighbors created from this selected solution is twice the number of inputs to the program. This technique consists of generating two neighbors for each input, one by increasing the value of the input and the other by decreasing the value of the input. This means that if the selected solution is $(v_1,v_2,\ldots,v_n)$, the values for all v_k that satisfy $k \neq i$ remain the same and two new values for v_i are produced. Thus, we have two new test data neighbors:

$(v_1,v_2,\ldots v_i+\lambda_1\ldots,v_n)$

$(v_1,v_2,\ldots v_i-\lambda_2\ldots,v_n)$

The variable, λ_x, changes dynamically during the search, and is decided upon through several factors, including limits on the value, the fitness of the current solution, and learning through past searches. Since each input has different results for these factors, λ is usually different for each and every neighbor. How the λ is evaluated will be provided later in this section.

So, for example, provided there is a simple four input section of code that we are testing, much like the test triangle program presented in Appendix A. A certain goal is chosen to try to execute, and the current solution for that is (5, 10, 7, 13). This solution has either been found through initialization or a previous search. The solution has many possible neighbors, but only eight are chosen. These could be:

- (10, 10, 7, 13)
- (3, 10, 7, 13)
- (5, 16, 7, 13)
- (5, 4, 7, 13)
- (5, 10, 19, 13)
- (5, 10, 5, 13)
- (5, 10, 7, 22)
- (5, 10, 7, 1)

Other tabu search approaches have created four new neighbors for each input, as the method for Oviedo [3]. This may reduce the number of iterations needed to find the goal, but it creates and executes a lot of unnecessary neighbors. Ultimately increasing the overall number of test sets needed to evaluate a program, and as a result, increases test times.

3.2.3 EXECUTION OF CODE

Instead of producing every neighbor prior to the execution process, each individual neighbor is created and executed before the next neighbor is made. This allows for the possibility to end the current iteration process before all neighbors have been executed. This will occur when an executed test set has a better fitness than the current solution. Once this happens, the iteration stops and the neighbor is set as the current solution for the next iteration. If no test sets are found that are better than the current solution, then the most fit between all of the neighbors is selected as the current solution for the next iteration.

The tabu search generally evaluates all neighbors all of the time. This is done so that the most learning can occur on each solution and the best neighbor will proceed to the next iteration. By halting the execution of the neighbors, it is possible that a more fit neighbor may have been missed. However, the advantage of greatly reducing the number of input vectors to the program far outweighs the possibility of skipping better neighbors. This is especially true when testing software, as the program run times may be relatively large as compared to other tasks that the tabu search has been previously used to solve.

Even though only one goal is targeted at a time, all unsatisfied goals are evaluated for each test. An input vector that may not correctly evaluate the targeted goal, may satisfy the conditions of the other goals. This is an obvious approach but very important in reducing the number of input vectors.

So, with the example from the previous section, a neighbor, such as (10, 10, 7, 13), will execute the code. All unsatisfied conditions are evaluated. If the current goal that is targeted is improved, (10, 10, 7, 13) is set as the current solution, else another randomly chosen neighbor, such as (5, 10, 5, 13), is executed. If all values are executed and none improve upon the current solution for the targeted goal, the most fit of the neighbors is set as the current solution.

3.2.4 SEARCH SPACE

The search space represents the possible inputs and their neighbors for the program under test. As a graph, each node corresponds to a single possible input to the program. If the nodes are only different by one input variable, as is a neighbor of the input vector, an edge connects the two nodes. This creates an extremely large and complex graph. For even a simple program with four, 32-bit inputs, there are $2^{(32*4)}$, or $3.4x10^{38}$, nodes, and many more edges.

The tabu search process allows various nodes or edges to be set to a tabu state to limit the search. With software program search spaces, the graph is initially so large that setting a node or edge to tabu has a negligible effect on shrinking the size of the graph.

The Oviedo's approach [3] noted this, but also pointed out that "avoiding repeating the best test would be to avoid repeating many tests candidates" [3]. This is intended to show that setting the previous solution as tabu does not just limit that one state from occurring again, but also prevents all the neighbor states that were evaluated from that state from also occurring again.

While this method of setting individual states as tabu does improve the searching process, it is still very weak as it only affects a very small amount of the possible states. To improve the power of the tabu process, the states have been partitioned, or compacted, for this research. This has been shown to reduce test times and improve learning in software testing [28][29]. Partitioning groups many of the input vectors into one state, so rules that affect one state, control many vector sets. Since the input vectors are grouped together, it is impossible to have tabu rules that explicitly affect a single input vector. Instead, the rules pertain to attributes that represent each state. The attribute that we have found that works the best is the location relative to the current solution. A graph of this state partitioning is provided in Figure 3.2. The advantage of this approach is that it resembles the neighborhood search and it is easy to draw cause and effect from the search process into rules for the state portioning.

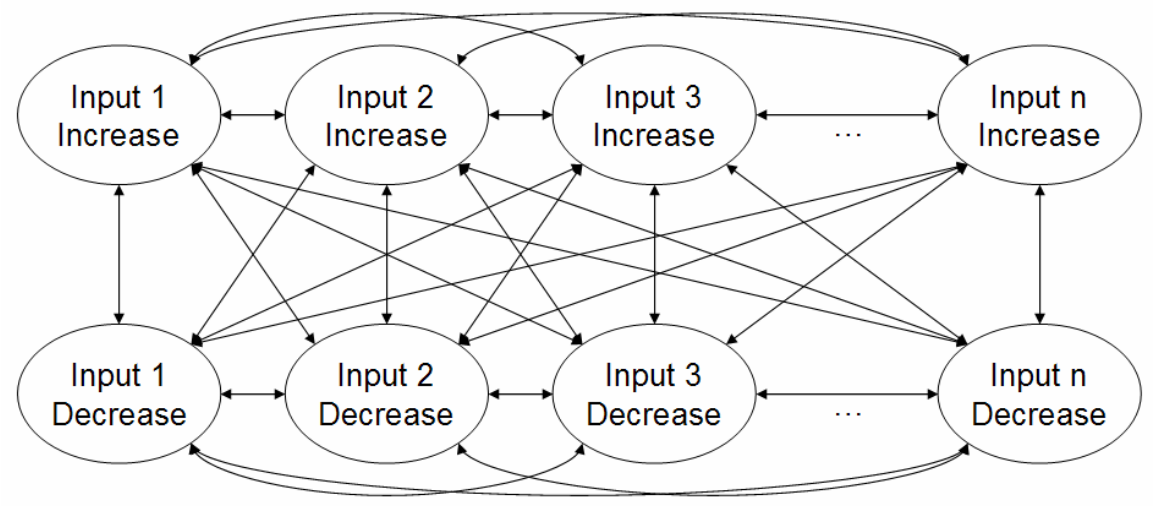

Figure 3.2: State Partitioning

The state portioning improves the tabu search for software testing immensely. It allows one of the strengths of the tabu search process to be applicable to the large state spaces found with software testing. To take advantage of this, rules must be properly set to force states into being tabu and limiting the search.

3.2.5 RULES

The rules set forth are intended to limit the search from testing areas of the search space that are expected to hold less fit solutions. This can be related to the intensification ideology [2] of the tabu search by only searching areas that are expected to possess strong solutions. The dimensions used in this case are quality and influence [23]. The rules are based on the results of the execution of the neighbor solutions. They are set only to a small number of iterations, as the affected states represent different input vectors for different solutions. There are four possible results that can occur when a new neighbor is executed.

- The fitness of the neighbor is better than the original solution. It is proved that the variable changed has an effect on the goal, and it is believed that changing this variable more can reach the goal. No states are set to tabu. The neighbor that found the better solution is set to be the first neighbor executed in the next iteration and the opposite neighbor, the increase to decrease or vise versa, is set to execute second if the first neighbor does not improve the fitness again.
- The fitness of the neighbor is worse than the original solution. It is proved that the variable changed has an effect on the goal and it is believed that moving in the opposite direction may find the goal. The effective state is set to tabu for one iteration. If the opposite neighbor has not been executed in this iteration, it is set to be the next neighbor to be executed.
- The fitness of the neighbor is the same as the original solution. It is believed that the variable changed has no effect on the goal. The effective neighbor is set to tabu for three iterations, and even the opposite neighbor is set to tabu for one iteration.

- The fitness of the neighbor is not evaluated. This occurs when the goal branch is nested within other branches, and the path to the goal branch is not executed. Thus, it is proven that the variable changed has an effect on some previous branch which contains the goal. The effective neighbor is set to tabu for one iteration.

To demonstrate how these rules are applied, we will continue to use the example from the previous sections. Given the current solution, (5, 10, 7, 13), has a fitness value of 10 and the eight neighbors presented in section 3.2.2. One of the neighbors is then executed among the program under test, let's say neighbor (5, 10, 5, 13), which is a decrement on the third input variable. The following are the possible results and the rules being applied to them:

- If the neighbor obtains a better fitness value, such as 6, then the iteration ends, and the current neighbor is selected as the current solution. In the next iteration, the first neighbor to be executed will decrement the third input variable, a possible input is (5, 10, 2, 13). If this neighbor does not improve the results, the next neighbor executed will increase the third input variable, but not to the point of the original solution. This could result in a vector (5, 10, 4, 13).
- If the neighbor obtains a worse fitness value, such as 14, then the current partition of decrementing the third input variable is set to tabu for one iteration. If the opposite neighor, in this case (5, 10, 19, 13) has not been executed, it is set to be the next neighbor to run.
- If the neighbor obtains the same fitness value, 10, then the current partition of decrementing the third input variable is set to tabu for three iterations, and the opposite partition of increasing the third variable is set to tabu for one iteration.
- If the neighbor does not obtain a fitness value for the goal, the then current partition of decrementing the third input variable is set to tabu for one iteration.

These rules provide a strong result due to their simplicity and accuracy in quality evaluation. Even though these rules provide very accurate results, it is possible to incorrectly estimate that a state contains only bad solutions. States are only tabu for a small number of iterations, so that after the set number, the state can then be searched again, hopefully finding better results.

3.2.6 LEARNING

The learning referred to in this section refers to the ability to make a more educated guess at which neighbor would be best to pursue. Simply put, it is the algorithms used to solve for λ, the change in value between the solution and its neighbor. This approach applies three aspects to control the value of λ: the current fitness, the bounds of the search, and the results from previous searches. The paper from Oviedo did not include a consideration of the results from the previous searches [3].

The idea behind using the current fitness aspect is to incrementally get closer to the goal. If the current solution is only one value away from reaching the goal, it is not in the best interest of the searching procedure to completely change the solution by wildly altering the inputs. As the solutions become closer to the goal over a number of iterations, the steps size taken to the neighbors becomes increasingly smaller.

The use of the bounds of the search in order to calculate the new neighbors is an obvious method. The new neighbors can not be illegal inputs, so if the current solution is near a bound, yet far away from the goal, the step to the neighbor in the direction toward the bound will have to be small in order to stay legal.

The new aspect employed by this research is learning how the changes from the past altered the fitness of the goal and applying that knowledge to the new neighbors. The idea behind this change is that when a neighbor is created to solve a goal, it is likely to be follow a program path similar to that of the current solution, and thus be executed by many of the same statements. If a step is made in a certain direction that gets a quarter better fitness, the next step in that direction will be three times the previous. As long as

the steps keep improving the signal, the new learned value will average with the previously learned values to calculate the next step. This applies more weight to the recent step as it is expected to be closer to the correct next step.

Continuing with the example from the previous sections, given is that the original solution, (5, 10, 7, 13), possessed a fitness value of 12 and an executed neighbor (10, 10, 7, 13) has a fitness value 8. The learning takes into account that increasing the first variable by 5, improves the fitness value by 4. Since the fitness still needs to be improved by 8, the next step will increase the first input variable by 10, in proportion to the increase needed in fitness, providing an input vector of (20, 10, 7, 13). If this still does not satisfy the goal, and provides a fitness value of 7. Then the current proportion of 5 input values to 4 fitness values is averaged with 10 input values with 1 fitness value, to make 15 input values to 5 fitness values, or 3 to 1. With 7 fitness values left, the next jump will be 21, or an input vector of (41, 10, 5, 13). This continues until a satisfying solution is found.

The use of a variable λ, which depends on its environment and past search history, produces a far more accurate, less random, search procedure. This can result in generating fewer iterations before the goal is reached, resulting in far fewer test vectors to be run and overall shorter test times.

3.2.7 TREATEMENT OF UNREACHED BRANCHES

There will be times where a goal branch is not being reached by the tabu search. This could be either because it is impossible to actually reach the branch, or that it is simply difficult to reach and the search technique has not been able to find it. The real difficulty is that if a branch does not execute in a certain way, other branches may not be able to be reached as they are contained in the previous branch's section of code.

A parameter has been defined in order to keep the search from getting stuck on these types of branches. This is the number of iterations that can occur from a solution without finding a more fit solution. If this parameter is ever reached, the program

backtracks to the previous best solution, and if that solution has been backtracked to previously, it backtracks one step further, or simply to the previous solution stored in the long term memory.

It is at this point where the long term memory becomes useful. Every time a solution is a local minimum along the fitness domain, being more fit than the previous and the next solution, it is stored into the long term memory. When backtracks are needed, such as when the stopping parameter is reached, the solutions stored in the long term memory are used to effectively backtrack to that solution and the searching process begins again.

If the case happens that there is not a solution to which to backtrack, the goal is skipped and pointed out to the operator when the program is complete. The operator can then rerun the search process, only looking for that specific goal. The search can also be seeded, hoping that will help find the solution, or it may be determined that the goal is unreachable by any possible input vector.

It is possible that some goals are just not reachable. It even occurs a few times for the code that was used to test this approach, which is provided in Appendix A. If this occurs, the operator can very easily set the goal as already being achieved, and the search process will skip over this. Adding the unreachable branches to the search can be done before even the first search has occurred, as long as the operator is sure that it is unreachable.

3.3 ERROR ANALYSIS

Once the software test program is creating data for the program under test, it is then important for the test program to be able to identify when an error has occurred and to calculate the amount of the program that has been covered. As stated previously, identifying errors is a difficult task, as the ability to find if values are incorrect requires the same software that has the error. One way to locate errors without requiring the same

executable software is through assertions, which work well for testing programs of the manner on which this research is focused.

Assertions are rules placed into code that check the values of variables in the program relational to another variable or a constant [19]. The rules represent areas that the variables should never be in and if any is ever violated, it alerts that an error in the code has been found. It is sometimes difficult to realize which rules can be placed into the code to ensure errors are found. For cognitive radios, the purpose of testing is to guarantee that that the waveform produced is legal, which can be easily turned into assertions. These assertions can be to check for legality and validity in the signals, check for accurate crossover and mutation, check for correct sorting, and check the fitness evaluation of the signal.

An example demonstrating the use of assertions would be for ensuring the validity of a signal. This assertion checks to see if all parts of a signal have been initialized and all segments are within possible ranges. If an error is located in assigning the signal, it is possible that the signal could possess invalid information and not be a producible by the hardware.

Assertions allow checking for errors inside the program, as some errors may be executed in the program and the erroneous values not propagate to an output where it is visible to the operator. It is still important for the operator to check output values for possible errors that have been propagated to a visible output. Both of these techniques should provide for the assurance that if an error is in the program and executed by the test vectors, that it will be noticed by the testing program or operator so that the error can be fixed.

3.4 RESULTS

It is very difficult to evaluate the quality of a software testing program. This is partially because there are so many different types of software programs that need testing.

When evaluating general software testing programs, certain algorithms may work well for some types of programs and not for others. Another challenge is that there are not a set of standardized programs that can be used to test software testing programs. This is done for hardware systems, as there are many standardized circuits that allow an algorithm to be evaluated and compared against other algorithms.

The previous approach using the tabu search to implement software testing did not provide any results [3], possibly due to the two reasons cited that create difficulties in evaluating approaches. Therefore, in order to compare this new approach with the previous one, a program needs to be written for these algorithms to be used upon, and the previous approach has to be built so that the results from that algorithm can be evaluated.

One standardized function that has been used in the past to evaluate testing program is the triangle function. This program takes three numbers as an input and evaluates whether it can be a triangle, and if so, what type of triangle it is, be it equilateral, isosceles, etc. It is a popular function to use since it has many branches, with some easy to reach and some difficult to reach, as they are deep inside other branches. However, it is not a perfect program to test, as it is very simple, very short, and does not possess any mathematical functions in it.

The triangle program was altered so that it would contain loops, one of the structures in software that make it difficult to test. This program is presented in Appendix A. The approach from Oviedo was then built with the available information. Obviously, since the exact equations and precise details used by this method are not presented in their paper, the re-creation of their method is not the exact same as their built approach. The information provided from their paper is done exactly the same in the built approach, while lacking information was done with creating the best results in mind. Table 3.4 presents the number of test generations needed, as well the average run-time, needed for both tabu search approaches, the one presented in this thesis and the Oviedo's approach. A completely random search which is applied to the triangle function is also

included. Since all the methods listed are a non-deterministic approach, all results are averages over 100 tests.

Triangle Function			
Test	Results		
	# Test Vectors	Test Times (s)	Coverage %
Completely Random	33108	0.012	88
Oviedo's Approach [3]	414	0.002	88
Presented Approach	41	0.0004	88

Table 3.4: Results on the Triangle Function

The previous results are slightly biased toward the tabu search approach presented in this thesis, since there are no mathematical operations in the test program. This skews the results because the approach presented here starts off by creating neighbors that are the same distance from the solution as the goal is, which, since there are not mathematical operations, is the exact change needed at the inputs. To counteract this, mathematical operations were applied to the input variables at the beginning of the program and the results were evaluated again. These are presented in Table 3.5.

Triangle Function w/ Mathematical Operations			
Test	Results		
	# Test Vectors	Test Times (s)	Coverage %
Completely Random	34057	0.012	88
Oviedo's Approach [3]	503	0.003	88
Presented Approach	133	0.001	88

Table 3.5: Results on the Triangle Function w/ Mathematical Operations

The tabu search presented in this thesis is significantly better than the previous approach for both variations of the test program. It is noticeable that the improvement is less than results presented in Table 3.5, which better reflects real world results, yet still shows a large improvement over the previous method. All methods obtained 88% coverage of the code. This is due to several unreachable branches. 88% of the branches is the total number of reachable branches in the code.

Now that the software testing program presented in this thesis has shown to be an effective approach, it is applied to test its main objective, the cognitive radio code. Since the code is so large and complex, certain sections are modularized and tested individually. To show the ability of testing the individual modules of the cognitive engine, Table 3.6 displays the testing results for the selection module in the cognitive engine. This module takes in two signals, and selects the signal with the better fitness. To allow for the optimal searching procedure, each knob of the signal is defined as the input for the program under test. Since the fitness of each signal is needed to be obtained, the evaluation module is placed before the selection module. Again, all results are averaged over 100 tests. A list of all modules in the cognitive engine that have been tested is presented in Appendix B.

Selection Module within Cognitive Engine			
Test	**Results**		
	# Test Vectors	**Test Times (s)**	**Coverage %**
Completely Random	38	0.002	100
Oviedo's Approach [3]	88	0.005	100
Presented Approach	14	0.001	100

Table 3.6: Results on the Selection Module within the Cognitive Engine

The results display the tabu searching algorithm presented in this thesis requires fewer input vectors and a shorter runtime than the other methods, while all tests covered 100% of the code. The runtime of each vector for the selection module is longer than the triangle program, which reduces the searching algorithm overhead compared to execution time. The random generation of input vectors actually performed better than the tabu search from Oviedo. This is due to no conditional statements being deeply embedded and hard to reach, as well as having many inputs and thus many neighbors for the Oviedo approach to execute. Finally, there are random values that are used in the selection module as a tie breaker. We kept these variables as random values, as opposed to another input variable controlled by the data generation algorithms, because they have a high chance of evaluating the conditions they are used in for both true and false.

Since the program under test is modularized, the software testing program is needed to test many sections of codes separately. This brings in one of the main advantages of the approach presented in this thesis, its ease of use. The program under test does not require large changes to the code, only the ability to check the coverage metric and the addition of assertions. This can be done by placing functions into the code at the necessary spots. Also, the algorithm does not need to be altered for different types of software programs, thus requiring few alterations to the testing program from testing one function to another.

It has been shown that applying the tabu search to software testing can create a testing algorithm that covers the program under test in a relatively few number of test cases. The ease of use and the low run times make this approach a very practical method to test software. It provides a quick method to ensure all the goals necessary to achieve a complete coverage of a program are met while only needing a minimal amount of setup time. This approach works for both large and small software programs and is highly flexible so that any necessary changes can be made to fit any program under test.

CHAPTER 4

BUILT-IN SELF-TEST FOR COGNITIVE RADIO

This chapter proposes a built-in self-test (BIST) for a cognitive radio. It provides the need and the details for the two part implementation. The first of these parts is a mask that is embedded into the cognitive engine to check that the signal being decided upon is legal. The second is a checker in the application program interface (API) that ensures that the production of the signal is correct. Finally, this chapter will prove that any signal developed by the cognitive radio is legal with respect to the rules it has been provided.

The previous chapter's work presents a general software testing program using the tabu search. This can be used to test any code segment that runs in the cognitive radio, but it is not perfect. Since it is a simulation-based testing program, and not all inputs are simulated, there is still the slight possibility that there are errors in the code. Also, this type of general testing focuses more on finding structural errors, as opposed to functional errors. Because of this, applying only the testing program to a code can not provide a 100% guarantee that the cognitive radio will never produce an illegal signal. In order to achieve a 100% success rate, using a simple BIST that continuously checks the cognitive functions to ensure correctness is proposed.

Built-in self-tests have been used in hardware components with much success [30]. They are built into the design of a circuit and provide the ability to test itself. That is the same idea that is being implemented in this study, only this time with software. The desired goal is to have the cognitive radio continuously inspecting itself with respect to rules put forth by the FCC in order to guarantee that the produced signal will be legal. It is also important to focus on the computational cost of the BIST, as the cognitive radio is very computationally intensive alone, so any additional processing work will result in a performance loss for the entire system.

There are two main sections of code that work to produce the signal, thus requiring two separate BIST implementations. The first is the cognitive engine, which selects which signal to use. The BIST for the cognitive engine needs to ensure that the signal that has been decided upon is legal and valid. The second is the API for the radio. This takes the selected signal from the cognitive engine and modulates it into a signal that can be produced by the hardware. This BIST will check to see if the waveform produces the same signal that was selected upon in the cognitive engine.

4.1 POLICY MASK

The mask acts like a filter, checking the signal for legality and only allowing those that pass to go through. Since we are dealing with radio communications where the term filter means something completely different, the implementation of this is referred to as the mask. It contains the rules set forth by the FCC as to what power a signal can be at certain frequencies [31]. The mask is used is the rules for UWB, which has been selected as an example. This same built-in self-test can be applied to public safety communications but would not posses a pre-built mask.

4.1.1 UWB MASK FOR COMMUNICATION SYSTEMS

The mask that is provided by the FCC is a very simple table. It lists ranges of frequencies along with the maximum effective isotropic radiated power (EIRP) in dBm allowed for each frequency range. A sample table that corresponds to a hand held UWB system is provided in Table 4.1, while the graph representing the table is presented in Figure 4.1. The FCC document *UWB Operation for Communications and Measurement Systems* [31] comments that the hand held communication devices are expected to be popular, and so they "recognize that the greatest concerns of interference in the record were centered about the potential for uncontrolled proliferation of these devices. Therefore, out of an abundance of caution the limits that we are adopting here are the most stringent for UWB operation." [31]

Frequency in MHz	EIRP in dBm
960-1610	-75.3
1610-1900	-63.3
1900-3100	-61.3
3100-10600	-41.3
Above 10600	-61.3

Table 4.1: Hand Held UWB Emissions Mask Table [31]

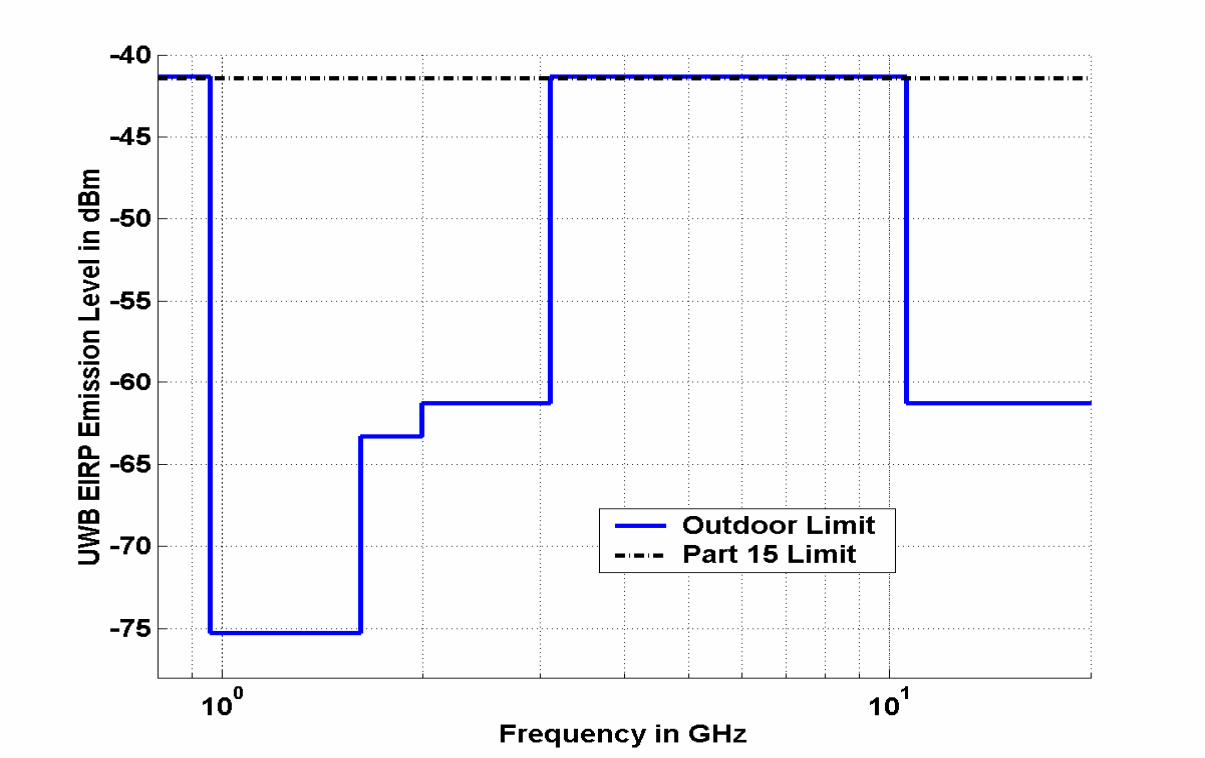

Figure 4.1: Hand Held UWB Emissions Mask Graph [31]

4.1.2 Legality focus

The primary purpose of implementing the mask into the cognitive engine is to ensure that the signal it selects will be legal and valid. The most simplistic way to accomplish this is to check the final selected signal and if it is legal, allow it to pass through, or else run the cognitive engine again to select another signal. The downside to doing this is that the cognitive engine takes a large amount of computational power and thus time to run. By having the program run again, the runtime needed until a solution is found is double an already long time.

Since the cognitive engine uses a genetic algorithm, it may be possible to select the second best signal in the final generation if the most fit signal is illegal. This will work, except in the case when every signal in the final generation is illegal. This is a real possibility since certain fitness aspects of the signal are in competition with the legality rules, e.g. adding extra power is the easiest way to improve the strength of the signal, but this intrinsically makes it more likely to be illegal.

In order to avoid having every signal in the final generation be illegal, and thus having no signal to select from, the mask will be embedded into the genetic algorithm. The mask will then be able to test not only the final generation's signals, but all signals that are generated throughout the algorithm. The mask can then alter the fitness of the signals so that a legal signal remains the final generation, which can be selected as the signal to produce.

4.1.3 QUALITY FOCUS

Along with ensuring that the selected signal is legal, the mask also serves the purpose of improving the quality of the selected signal. The idea behind this is that the mask will help the genetic algorithm focus on high quality results that are also legal. Without the mask being embedded into the genetic algorithm, many of the final generation's signals could be illegal. This may leave only a small number of signals that can be selected from, which may or may not be optimal as a solution. Instead, the mask can check every signal along the algorithm and focus in on the most fit, legal signals that will produce the best results

There are two factors that need to be observed with the focus on the quality of the selected signal. These are diversification and intensification. Even though we desire the algorithm to intensify its search on the strong and legal signals, we do not want the algorithm to lose its strength of searching through diversified results. By deleting certain individual signals during the process, we are limiting the search for the best signal and

possibly deleting good information, that when combined with parts of another signal, produces a very strong result.

It is possible to demonstrate this dilemma through an example using the information provided in Table 4.1 and Figure 4.1 earlier in the chapter. Given a generation that produces two signals, one at 3GHz with an EIRP of -50dBm and another at 3.2GHz with EIRP of -70dBm. The 3GHz signal is illegal under the FCC's hand held UWB emission's mask. If this signal were to be deleted because it is illegal, all the information it possess will be lost to future generations. Instead, if its information was able to propagate to the next generation, these two signals could combine to form a new signal that is 3.2GHz with an EIRP of -50dBm. This new signal is completely legal and may possibly be more fit than either of the two previous signals. Because of this possibility, it is not desirable to delete all illegal signals, as it may cause the removal of information that may be useful for future generations.

4.1.4 SIGNAL MODELING

The cognitive engine only knows the signals based on their specifications, e.g. power, center frequency, bandwidth, etc. The specifications then create a signal that forms a very complex shape in the frequency domain. The best way to describe this signal is as a downward facing parabola, with smaller roll-off side lobes on the sides of the main shape. This shape is shown in Figure 4.2. The main shape is centered at the frequency of the signal, the height is based on the power of the signal, the bandwidth deals with the width, and the other factors have various effects on how the shape is produced.

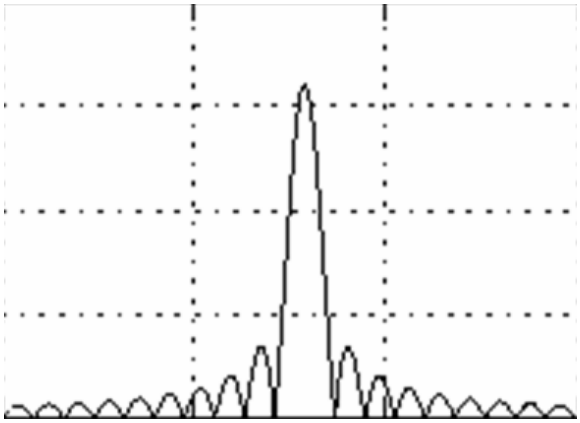

Figure 4.2: Signal Shape in the Frequency Domain [32]

To confirm if the signal is legal, the signal is placed over the mask to check for any overlap of the signal into an illegal section. This means that the signal has to be converted into the shape presented in Figure 4.2. Creating this shape from the signal specifications is a very computationally intensive process. This process needs to occur for each signal being tested throughout the genetic algorithm, which takes place many times in the cognitive engine.

The genetic algorithm already requires a large amount of computational power to run the selection process, so it is imperative that the signal modeling is as simple as possible. In order to accomplish this, the signal is modeled as a rectangle. A sample of how this modeling looks around the exact signal is shown in Figure 4.3.

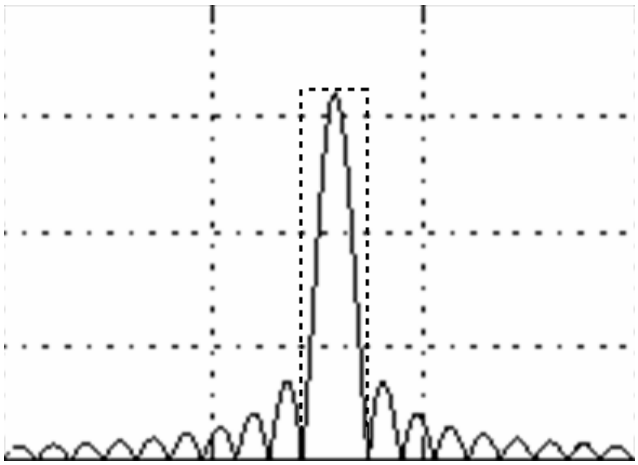

Figure 4.3: Modeled Signal Rectangle

The height of the rectangle is the exact same as the signal, with the width being dependent on which signal is being produced and the bandwidth or symbol rate of the signal. For example, an analog AM signal that uses a double sided band has a bandwidth based on the baseband bandwidth, while a digital PSF signal uses the symbol rate to evaluate the bandwidth. The formulas to calculate both of these bandwidths are presented:

- PSF: $BW = 2 \times 1.10 \times (SR) \times (1 + \alpha)$;
- AM-DSB: $BW_{BP} = 2 \times 1.10 \times BW_{BB}$;

The rectangle completely overlaps the entire main lobe. This way, if any part of the main lobe is illegal, the modeling rectangle is also illegal. The rectangle does extend past the lobe, which can then produce false failures. The likelihood of a false failure is small, as the amount of overlap is minimal compared to the ranges of frequency in a mask. An example of this is shown in Figure 4.4 with a modeled signal placed onto the mask from Figure 4.1. The signal used in this example is provided with a bandwidth of 40kHz, which represents the bandwidth of an average signal that the cognitive radio may produce. Each mark along the x-axis is one gigahertz.

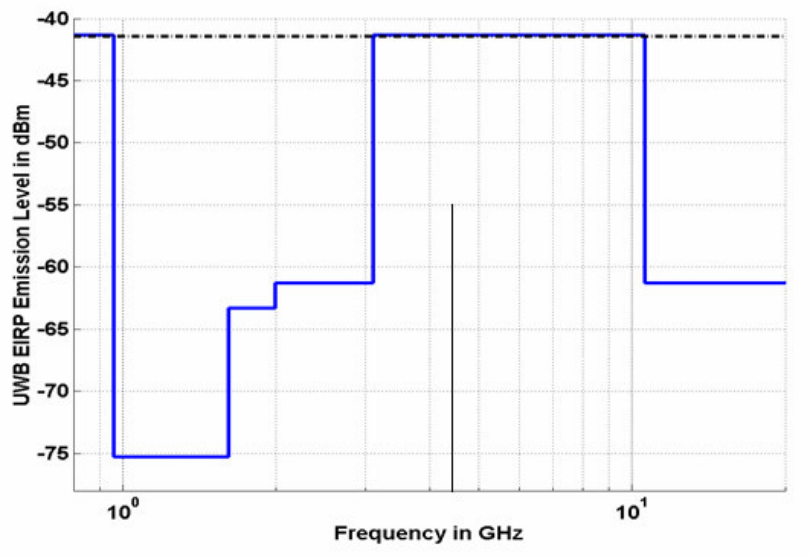

Figure 4.4: Modeled Signal Overlaying Mask

The actual signal produced will also have smaller side lobes. These lobes have much less power than the main lobe and will be legal as long as the main lobe is legal. Because of this, the side lobes are not taken into consideration while checking the signal for legality.

4.1.5 GENETIC ALGORITHM

The cognitive engine is controlled by a genetic algorithm that produces several signals throughout the generation process. The mask is embedded into the genetic algorithm in order to have an effect on the individual signals inside the engine. This is directed to ensure a high quality, legal output.

Without the mask in place, on average 24% of the signals produced in the genetic algorithm are illegal with respect to the FCC hand held UWB emissions mask. The process is non-deterministic and therefore each case will have a different number of illegal signals. From running the cognitive engine a number of times, the range of illegal signals has always been between 18% and 31% of the total number of signals produced throughout the generations.

Throughout the evaluation and testing process, the selected final signal from the output of the cognitive radio has never been shown to be an illegal signal. This is probably due to the various fitness functions that are improved by a less powerful signal, e.g. power consumption. Even though the final signal has yet to be shown as illegal through many trials, this does not prove that the all selected signals will be legal, so a final legality check still needs to occur to ensure that this final signal will always be legal.

4.1.6 IMPLEMENTATION

Besides computational costs and quality of the results, flexibility is also an important aspect to consider when implementing the mask. It is important to be able to easily change or alter the mask, as there are different rules for different situations. Also,

the rules might change and need to be updated inside the radio. To provide the simplest solution to this, the mask is implemented as a table in a SQL database.

The database holds three sets of information: the minimum frequency, the maximum frequency, and the maximum power for that range. This information is then able to be used to create the mask. For the hand held UWB emissions mask, the SQL database appears as Table 4.2.

ID	Min Frequency	Max Frequency	Max EIRP
1	0	960000000	-41.3
2	960000000	1610000000	-75.3
3	1610000000	1900000000	-63.3
4	1900000000	3100000000	-61.3
5	3100000000	10600000000	-41.3
6	10600000000	999999000000	-61.3

Table 4.2: Hand Held UWB Emissions Mask Database

A wrapper called “mysql++” is used to allow the c++ code to interact with the database. It possesses all actions that a SQL database can accomplish, such as queries, table manipulations, and mathematical operations. Thus, in order to determine if a signal is legal, the program simply has to query the database to obtain the entries within the range of the signal and check to see if the maximum power is greater than the signal’s power.

The problem with running a query to check for each signal is that the queries take a relatively large amount of time. With many checks needed throughout the genetic algorithm, this adds a significant amount of run-time. Instead, only a single query has been placed at the initialization of the genetic algorithm, which brings in the entire mask table. The information on the table is stored in variables in the code and then used to evaluate the legality of a signal. This reduces the number of queries down to one, but does slightly increase the memory needed for the system.

The difference in run-time by only using one query is considerable. Table 4.3 provides the timing overhead from the mask for various mask implementations.

	Time (s)	
Pattern	Multiple Queries	Single Query
1 Generation	0.4178	0.055
2 Generations	0.1703	0.03
5 Generations	0.0703	0.0128
10 Generations	0.0375	0.0075
20 Generations	0.0203	0.0025
50 Generations	0.01	0.0025

Table 4.3: Mask Timing Overheads

The table shows a dramatic decrease in time between running multiple queries and a single query. The decrease is so much that even running the mask for every generation does not add enough overhead to prevent it as a possibility. This is because the genetic algorithm alone is already so computationally intensive that it takes a few seconds to run. Also, since it is a non-deterministic algorithm, it does not have the same runtime each execution. The resulting range in runtime is over a second, which is so much greater than the mask overhead that the additional time is unobtrusive. Because of this, time is no longer considered a factor in how the mask is implemented.

To allow for the simplest way to update or change the SQL database, the database information to be used is set as an input to the program. The database name, table name, username, and password are all stored into an XML document which the code parses during initialization. So to change the mask, all that needs to be done is to load the new SQL database and corresponding XML file.

4.1.7 Results

Since the amount of run-time is no longer an issue in how the mask is applied, the fitness of the final signal is the only factor in choosing how to employ the mask. To find the method that resulted with the highest quality, many different approaches were

executed and the results compared. The approaches differed in how often the mask was executed and how much of a penalty was provided if the signal was shown to be illegal.

The fitness used for this cognitive engine is based on relative quality as compared to another signal or sets of signals. Thus, to find the quality of the various approaches, results were obtained for each approach and then evaluated with respect to the results from the genetic algorithm without the mask. The evaluation calculates the fitness for all signals and the amount of improvement, or lack of, so that each application of the mask can be compared against all others.

Results from genetic algorithms are non-deterministic and thus a single comparison is not enough, as it may not be an accurate example of the results. In order to alleviate this, 100 results were taken for each approach and evaluated with 100 base results of the cognitive engine without a mask. The fitness values for each result are then summed together and provided in Table 4.4, along with the percentage of improvement. The lower the fitness value, the more fit the individuals are.

		Penalty				
Pattern		**10%**	**30%**	**60%**	**100%**	**Total**
1 Generation	Masked	1340	1416	1268	1432	5456
	Base	1512	1482	1364	1482	5840
	Improved %	11.38%	4.45%	7.04%	3.37%	6.58%
2 Generations	Masked	1320	1424	1350	1310	5404
	Base	1520	1426	1470	1316	5732
	Improved %	13.16%	0.14%	8.16%	0.46%	5.72%
5 Generations	Masked	1200	1402	1316	1320	5238
	Base	1440	1416	1552	1412	5820
	Improved %	16.67%	0.99%	15.21%	6.52%	10.00%
10 Generations	Masked	1352	1380	1336	1430	5498
	Base	1558	1382	1352	1520	5813
	Improved %	13.22%	0.14%	1.18%	5.92%	5.42%
20 Generations	Masked	1596	1414	1562	1380	5952
	Base	1438	1554	1482	1400	5874
	Improved %	-10.99%	9.01%	-5.40%	1.43%	-1.33%
50 Generations	Masked	1356	1338	1284	1446	5424
	Base	1448	1402	1372	1354	5576
	Improved %	6.35%	4.56%	6.41%	-6.79%	2.73%
Total	Masked	8164	8374	8116	8318	32972
	Base	8916	8662	8592	8484	34655
	Improved %	8.43%	3.32%	5.54%	1.96%	4.86%

Table 4.4: Fitness Results for Various Mask Executions

Table 4.4 presents the quality of the final selected signal from the cognitive radio based on two variable aspects: the mask pattern and the fixed penalty amount. Take for example the pattern of 10 generations. This represents the mask being exercised once every 10 generations. For a penalty of 10%, this means that if a signal where found to be illegal, its fitness value would be increased by 10%, thus making it less fit, no matter how illegal it is. The total fitness values are presented for the cognitive radio when the mask is implemented, and one, the base, without the mask. The improved percentage is then the improvement of the masked results over the base results. So, for every 10 generations with a fixed penalty of 10%, the selected signals that are found when using the mask are, on average, 13.22% more fit than when not using the mask.

Besides a fixed penalty, another penalty approach was tested where the penalty was proportionate to the percentage of the signal that was found to be illegal. For these cases, the amount that a signal is illegal, whether it be 30% or 60%, is calculated and the percentage of the signal that is illegal is used as the penalty for an illegal signal. Since the signal does not have a defined minimum to calculate the area, -80 dBm is used. The area above this minimum power, below the signal's power, and within the bandwidth is then calculated to find the signal's effective area. The affects of a variable penalty on the fitness of the final signal is presented in Table 4.5.

	Penalty	
Pattern		**Variable**
1 Generation	Masked	1439
	Base	1518
	Improved %	5.20%
2 Generations	Masked	1545
	Base	1534
	Improved %	-0.72%
5 Generations	Masked	1473
	Base	1487
	Improved %	0.94%
10 Generations	Masked	1419
	Base	1473
	Improved %	3.67%
20 Generations	Masked	1443
	Base	1385
	Improved %	-4.19%
50 Generations	Masked	1533
	Base	1475
	Improved %	-3.93%
Total	Masked	8852
	Base	8872
	Improved %	0.23%

Table 4.5: Fitness Results for Variable Penalty Mask Executions

The results overall do not show an overwhelmingly large improvement for any case, but that is to be expected. This is because the genetic algorithm finds strong results on its own and does not have much room for improvement. Nevertheless, the mask is shown to improve the results and any improvement is highly desirable.

The results are not exactly uniform, due to the simple randomness of the genetic algorithm. It is possible to see basic patterns provided though. For the fixed penalty, applying the mask every fifth generation provides the strongest results. Applying it every first or second generation is too often and does not allow the information that may be valuable, but illegal for the selected signal, to propagate into a legal signal. Less frequent than every fifth generation loses any affect as it is not executed enough. This is noticeable in Table 4.4 as the improvements for these patterns are closer to null.

The smallest fixed penalty tested, 10%, provided the best results of all types of penalties. The reason for this is that the smaller penalties act simply as a tiebreaker between two equally fit signals. The fitness evaluation used results in many signals with the same fitness value. The smaller penalties allow the legal signal to propagate to the next generation over the illegal signals. Greater penalties can then allow less fit, but legal, signals to propagate over more fit, but illegal signals. This then can hinder the fitness improvements from one generation to the next.

The variable penalty results do not show any promising or uniform results. The idea for using a variable penalty was to punish more illegal signals a higher amount than less illegal signals. We were hoping that less illegal signals were more likely to possess information that would become legal through the genetic algorithmic process and should be more likely to be propagated to future generations. The reason for this likely has to do with the same reasoning behind why greater fixed penalties are less helpful than lower penalties. When signals are illegal, they are usually illegal by larger amounts than the previously found best penalty of 10%. The variable penalty also adds another area of randomness in the code. All together, the variable penalty is not an effective method to penalize the fitness of illegal signals inside the cognitive engine.

The smaller penalty of 10% was the best of all the penalties, and applying the mask every fifth generation was the best pattern. Therefore, not surprisingly, the best selection of penalty and pattern turned out to be these two. This will provide a result that is likely to be somewhat more fit than without the application of the mask.

Appling the mask provides a very practical solution to ensure that the signal produced by the cognitive engine is legal and valid. It is simple to integrate and allows for high flexibility with its ability to be altered in a quick and easy way. It only adds a minimal amount of computational power, which when compared with the entire system is very inconsequential. It also provides a result that is likely to be more fit than without the mask. Overall, the mask and how it is applied, offers a reasonable way to ensure the legality of the signal.

4.2 API CHECKER

Once the cognitive engine has selected a legal signal, the signal is passed on to the radio API before it is sent to the hardware. The API interprets the cognitive engine commands to the radio and interprets the radio's responses back to the cognitive engine. This is another spot where an error may occur in the code and thus another BIST is used to ensure the output of the API is correct. For this research, the focus is on ensuring that the interpretation of the cognitive engine commands to the radio is correct. The interpretation is done by modulating the signal, which is the primary concern because incorrectly doing so can cause the radio to output an illegal signal.

4.2.1 RADIO API

As stated before, the API modulates and demodulates signals so that the cognitive engine and radio can interact with each other. The API takes in an XML document from the cognitive engine with the radio specifications and uses that to modulate the information into the correct signal. The basic structure of this is provided in Figure 4.5.

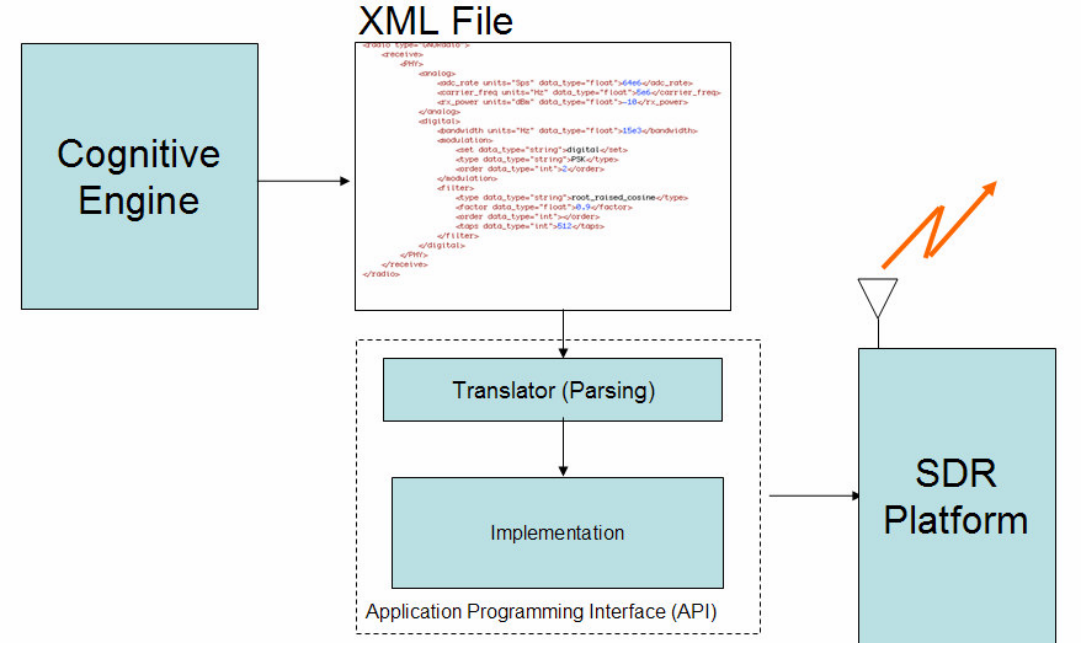

Figure 4.5: Basic API Structure [33]

The API is implemented through various operational blocks. This is represented by the flow graph in Figure 4.6. The input information is first sent to the packet construction block. That information is then modulated and amplified before it is sent to the radio.

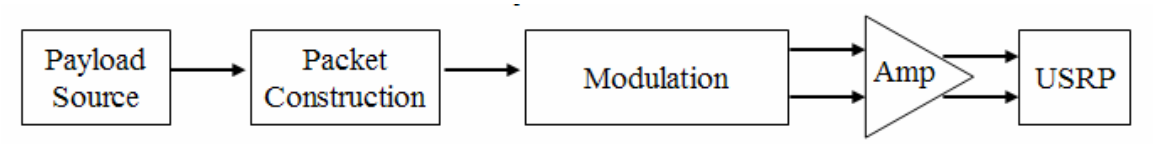

Figure 4.6: API Flow Graph [34]

Any error in any block can cause a signal to be sent to the radio that is illegal, and could interfere with other, possibly very important signals, such as military and public safety communications, as well as signals controlling medical devices. Therefore it is imperative to ensure that the modulation of a signal is done correctly and legally.

4.2.2 CHECKER

With the policy mask embedded into the cognitive radio ensuring that the selected signal is legal, it is not necessary to recheck that the modulated signal is also legal against the policy mask. It is only necessary to make sure that the outputted signal is the same signal that was selected by the cognitive engine.

The checker is embedded in the API. If an incorrect modulation is found, the results from the modulation are blocked from the USRP. In order to be able to halt the incorrect modulation being sent, a buffer is placed between the API and the hardware radio so that the connection to the radio can be cut when needed. The flow graph of how this is connected is presented in Figure 4.7. The graph depicts how the API, the dotted block on the left, is separated from the USRP, which is the radio hardware. The buffer can be cut at anytime, preventing the radio from obtaining that modulation that may produce an illegal signal.

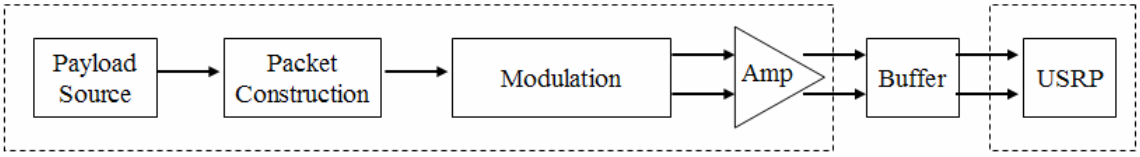

Figure 4.7: API Flow Graph with Buffer [34]

The checker tests for an incorrect signal and the results are used to select if the buffer continues to propagate the signal or not. It is applied on the same information that is going to the USRP, so it is placed at the output of the amplifier. If the check finds an incorrect signal, it sends information to the buffer to halt propagation. The flow graph for this is presented in Figure 4.8.

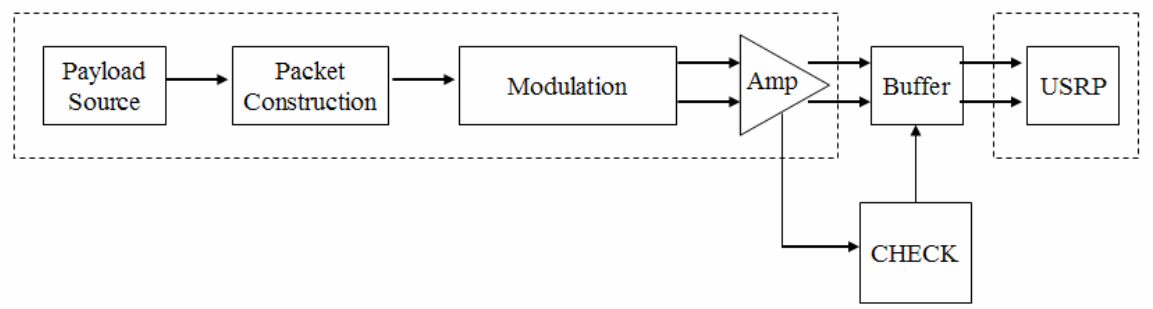

Figure 4.8: API Flow Graph with Checker [34]

Again, the checker does not need to verify for legality against the policy mask, but rather just confirm if the modulated signal is the correct signal that the API was supposed to produce. To do this, the signal from the amplifier is sent to an FFT. This will provide information on the center frequency and power of the signal. The power and frequency can then be compared with the radio specifications that are inputted from the XML document. If they are the same, then the buffer is allowed to propagate the signal, else the signal will be blocked.

Using only the power and the center frequency to check if the modulated signal is correct is not a full check. There are many other specifications that are not examined, such as the bandwidth. These specifications are not checked primarily because they are much more difficult and computationally intensive to check. Also, the power and frequency are the most important factors in determining whether a signal is legal or not.

4.2.3 Summary

Combined with the policy mask that guarantees the selected signal is legal, ensuring that the modulated signal is the same will prove that all signals being sent to the radio hardware are legal and valid. Since all hardware for the radio must be approved by the FCC, we can assume that the hardware will correctly produce the signal. All together, this ensures that all signals being produced by the cognitive radio will be legal with respect to the FCC rules and regulations.

Applying the checker to the API is a simple and practical way to ensure that the signal produced by the API is the same as the signal selected by the cognitive engine. It is easy to apply, not computationally intensive, and can quickly halt the execution of the signal if an error is found.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This chapter presents conclusions drawn from the work and recommendations for future work.

5.1 CONCLUSTIONS

The work presented in this thesis provides a complete and practical approach ensure the legality of a signal in a cognitive radio. It was important to achieve a 100% success rate as that is what the FCC requires such and any error could produce very harmful results.

A software testing program that is aimed at finding structural errors has been produced. The approach is based on the tabu search algorithm with slight alterations to improve performances on software testing. This thesis has shown that this program provides strong results with few test vectors. It also has been demonstrated that it does not require much effort by the user in applying to a particular code segment, as the algorithm does not need many specifics on which program is being tested.

A built-in self-test (BIST) for the software has also shown to be useful. The work done in this thesis was able to show the most effective way to implement the BIST, how it is able to assure 100% legality of the signal, and that it does not add additional run-time to the cognitive engine.

Overall, the FCC has been searching for ways to ensure that cognitive radios will not effect existing license holders [35]. The complete approach presented in this thesis does a good job of completing this goal. The BIST will ensure any signal produced is legal and valid, and the software testing program will help to find many errors offline, so that the disruption that occurs when the BIST does find an erroneous signal is minimized.

5.2 FUTURE WORK

The work presented in this thesis opens up the possibility for future work.

The software program shows the viability of the tabu search algorithm in software testing. With only a minimal amount of research done on this area, many possible areas to focus on in future work are available. One main area is the use of the searching algorithm with different coverage metrics, such as path coverage and loop coverage. Also, other partitioning schemes and other rules to set items as tabu are definitely possible.

There are several paths for future work with the mask. One is a more flexible, and possibly dynamic, pattern, such as applying the mask at different generational spacing. Another is to improve the model, so that it might take into affect the side lobes. These two research possibilities could improve the effectiveness of the mask in the genetic engine.

The work done on the API checker is only to ensure that an illegal signal is not produced. Therefore, a lot of future research can be done to find the best way to correct and resend the signal.

REFERENCES

[1] T. W. Rondeau, B. Le, D. Maldonado, D. Scaperoth, C. W. Bostian. "Cognitive Radio Formulation and Implementation", IEEE Proc. CROWNCOM, Mykonos, Greece, 2006.

[2] A. Hertz, E. Taillard, D. de Werra. "A Tutorial on Tabu Search", www.cs.colostate.edu/~whitley/CS640/hertz92tutorial.pdf

[3] E. Diaz, J. Tuya, R. Blanco. "Automated Software Testing Using a Metaheursitic Technique Based on Tabu Search", Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003. 6-10 Oct. 2003 Page(s):310 – 313.

[4] J. Mitola III. "Cognitive Radio for Flexible Mobile Multimedia Communications", 1999 IEEE International Workshop on Mobile Multimedia Communications, (MoMuC '99). 15-17 Nov. 1999 Page(s):3 – 10.

[5] J. Mitola III, G. Q. Maguire Jr. "Cognitive radio: making software radios more personal", IEEE Personal Communications. Aug. 1999 Vol. 6, Issue 4, Page(s):13 - 18

[6] J. Mitola III, "Cognitive INFOSEC," Proceedings of the IEEE International Microwave Symposium 2003. 13-16 June 2003 Page(s): 1051 – 1054, vol. 2

[7] J. Mitola III. "Signal Processing Technology Challenges of Cognitive Radio", Dec. 2005.

[8] J. Mitola III, Cognitive Radio, Doctoral Dissertation, Stockholm: KTH, The Royal Institute of Technology

[9] T. Back. Evolutionary Algorithms in Theory and Practice, Oxford University Press. 1996.

[10] D. S. Alberts. "The economics of software quality assurance", Proceedings: 1976 National Computer Conference. Volume 45, Page(s):433-442.

[11] W. Lin, Z. Wei, M. Xueyan. "Research on Searching Algorithm for Test Data Generation", IEEE Proceedings of Intelligent Transportation Systems, 2003. Volume 1, 2003 Page(s):384 – 388.

[12] J.E. Heiser. "An Overview of Software Testing", 1997 IEEE Autotestcon Proceedings, 22-25 Sept. 1997 Page(s):204 – 211.

[13] C. Jones. "Software Quality in 1997: What Works and What Doesn't", Software Productivity Research, Inc., One New England Executive Park, Burlington, MA, 1997.

[14] S. Beyleda, V Gruhn. "BINTEST - Search-Based Test Case Generation", Proceedings from the 27th Annual International Computer Software and Applications Conference, 2003, COMPSAC 2003. 3-6 Nov. 2003 Page(s):28 – 33.

[15] J. T. Webb. "Static Analysis [Software Testing]", IEEE Colloquium on Software Testing for Critical Systems. 19 Jun 1990 Page(s):4/1 - 4/3.

[16] P. D. Coward. "Symbolic Execution and Testing", IEEE Colloquium on Software Testing for Critical Systems. 19 Jun 1990 Page(s):2/1 - 2/3.

[17] Disjunctive Image Computation for Embedded Software Veri_cation

[18] R. Ferguson, B. Korel. "Software Test Data Generation Using the Chaining Approach", Proceedings from the International Test Conference, 1995. 21-25 Oct. 1995 Page(s):703 – 709.

[19] B. Korel, A. M. Al-Yami. "Assertion-oriented automated test data generation", Proceedings of the 18th International Conference on Software Engineering, 1996. 25-30 March 1996 Page(s):71 – 80.

[20] B. Korel. "Automated Software Test Data Generation", IEEE Transactions on Software Engineering. Volume 16, Issue 8, Aug. 1990 Page(s):870 – 879.

[21] A. Denise, M. C. Gaudel, S. D. Gouraud. "A Generic Method for Statistical Testing", 15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2006. 2-5 Nov. 2004 Page(s):25 – 34.

[22] J. A. Whittaker, M. G. Thomason. "A Markov Chain Model for Statistical Software Testing" IEEE Transactions on Software Engineering. Volume 20, Issue 10, Oct. 1994 Page(s):812 – 824.

[23] F. Glover, M. Laguna. *Tabu Search*. Kluwer Academic Publishers. 1997

[24] N. Gupta, A. P. Marthur, M.L Soffa. "Generating Test Data for Branch Coverage", 15th IEEE International Conference on Automated Software Engineering (ASE'00). Sept. 2000.

[25] J. Lin, P. Yeh. "Automatic test data generation for path testing using GAs", Information Sciences 131. 2001.

[26] C. Michael, G. McGraw, M. Schatz, C. Walton. "Genetic Algorithms for Dynamic Test Data Generation", 12th IEEE International Conference on Automated Software Engineering (ASE'97). Nov. 1997.

[27] N. Tracey, J. Clark, K. Mander. "Automated program flaw finding using simulated annealing", International Symposium on software testing and analysis ACM/SIGSOFT. 1998.

[28] R. Zhao, M. R. Lyu, Y. Min. "A new software testing approach based on domain analysis of specifications and programs", 14th Interational Symposium on Software Reliability Engineering, 2003. ISSRE 2003. 17-20 Nov. 2003 Page(s):60 - 70

[29] T. Vagoun. "Input domain partitioning in software testing", Proceedings of the 29th Hawaii Interanational Conference on System Sciences, 1996. 3-6 Jan. 1996 Page(s):261 – 268 vol. 2

[30] G. Hetherington, T. Fryars, N. Tamarapalli, M Kassab, A. Hassan, J Rajski. "Logic BIST for large industrial designs; real issues and case studies", Proceedings from them International Test Conference, 1999. 28-30 Sept. 1999 Page(s):358 - 367

[31] FCC. "FCC UWB Operation for Communications and Measurement Systems"

[32] J. O. Smith III. "Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications", Center for Computer Research in Music and Acoustics (CCRMA). http://ccrma.stanford.edu/~jos/mdft/mdft.html

[33] D. Scaparoth. "gnuradio_api_tutorial.ppt"

[34] D. Scaparoth. "gnuradio_policymask.ppt"

[35] J. Walco, "Cognitive Radio", IEEE Review, May 2005, Vol. 51, Issue 5, Page(s):34 - 37

APPENDIX A

TRIANGLE PROGRAM

```
void triangle(values *val)
{
	int x;
	char type[4];
	char a,b,c;

	for (x=0;x<4;x++)
	{
		if (x==0)
		{
			a = val->length[0];
			b = val->length[1];
			c = val->length[2];
		}
		else if (x==1)
		{
			a = val->length[0];
			b = val->length[1];
			c = val->length[3];
		}
		else if (x==2)
		{
			a = val->length[0];
			b = val->length[2];
			c = val->length[3];
		}
		else if (x==3)
		{
			a = val->length[1];
			b = val->length[2];
			c = val->length[3];
		}

		if (a <= 0 || b <= 0 || c <= 0)
		{
			type[x] = 0;
		}
		else
		{
			if (2*a < a+b+c && 2*b < a+b+c && 2*c < a+b+c)
			{
				if (a==b)
				{
					if (b==c)
					{
						type[x] = 2;
					}
					else
					{
```

```
                                    type[x] = 3;
                              }
                        }
                        else
                        {
                              if (a==c)
                              {
                                    type[x] = 3;
                              }
                              else
                              {
                                    if (b==c)
                                    {
                                          type[x] = 3;
                                    }
                                    else
                                    {
                                          type[x] = 4;
                                    }
                              }
                        }
                  }
                  else
                  {
                        type[x] = 1;
                  }
            }
      }
}
```

APPENDIX B

COGNITIVE ENGINE MODULE TESTING OVERVIEW

Population Functions			
Module	**Tested**	**Untested**	**Notes**
SelectT	X		
Crossover		X	Few and simple conditional statements to apply coverage metric
Validate	X		
Evaluate	X		
Replace		X	No conditonal statements to apply coverage metric
Sort	X		
Reproduce		X	No conditonal statements to apply coverage metric
Find_best_member	X		
WriteHistogram		X	Doesn't affect results

Individual Functions			
Module	**Tested**	**Untested**	**Notes**
Init		X	No conditonal statements to apply coverage metric
Define_chromosome		X	No conditonal statements to apply coverage metric
Decrypt_chromosome	X		Not complete test, not all branches applied on coverage metric
Decrypt_gene	X		
Copy		X	No conditonal statements to apply coverage metric
Evaluate	X		Not complete test, not all branches applied on coverage metric
Mutate	X		
Display_individual		X	No conditonal statements to apply coverage metric

VITA

Patrick Cowhig was born in Sherman, TX, where he lived until graduating from high school. He started his studies at Baylor University in 2000 and received a Bachelor's Degree in Electrical and Computer Engineering in 2004. Afterwards, he joined Virginia Polytechnic Institute and State University to pursue a Master's Degree in computer engineering. While at Virginia Tech, he joined Dr. Michael Hsiao's research group, PROACTIVE, and began his research in the area of software testing. He recently obtained a job with Freescale Semiconductor in Austin, TX, where he will be working as a functional verification engineer. Patrick's hobbies include watching and playing sports, lifting, and running.