U.S.N.A. — Trident Scholar project report; no. 337 (2005)

PREDICTION AND IMPROVEMENT OF SAFETY IN SOFTWARE SYSTEMS

by

Midshipman Sean A. Jones, Class of 2005
United States Naval Academy
Annapolis, Maryland

Certification of Adviser Approval

Associate Professor Donald M. Needham
Computer Science Department

(signature)

(date)

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Deputy Director of Research & Scholarship

(signature)

(date)

USNA-1531-2

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 9 May 2005	3. REPORT TYPE AND DATE COVERED

4. TITLE AND SUBTITLE Prediction and improvement of safety in software systems	5. FUNDING NUMBERS
6. AUTHOR(S) Jones, Sean A. (Sean Alexander), 1982-	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Naval Academy Annapolis, MD 21402	10. SPONSORING/MONITORING AGENCY REPORT NUMBER Trident Scholar project report no. 337 (2005)
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT This document has been approved for public release; its distribution is UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT: The modern military's abilit to fight depends heavily on complex software systems, making the safety of such of software of paramount importance. The transformation of the military's analog combat systems to computer-based systems has been plagued by software problems ranging from benign flight simulator issues to 'smart' ships finding themselves dead in the water. The military's interest in increasing automation in order to reduce manpower requirements makes even trivial software safety issues a serious concern. The software engineering community is not well equipped to reduce the safety risks incurred through use of such systems, and stands to benefit from metrics, analysis tools, and techniques that address software system safety from a design perspective. The purpose of this research project was to propose and develop tools that software engineers can use to address the issue of software safety. The project focused on safety prediction and improvement through the use of software fault trees coupled with "key nodes," or fault treebased safety metric, and an algorithm for estimating the improvement costs necessary to achieve a targeted level of software safety. The safety prediction metric uses the key node property of fault trees while the improvement algorithm is based on the mathematical relationship between nodes in a fault tree, and yields an estimate of the man-hours necessary to improve a system to a targeted safety value based on cost functions supplied by a component's developer. These metrics and algorithms allow designers to measure and improve the safety of software systems early in the design process, allowing for a reduction in costs and an improvement in resource allocation.

14. SUBJECT TERMS: safety, metrics, software engineering, software development	15. NUMBER OF PAGES 72
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Abstract

The modern military's abilit to fight depends heavily on complex software systems, making the safety of such of software of paramount importance. The transformation of the military's analog combat systems to computer-based systems has been plagued by software problems ranging from benign flight simulator issues to 'smart' ships finding themselves dead in the water. The military's interest in increasing automation in order to reduce manpower requirements makes even trivial software safety issues a serious concern. The software engineering community is not well equipped to reduce the safety risks incurred through use of such systems, and stands to benefit from metrics, analysis tools, and techniques that address software system safety from a design perspective.

The purpose of this research project was to propose and develop tools that software engineers can use to address the issue of software safety. The project focused on safety prediction and improvement through the use of software fault trees coupled with "key nodes," or fault tree-based safety metric, and an algorithm for estimating the improvement costs necessary to achieve a targeted level of software safety. The safety prediction metric uses the key node property of fault trees while the improvement algorithm is based on the mathematical relationship between nodes in a fault tree, and yields an estimate of the man-hours necessary to improve a system to a targeted safety value based on cost functions supplied by a component's developer. These metrics and algorithms allow designers to measure and improve the safety of software systems early in the design process, allowing for a reduction in costs and an improvement in resource allocation.

Keywords: safety, metrics, software engineering, software development

Acknowledgements

I would like to thank everyone who has supported me in the past year in this endeavor. In particular I would like to thank:

- Associate Professor Needham for always being there to help in any random way, including early morning drafts.
- CAPT and Mrs. Baker provinng me a home away from home on the yard.
- Dr. Callahan for pointing me in an interesting direction.
- My Company Officer, Major Powers, USMC, for being behind me through the entire adventure.
- Julie for always being patient and understanding despite my repeated statements of "I'm busy with Trident."
- Professor Madison for helping edit the mess that was this document.
- Assistant Professor Brown and Assistant Professor Crabbe for suggesting the presentation format.
- Mr. Overbey for assisting in presentation technique and delivery.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the number of software controlled systems expands, the issue of software safety becomes an increasingly critical aspect of system development. This is especially true in the fields of aerospace, astronautics, and nuclear engineering, where many analog safety critical systems have been transformed into digital systems that provide greater flexibility of control setups, improved runtime efficiencies, and reduced operating costs. However, the downside of transitioning to digital systems includes the difficulty of verifying the reliability of the software system. Despite these issues, software control systems have been introduced to many hazardous systems including nuclear power plant controls, aviation guidance systems, and weapons systems. As an example, in 1955 only 10 percent of weapons systems required software, but by 1981 over 80 percent of systems required software support [12]. Even in increasingly hazardous systems, the use of software has been justified through the large number of benefits and the few practical alternatives. The benefits include flexible programming, increased capability, greater reusability, and decreased deployment costs, while the only alternatives are inflexible hardware-only systems, which must be redesigned for each new system or major system change and tend to be expensive. Another concern with the increased role of software control systems is that the environments in which they run preclude manual intervention and require automatic failover to backup systems [12]. While reliability in general is an important concern in the software control systems, safety is highlighted as one part that is required before software systems can completely replace hardwired systems.

The remainder of this report is organized as follows: Chapter 2 discusses background information including the principals of software safety, software fault injection, and software fault trees, as well as treatment of the basis for software safety prediction. Chapter 3 examines theoretical foundations, with a particular emphasis on the “key nodes” of software fault trees, and proposes a key nodes-based metric for predicting system safety based on analysis of the structure of the system’s representation as a fault tree. Additionally, Chapter 3 also presents an algorithm for improving a software system’s probability of failure. Chapter 4 presents experiments, focused on improving system safety conducted using key nodes of software fault trees. Chapter 5 analyzes the performance of the key node-based metric as well as the improvement algorithm. Chapter 6 reviews the conclusions this research, and presents areas of future work.

Chapter 2

Background

Software engineering is a young field, tracing its development to a 1968 NATO conference tasked with solving what was described as the "Software Crisis" [17]. The growing use of computers combined with the increasing complexity of software programs caused the quality of future software development to become a major concern. Turning to the general engineering community for an example, early software engineers adopted a design model similar to what a civil engineer might use in constructing a bridge over a river. Their process consisted of six phases:

1. requirements
2. specification
3. design
4. implementation
5. integration
6. maintenance

Neither then, nor now, did software engineers consider testing to be a separate phase, since testing occurs to varying degrees throughout all of the phases.

Software engineers use metrics to evaluate internal software qualities, such as size or structural complexity, as well as to measure external traits like reliability. The early metrics, developed in the mid 1960s and referred to as Lines of Code metrics, were based on the concept of program length, and included variations such as thousands of lines of source code, object code, and assembly code [5]. In the 1970s, several major advances in the area of software metrics were made, including McCabe's Cyclomatic Complexity Metric, focusing on a program's control flow, and Halstead's Software Volume Metric, focusing on the number of operands and operators [8, 15]. In the 1980s, software engineers began to focus on two diverse areas: dynamic methods of verification such as software fault injection in which incorrect source code is intentionally inserted into a program, and formal methods such as mathematically proving program correctness. Metrics are more closely aligned with formal methods because they calculate a value based on the intrinsic characteristics of a program rather than the trial and error methods typical of dynamic testing. The emergence of the DoD-initiated Software Engineering Institute's Capability Maturity Model (CMM) in 1986 provided an expanded interest in metrics as firms began focusing on metrics as a requirement of earning higher CMM levels [5]. CMM is a method of assessing the maturity of a business's software development process. The DoD requires that all contractors who wish to bid on software projects must be of CMM level 3 or higher [1].

2.1 Software Safety

Software safety is a sub-area of software testing concerned with preventing software from entering what is regarded to be a hazardous state. A hazardous state, in this context, is defined as a system in a condition that is a risk to life, lacks the ability to complete its mission, or that may incur severe financial loss [13]. Any faults that are not hazardous, are considered to be of secondary concern – a primary difference between software safety and software reliability. By comparison, in terms of software reliability, all faults, regardless of the scale of any consequences, are viewed the same: no faults are acceptable and therefore any fault is bad [6]. Safety takes a subtle yet dramatically different approach: any fault that does not have catastrophic consequences can be ignored as not important [13].

A major benefit provided by the software safety perspective becomes evident when the input and output spaces of software safety versus software reliability are evaluated. The input space of a program consists of all possible input values a program will accept. Similarly, the output space contains all possible output values produced by the program. For example, the input space of a function that accepts two 32 bit integers and returns a 32 bit integer contains 2^{64} elements in its input space while the output space consists of 2^{32} unique values. In order to completely test the reliability of a function, every possible input, i.e. the complete input space, must be empirically evaluated by the system. The output space is only critical for reverse testing methods, such as fault-tree analysis, that are used in safety assessment (Fault trees are a systems-engineering construct that creates a tree based on the failure relationships between the components of a system [21]). The large sizes of the input and output spaces are of major concern for software testing. Huang demonstrated that testing the reliability of a program with a combined input-output space of a size comparable to the two 32 bit integers example is an "intractable" problem – one which requires inordinate, or infinite, resources due to the size of the problem space [9, 6]. In the example of a function accepting two 32 bit integers, assuming the conduction and evaluation of one input case per millisecond, inordinate resources are required since it would take approximately 50 billion years to test every element in the 2^{64} input space.

Reliability analysis provides two methods to work around the issue of tractability: sampling and folding. Sampling, as its name suggests, involves random sampling of the input space and testing the sampled values. An example of sampling is branch coverage, a technique that seeks to ensure that every branch in a software program is executed at least once during testing [24]. Branch coverage is considered sampling because only enough inputs are examined to verify that all reachable code is executed. Folding, the other approach to making software testing more tractable, consists of reducing the input space by combining multiple input values into a single class of inputs that can be evaluated as a single unit. An example of folding is the use of equivalence classes: reducing the number of inputs by grouping all inputs that produce similar outputs into a single class, and then ensuring that a subset of each equivalence class is tested [24]. While both sampling and folding seem like solutions to the issue of software testing tractability, each approach alters the actual problem being solved, and the farther the problem is abstracted from the original the more significant are the alterations [24]. Both methods reduce problem size, but as a result of the loss of resolution created by the abstraction of the input space, they do not satisfactorily solve the tractability problem.

2.2 Analyzing Software Safety

Safety analysis avoids the intractability problem by narrowly defining the input and output spaces, focusing on only those output states that place the system in a hazardous condition. The output space is culled to include only those values which describe the predefined hazardous states. The input space is then defined as the subset of inputs necessary to produce the outputs still present in the output space. Since it may take many output values to completely describe each hazardous state, the safety perspective still leaves in the order of hundreds or thousands of output values to be tested. In turn, each output value can be mapped directly to a small number of input values. Continuing with the two 32 bit integer input example above, we assume the output space consists of 2^{32} values, and that there is some number, H, of hazardous states, where H is less than 50 and each hazardous state can be triggered by 1000 different output values. Each output value is mapped directly to four unique ordered pairs of the two input variables. The output space then shrinks to at most 50,000 output values and a maximum of 200,000 input pairs. Unlike the 2^{64} inputs, 200,000 inputs are tractable. Using the same assumption of evaluating one input case per millisecond, complete testing would take 200 seconds. Run times of up to several hours or even days would still be considered tractable. Figure 2.1 outlines the differences and overlap between faults, hazards, area of interest for reliability, and the area of interest for safety. The figure also displays how reliability and safety are related through the input and output spaces of a program.

Since the safety output and input spaces are subsets of the reliability input and output

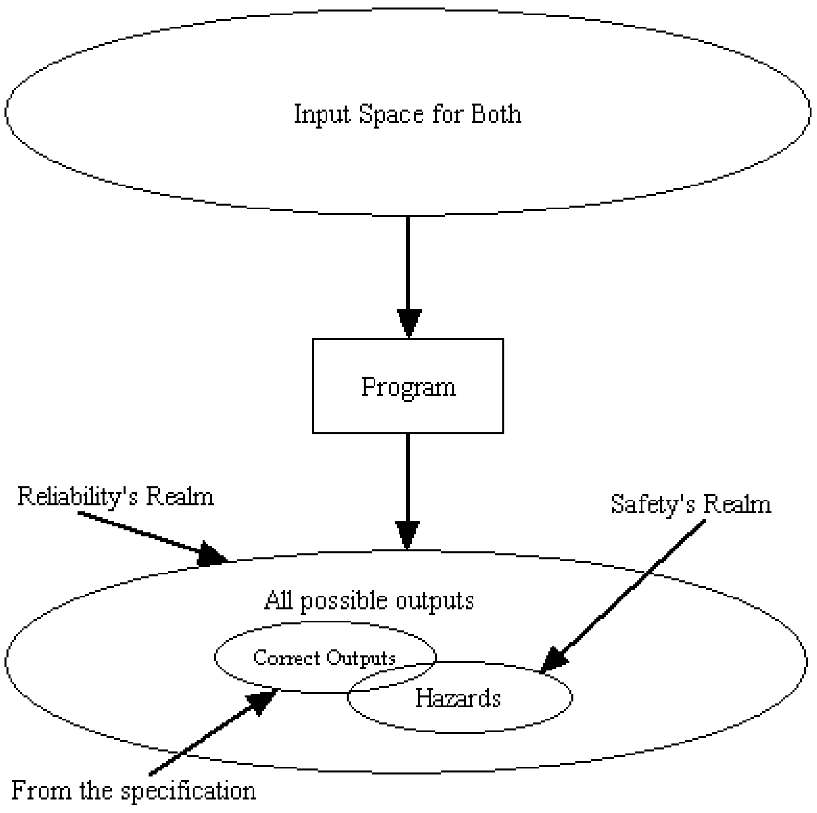

Figure 2.1: Output Spaces of Reliability and Safety

spaces, the concepts of safety and reliability are generally orthogonal. A 100% reliable program always produces a correct output as determined from the program's specifications. However, unless the program's specifications include all the necessary safety specifications, there is no guarantee that a correct output is a safe output. An example can be found in a firearm that is supposed to fire when the firing pin strikes the primer, regardless of whether the trigger was pulled. If a firearm fails to fire when the trigger is pulled, the firearm is safe, but unreliable. Conversely, if the same firearm is dropped, and inertia causes the firing pin to strike the primer, the weapon may be reliable, but it is unsafe. A software example related to reliability and safety can be found early in the U.S. Air Force B-1 Lancer program. During scheduled maintenance and testing, the bomb bay doors where held open by a mechanical interlock for maintenance at the same time that flight avionics tests were being conducted in the cockpit. As part of the avionics test, the command sequence to close the bomb bay doors was entered. Since the doors were held open by mechanical interlock, the doors didn't close; however two hours later, when maintenance was complete and the mechanical interlocks were removed, the doors closed unexpectedly. The software control system had kept trying to execute the close door command until it was successful, despite the two hour delay from the initial close-doors command. The problem was fixed by adding a timeout to disregard any commands not executed within a specified time frame [7]. Graphically, the relationship between reliability and safety can be seen in Figure 2.2. The Venn diagram displays how the realms of reliability and safety interact with predictors and metrics. The precise amount of the overlap is defined by the amount of requirements overlap between the two attributes.

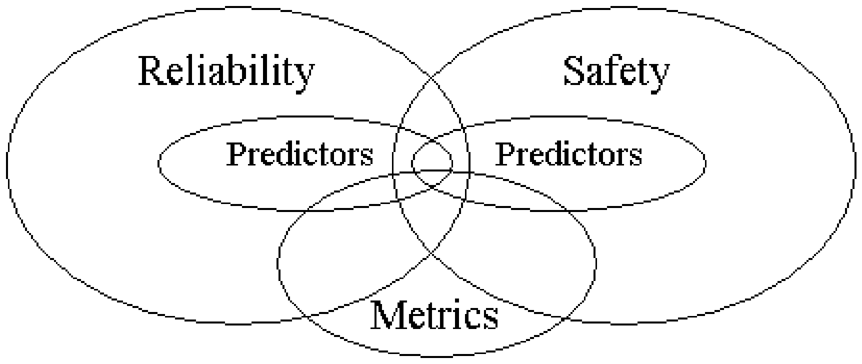

Figure 2.2: Relationship of Safety and Reliability

2.3 Fault Injection

Fault injection is a non-traditional dynamic testing technique based on verifying how a program will react when certain faulty conditions are deliberately introduced, and traces it roots to Mills's work on fault seeding [22]. Mills proposed seeding a program containing N faults (where the value of N is unknown) with M known faults [16]. The seeded program was then tested to find faults. The maximum estimate of N is given by:

$$N = \frac{M(r-k)}{k} \tag{2.1}$$

Here N is the total number of faults in the program, M is the number of seeded faults, r is the number of faults found in testing, and k is the number faults found in testing that were seeded. This estimate of N also assumes that none of the faults found were fixed and ignores compound faults. The difference between Mills's proposal and current fault injection techniques is that injected faults are not presently being used as a measurement aid, but rather as a form of dynamic testing to verify a program's response to a particular type of input.

As with many of the inter-disciplinary techniques found in software engineering, the concept of fault injection is also common practice in other domains such as electrical engineering. For example, a common way to test a circuit, whether it is digital or analog, is to force various chip and system inputs into erroneous states. This method of fault injection is most similar to the practice of software fault injection used in this work. While seen as a staple of testing in electrical engineering, fault injection has met with resistance in the area of software engineering primarily due to the question of the plausibility of injected anomalies [22]. In electrical engineering, fault injection is accepted as a valid testing technique because the physical processes are well understood, while the processes behind software fault injection are not as widely accepted [22].

Voas's view of modern fault injection focuses on four fault injection techniques: message-based, memory/storage-based, debugger-based, and process-based [22]. Message-based fault-injection corrupts the messages sent between processes. Memory/storage-based fault injection allows the testing of components without a message passing interface through the alteration of data stored in memory or other form of data store. Debugger-based techniques inject faults into the memory states of running processes. Process-based injection considers the idea of a higher priority process corrupting the process being tested in memory. Once the method of fault injection is chosen, the focus turns to precisely what is being injected and where it is being injected. In this case, the answer to the question of what and where can be solved by applying of fault trees.

2.4 Fault Trees

Fault trees are interdisciplinary, finding application for both hardware and software systems in many engineering disciplines, including aerospace, astronautics, and nuclear engineering. Fault trees originated at Bell Labs in the 1960s in support of the U.S. Air Force's Minuteman Missile Program, and are composites of fundamental entities known as events and gates, which form the nodes and connections of the tree [21]. The root of a fault tree is the predefined hazard event which the designer of the tree seeks either to prevent or to minimize the probability of occurring. A hazard event is any event in a system that has the potential of causing a variety of undesirable results such as loss of life, equipment loss, unacceptable loss of functionality, or undesirable operating conditions. The leaves of the tree represent the fundamental events (inputs) to the system. The root and leaves are connected by a series of intermediate events through Boolean operators as shown in Figure 2.3. In a fault tree, because the focus of the tree is on the probability of failure rather than the probability of success, Boolean true is defined somewhat counter-intuitively as the failure of a node and Boolean false is the success of a node. Because intermediate events are themselves Boolean expressions, the entire tree can be expressed as a composite Boolean expression that can be simplified using straight-forward algebraic manipulations. When the probability of the leaf elements are inserted into the Boolean expression describing the system, a probability of occurrence can be determined for the specified hazard at the root of the tree. For all but the simplest systems, computing the probability of occurrence is a non-trivial task because of the computation of the conditional probabilities at each node.

In Figure 2.3, the leaf nodes are labeled d, e, f, and g and the internal nodes are a, b, and c with a also being the root. In order for node b to enter a failure state, both nodes d and e must fail, however for node c, either nodes f or g can fail to cause the system to enter a failure state. Node a is similar to node c in that either nodes b or c can fail to create a

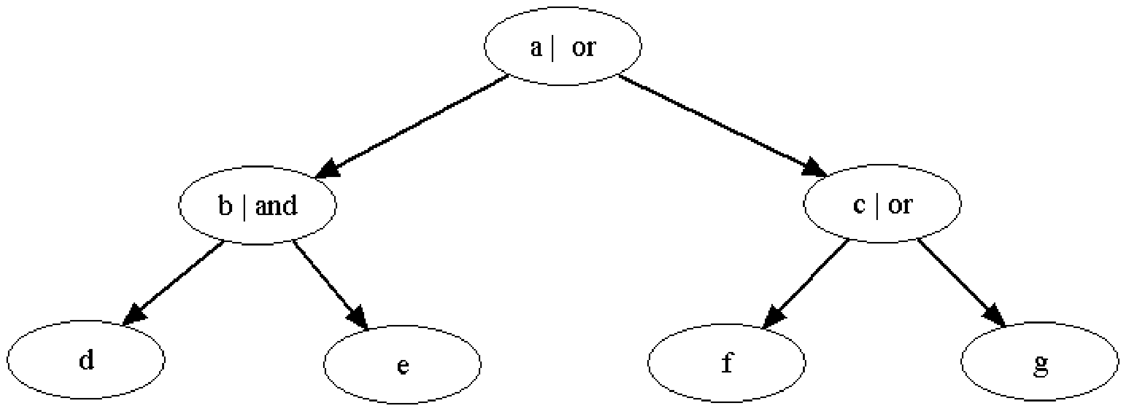

Figure 2.3: Example Fault Tree

failure condition. When node a is in a failure condition, the hazard described by the fault tree occurs.

For purposes of explanation, consider a hazard representing loss of control of an automobile realized as the example fault tree of Figure 2.3. Allow node d of Figure 2.3 to represent Loss of Breaks, and node e to represent Loss of Ability to Use Engine Braking due to a failed transmission. Node f could represent loss of Directional Control due to a steering system failure and node g could represent Loss of Directional Control due to icy road conditions. In this case, node b represents the loss of ability to slow the vehicle via the failure events of both node d and node e, while node c would represent the loss of directional control due to either the loss of steering, icy road conditions, or both. Node a, the root hazard of Figure 2.3, would then represent the loss of vehicle control due to the hazards represented by node b, node c, or both.

In computing the probability of a fault tree entering a hazard state, we must consider equations for AND or OR node systems. Equation 2.2 is for an AND node system such as in Figure 2.4 and the equation 2.3 is for an OR node system such as in Figure 2.5. In equation 2.2 the failure probability of the two children, a and b, are multiplied together because the probability of an AND system requires both two fail. Since an OR system has the opposite probability relation of an AND system the minus terms are required to be able to use the same type of input probability terms [21]. While not discussed in this report, there are also exclusive or (XOR) relationships. The extension of the this research to XOR relationships is for future work.

$$P(a,b) = ab \tag{2.2}$$

$$P(c,d) = 1-(1-c)(1-d) \tag{2.3}$$

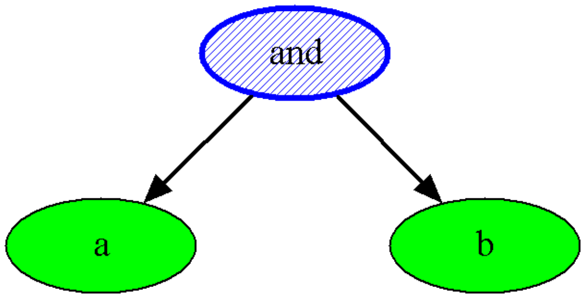

Figure 2.4: AND probability relationship

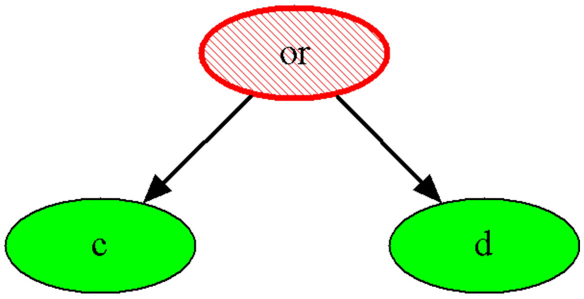

Figure 2.5: OR probability relationship

2.5 Cost Functions

Any design change, component improvement, or component redesign has costs attached to it which can be used to yield cost functions describing how difficult or costly the changes are. Without an accurate cost function, the process of making a design safer will likely result in an over-designed, and probably over-budget system. Although cost functions can combine all possible costs (such as retooling costs and redesign costs) in terms of some quantity, we consider cost functions exclusively from the perspective of failure rate. It should be noted that the quantity chosen will impact the characteristics of the cost function including range, asymptotes, and slope. A cost function for failure rate will have a positive asymptote at $x = 0$ and continue to $x = \infty$, as a result of the tendency for work to become increasingly difficult as fault rate of zero is approached. Figure 2.6 shows a representative cost function for failure rate based on a cost curve of $\frac{1}{x^{0.4}}$. The x-axis of Figure 2.6 is the failure rate of the system. As expected it grows in an asymptotic fashion as the failure rate approaches 0. The y-axis represents the relative difficulty of reducing the failure rate. Cost function plots

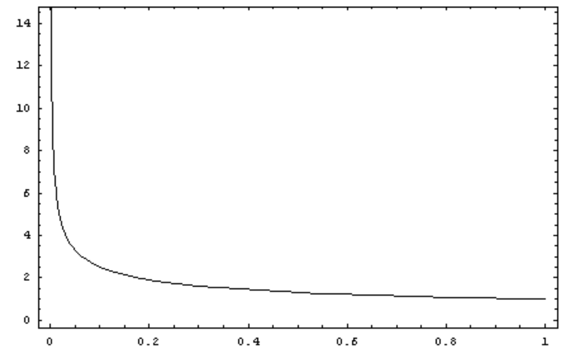

Figure 2.6: Representative Cost Function

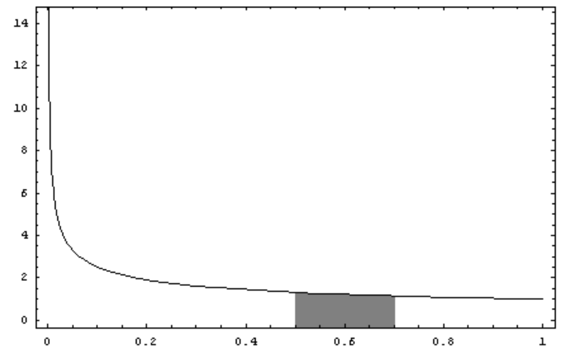

Figure 2.7: Example Calibration Cost Function

such as that shown in Figure 2.6 can be used as a comparison with other component cost functions.

In order for a cost function to be used to predict how much work is required for an improvement, there must be a way to infer a unit of measurement, such as man-hours, from the function. Such inference requires a reference function serving as a scaled version of the component cost function such that precisely how many man-hours are required to improve that component from a known initial value to a predefined final value is known. For software development companies operating at CMM level 3 or higher, such reference functions may be reliably available from measurements and metrics gathered from previous projects [18]. For companies without such references, a less precise approximation or educated guess may be substituted with a corresponding reduction in overall confidence of the reference function accuracy. The value of the integral of the cost function between those two points then can serve as the conversion factor between the integral of the cost function and man-hours to complete the work, and represents the cost required to improve the component from an initial probability of failure to the desired probability of failure. An example of how the conversion is applied is discussed in section 3.4.3. Figure 2.7 shows a representative calibration cost function. The shaded area represents the integral of the function that is equal to a known quantity of man-hours.

2.6 Predictors: What does not work

Since the early days of software engineering, metrics have been used as predictors of how something will behave [5]. Metrics such as McCabe's Cyclomatic Complexity Metric, Halstead's Software Volume Metric, and variations on the lines of code metric have been used to measure complexity and, by extrapolation, reliability [8, 15]. While metrics predate software engineering, their use as mainstream techniques have been limited as a result of the metrics failing to address an important requirement: to provide information that supports quantitative decisions throughout the software life cycle [5]. Fenton argues that the problem metrics tend to have is that they are based on naïve models. Using Feneton's example based on statistical data of auto accident fatalities, it is safest to drive in the months of January and February; however, experience tells differently. This is similar to the problem metrics have of using a simple criteria, such as size, to determine complexity. A byproduct of basing metrics on naïve models is that they are poor predictors of risk, which is a primary quantity that is sought to be reduced throughout the software life cycle [12]. An effective predictor is a metric that uses causal relationships to model a system, accounting for items, like the system's operating environment which have a large impact on the safety of a system. Currently, no major metric takes into account what environmental conditions the system is operating in, and hence none can accurately predict how a system will behave. As a result of most metrics providing a poor measure of risk, they tend to be ill suited as predictors of software safety since they do not measure the degree to which a program is fault-tolerant or safe.

Despite being unable to predict software safety, metrics are still a useful tool for designing safe software because they have been shown to accurately reflect the complexity of a program's structure [19]. Metrics can provide a measure of the degree of complexity of a module, which can help improve programming practice and the reliability of code. Metrics aid in the reduction of complexity and indirectly reduce the difficulty of adding fault tolerance. Such measurements are only valid when it is assumed that the outside environment cannot affect the input values of the system. If the environment can influence input values, as in safety-critical systems, then a metric based on a naïve model will ignore the interference and generate false results.

2.7 Predictors: What does work

The key for a metric being an effective estimate of safety is that the metric be able to reduce the input and output spaces to a range that is tractable. A way to accomplish this with software safety is to take advantage of the enumerated hazards list which is finite and, therefore, of tractable size. The enumerated hazards list is a product of the design of the system and is an explicit list of all system events that are considered hazards. Metrics such as Mean-Time to Hazard use fault trees to make predictions based on the likelihood of a particular hazard's occurrence [12]. The main alternative to fault trees, for design time analysis, are formal methods which involve mathematically proving that a program is safe [6, 14]. Although rigorous, program proving is difficult, costly, manpower intensive, and is typically required only in the most critical safety systems such as nuclear weapons fire control software. Formal methods such as program proving have been a part of software engineering since its inception; however, these techniques have not been encouraged due to the difficulties of proving the correctness of even simple programs. In 1970, the line editing problem was introduced as a demonstrative vehicle to be proved correct by formal methods. The program consisted of 25 lines of Algol 60 code, and after several attempts at the proof, those involved found they were unable to detect all the faults through the use of formal methods [19]. The inability to find all of the faults through formal methods was traced primarily to incorrect and imprecise specifications.

2.8 Related Work

Current work focusing on software system safety involves development of more accurate causal models than the traditional accident-oriented models in use today. Efforts such as Leveson's work on a system-theoretic approach to modeling safety in software-intensive systems use systems theory approaches to model accidents resulting from component interaction rather than individual component failure [2, 10, 11]. Leveson's approach is similar to this Trident effort in its focus on the composite causes leading to a system failure, but unlike the approach of this Trident project, focuses on post-accident analysis rather than design-time safety improvement.

Fenton's research on causal models is similar to this work [5]. However unlike Fenton's work, which used Beysian Belief Nets to create the causal relationsips, this report focuses on using fault trees to provide a causal model to build a metric from.

This work is similar to Sullivan's approach to converting fault trees into combinatorial representations [20]. However, Sullivan's approach requires selectively reverse-engineering a specification from which well-formed inputs, as well as input from an oracle, are derived, and yields a large test suite that covers all inputs up to a given size. With the approach researched here, the input space is made manageable via the use of a cost function vector field, allowing a greatly reduced state space from which to select improvement to meet the targeted safety goal of the fault tree's root.

Finally, this work is similar to Coppit's effort [3] on evaluating the cost effectiveness of developing and validating complex safety-critical systems through dynamic fault tree analysis of fault-tolerant systems [4]. However, where Coppit focuses on the specification ambiguity regarding the simultaneous occurrence of dependent failure events, this work focuses on the failure probability of components leading to a specific hazard at the root of the fault tree regardless of any failure simultaneity [23].

Chapter 3

Theoretical Foundations

This chapter defines the theoretical underpinnings of this research, and examines how it interacts with the overall software life cycle. The first section discusses the basis for the concept of a "key node." The second section applies key nodes in the creation of a design-level testing technique similar to mutation testing. The third section develops an algorithm for improving the safety of a software system during the implementation and maintenance phases of the software life cycle. The fourth and final section examines an application of the methods in improving the safety of a software system.

3.1 Hypothesis and Validation of Key Nodes

A hypothesis related to system safety is that the analysis of a fault tree can be simplified through the identification of "key nodes." A key node is a node that requires multiple inputs to fail before the failure will affect the next part of the system. An analysis of the three primary relationship types (OR, XOR, and AND) found in fault trees shows that one completely meets the key node requirements and one partially does (Analysis in Appendix B). Of the three primary relationship types, the XOR relationship only qualifies as a key node when there are more than three inputs into the node, while the AND relationship always qualifies as a key node. Since the AND relationship always qualifies as a key node, it will be the only type focused on as a key node in this paper. Unlike the XOR or AND relationships, the OR relationship fails to meet the requirements of a key node because the if any one or more inputs enter a failure state (logical true) then the failure propagates to the next level.

3.2 Key Node Metric

When considering the design of a safety critical system, the verification of changes to a system's safety is important. The Key Node Metric is an attempt at providing such a measure. Section 3.2.1 discusses some of the reasons that the Key Node Metric is useful and why it was created. Section 3.2.2 gives an overview of how the metric works, while section 3.2.3 goes in depth to describe the mathematics behind the metric.

3.2.1 Purpose

The Key Node metric has been developed to provide a design tool to compare fault trees without requiring *a priori* knowledge of component reliability. Such a metric allows designers to proactively improve the safety of a system before final component selection or the completion of component reliability studies. The ability to improve system safety without the knowledge of component reliabilities is important in software engineering where "typical" component reliability values for a type of component are often unknown.

3.2.2 Basis

The creation of a metric requires that several things be known: What is the lower bound of the metric? What, if any, is the maximum of the metric? What criteria will be used for scaling between the minimum and maximum values of the metric? The first criterion, the lower bound, is found in a system where the failure of any component will cause a hazard. A fault tree based on this system is composed entirely of OR relationships and has a safety factor of 0, as seen in Figure 3.1. The second criterion, the maximum, is the inverse of first criterion, in which a system would fail if and only if every component failed. Numerically there is no universal upper bound; however, the upper bound for an individual tree can be calculated by running the metric on the mutation of the tree containing only AND relationships. The fault tree derived from such a system has all components connected by AND relationships, seen in Figure 3.2. The third criterion, the scaling factor, is less intuitive and is the subject of the remainder of this section.

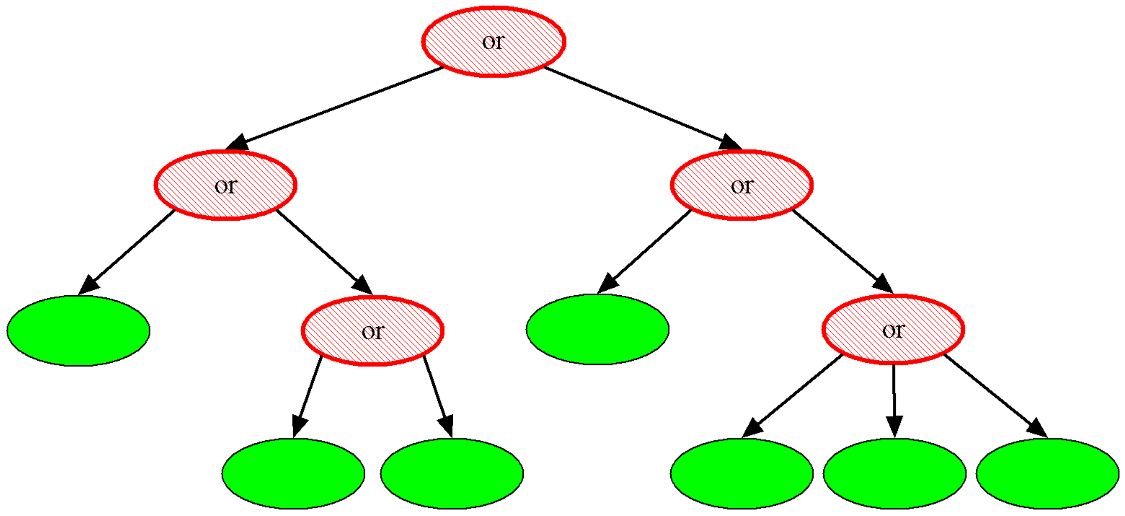

Figure 3.1: Fault Tree with no redundancy

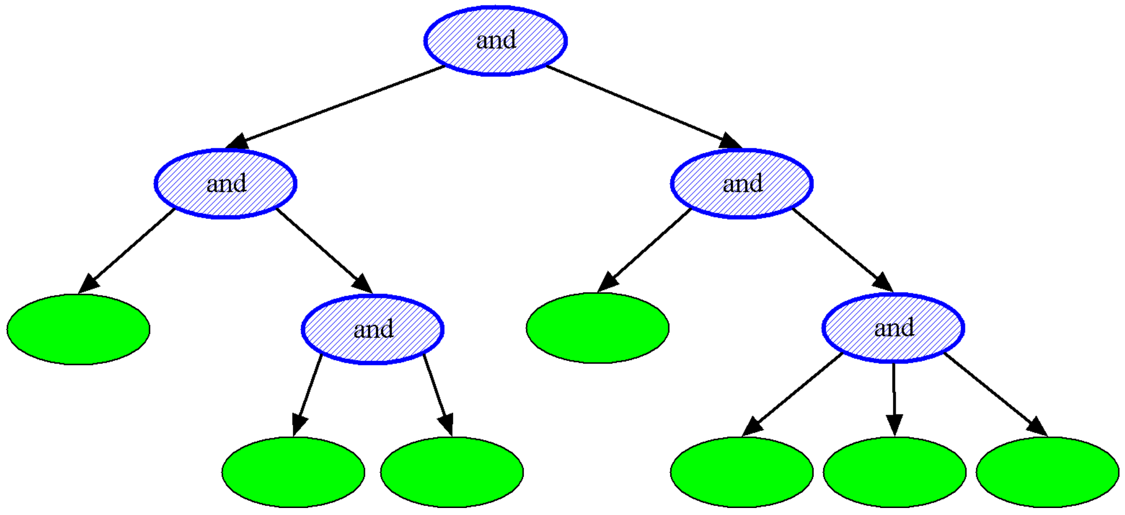

Figure 3.2: Fault Tree with complete redundancy

In order to develop the scaling factor criterion it is necessary to understand how the addition of key nodes impacts a fault tree. One way of evaluating the changes is to start with the tree in Figure 3.1 and change each relationship to an AND relationship until the tree looks like Figure 3.2. An analysis of the mutation of the tree in Figure 3.1 into the tree in Figure 3.2 reveals that key nodes provide a possible basis for a safety metric. The metric directly measures fault tolerance, which then can be extrapolated to predict safety, just as the extrapolation of complexity can be used to predict reliability. The first element used to develop the metric is the number of key nodes. Beyond the raw number of key nodes, the individual properties of each key node are also useful. The metric is divided into two segments: the first is the overall tree segment, which considers the number of key nodes. The second is the individual key node segment, in which the properties of each key node are factored in. The properties of key nodes include depth (or location relative to the root of the tree) and the size of the sub-tree rooted at the key node itself. A key node that has a smaller depth (higher in a tree) provides a greater amount of fault tolerance because it requires a greater number of failure events before the hazard can occur. This is identical to the effect derived from the size of the sub-tree rooted at a key node. Both properties must be included in the metric because it is possible that a fault tree will be unbalanced and a node with a lesser depth will not necessarily have a larger sub-tree.

3.2.3 Mathematical Foundations

This section discusses the mathematics involved in computing the metric value, S, of a fault tree, and contains fundamental definitions that are used to create the complete metric equation.

Definition 3.1. The height of a tree, h, is defined as the number of edges on the longest simple path from root to a leaf. A simple path is a path between two points containing no cycles.

Definition 3.2. The depth of a node, d_i, is defined as the number of edges from the root to node i.

Definition 3.3. The depth of the root of a tree is defined as: $d_{root} = 0$

As a result of Definition 3.3 the following are defined to prevent possible divisions by zero:

Definition 3.4. $d'_i = d_i + 1$

Definition 3.5. $h' = h + 1$

Definition 3.6. The size of sub-tree, c_i, is defined as the number of nodes in the tree rooted at node i, not including node i

Definition 3.7. The size of the sub-tree of a leaf, c_{leaf}, is defined to be 0

Definition 3.8. The size of the sub-tree of the root,c_{root}, is defined to be $n - 1$, where n is the number of nodes in the tree

The tree segment, ts, of the metric compares the number of key nodes and the total number of nodes in the tree.

$$ts = \frac{k}{n} \tag{3.1}$$

The individual node segment, ns_i, accounts for the relative depth of the key node and the relative size of the sub-tree rooted at the key node.

$$ns_i = \frac{h'c_i}{(n)(d'_i)} \tag{3.2}$$

The compilation of all the individual node segments creates the total node segment, ns:

$$ns = \sum_{i=0}^{k} \frac{h'c_i}{nd'_i} \tag{3.3}$$

Combining ts and ns yields:

$$S = \frac{k}{n} \sum_{i=0}^{k} \frac{h'c_i}{nd'_i} \tag{3.4}$$

Equation 3.4 simplifies to:

$$S = \frac{kh'}{n^2}\sum_{i=0}^{k}\frac{c_i}{d'_i} \tag{3.5}$$

Equation 3.5 is the form of the metric that will be used to compute the S values throughout the rest of the report.

3.3 Improvement Algorithm

In Section 3.3 the origin and theory behind the second part of this research, improvement algorithm, will be discussed in detail. Section 3.3.1 discusses what the improvement algorithm provides the developer. Section 3.3.2 discusses the methodology behind how the improvement works, while Section 3.3.3 presents the mathematics behind the algorithm.

3.3.1 Purpose

The purpose of the Improvement Algorithm is to provide an objective method of improving a system's safety by determining which components need to be improved and by what amount in order to achieve the desired increase in safety. The algorithm also provides an estimate of how many man-hours are required to meet the safety requirement, and thereby a tool for managing project resources.

3.3.2 Basis

In this section, we present an algorithm for improving the structure of a fault tree in a manner that improves (decreases) the probability of occurrence of the hazard at the tree root. The prerequisites of the improvement algorithm are that: there exists a fault tre to describe how a hazard can occur, there exists failure probabilities for each component, and there exists a goal failure probability, G.. The improvement algorithm is based on the reduction of any fault tree into a polynomial expression of degree g, where g is the number of leaves, by a post order traversal of the fault tree. $P(p)$ is the value of the polynomial expression at some point p. For any one value of P, there are an infinite number of sets of inputs. The objective of the algorithm is to pick a unique set of inputs to P, such that P is equal to a goal value. In order to select the unique set of inputs in this manner, additional information, such as a cost function for improving a nodes failure probability, is required.

Many hardware components have cost functions that describe how expensive (in man-hours) it is to improve the reliability of the components. Likewise, software components typically have historical data, or approximations, that can fill the role of cost functions in the improvement algorithm. The combination of the cost functions for all the components yields a vector field, V, in which each dimension of the vector field is the cost function of a component. An example is where the cost functions $c0$, $c1$, $c2$, $c3$, $c4$, $c5$, and $c6$ would create the vector field $< c0, c1, c2, c3, c4, c5, c6 >$. Figure 3.3 shows a vector field formed from combining two cost functions ($< \frac{1}{x^{0.4}}, \frac{1}{x^{0.6}} >$). The arrows point toward regions of greater safety The length of each arrow shows relative difficulty of improvement. The small arrow in the upper right corner of Figure 3.3 is in a region of low safety, and the small size represents how it is not difficult to improve the safety of a system at that point. However, the large arrow in the lower left corner of Figure 3.3 is in a region of high safety and the arrows large size represents the greater difficulty of improving the safety of a system in that region. Evaluating the vector field at $V(p)$, yields a vector that points toward the region of greater safety; which accounts for the different difficulties of improvement for each component.

The initially known quantities are the set in initial system proabilities p_i, the value of the fault tree's polynomial expression at the initial point $P(p_i)$, the value of the vector field at the initial point $V(p_i)$, and the desired value G. The object is to find a p_f, such that $P(p_f) = G$. Starting at the point p_i, the vector $V(p_i)$ will point towards the level curve defined by $P = G$. A scaling factor, k, is then used to stretch the vector $V(p_i)$ so that it intersects the level curve $P = G$. The point where $kV(p_i)$ intersects with $P = G$ is p_f. Using substitution, the polynomial expression P with g unknowns is changed to have one unknown, k, yielding, the new polynomial expression P_k. A 2-dimentional example, based

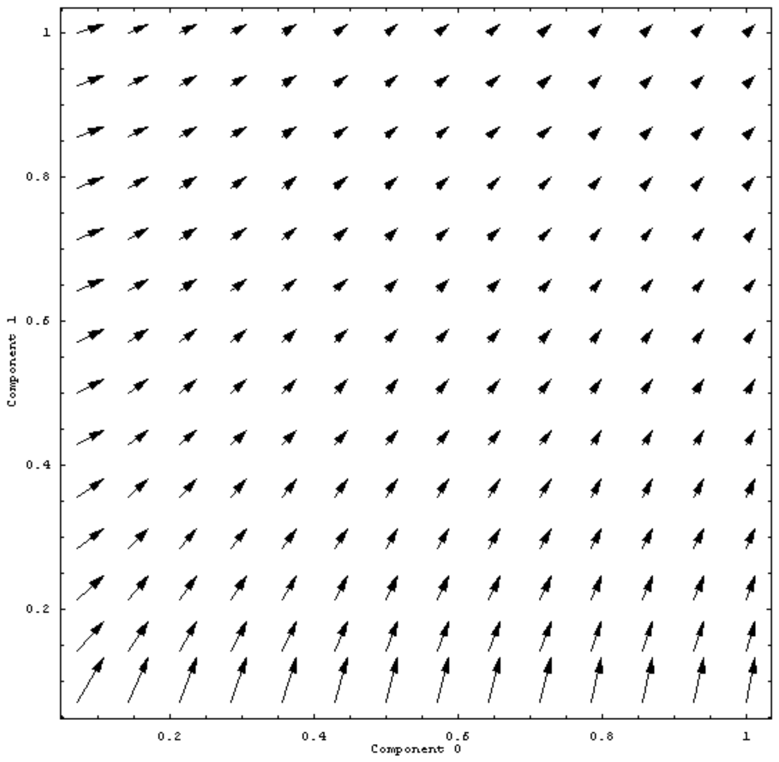

Figure 3.3: Sample Vector Field

on Equation 2.2, of the substitution is:

$$\begin{aligned}
P(a_i, b_i) &= a_i b_i \\
P(a_f, b_f) &= a_f b_f \\
kV(a_i, b_i) &= k < e, f > \\
a_f &= a_i - ke \\
b_f &= b_i - kf \\
P_k(a_f, b_f) &= (a_i - ke)(b_i - kf)
\end{aligned}$$

There will be at most g real roots of P_k. This is because that a polynomial of degree g will have a total of g real and imaginary roots. The real root of k that creates the smallest vector $kV(p_i)$ such that $p_i + kV(p_i)$ yields a result within the domain of $(0 \le p_f \le 1)$. Once

p_f is known, it can be compared to p_i to determine the percent change in the reliability of each component.

The man-hours required to improve a component from its initial reliability to its final reliability can be found by taking the integral of the cost function for that component from the initial reliability to the final reliability. The accuracy of the work estimate relies on how accurately the cost function models reality.

3.3.3 Mathematical Foundations

In 2-space, the polynomial expression P is of the forms:

$$P(x, y) = 1 - (1 - x)(1 - y) \quad (OR\ case) \tag{3.6}$$

$$P(x, y) = xy \quad (AND\ case) \tag{3.7}$$

The vector field V is of the form:

$$V(x, y) = < f(x), f(y) > \tag{3.8}$$

Using Equation 3.7 and the initial conditions of x_i and y_i:

$$P(x_i, y_i) = x_i y_i \tag{3.9}$$

If $P(x_i, y_i)$ is greater than G then:

$$V(x_i, y_i) = < a, b > \tag{3.10}$$

$$kV(x_i, y_i) = < ka, kb > \tag{3.11}$$

$$P(x_f, y_f) = x_f y_f \tag{3.12}$$

$$x_f = x_i - ka \tag{3.13}$$

$$y_f = y_i - kb \tag{3.14}$$

By substitution:

$$P(x_f, y_f) = (x_i - ka)(y_i - kb) \tag{3.15}$$

Equation 3.15 is then set equal to the goal, G:

$$G = (x_i - ka)(y_i - kb) \tag{3.16}$$

Rearranged:

$$G = k^2ab - kay_i - kax_i + x_iy_i \tag{3.17}$$

Solving for k. The smallest root of k that results in x_f and y_f in the domain [0,1] is selected as the scaling factor. Once x_f and y_f are known percent change can be found by:

$$\%\Delta x = (x_f - x_i)/x_f \tag{3.18}$$

$$\%\Delta y = (y_f - y_i)/y_f \tag{3.19}$$

3.4 An Application of the Improvement Algorithm

In this section, we apply the improvement algorithm to a maintenance scenario of safety critical software in which a client has both requirements related to the overall safety of the system, and cost functions that can be applied to system components.

3.4.1 Example Improvement Scenario

Consider the scenario of a customer requesting that additional features be added to a safety critical software system. The customer specifies that the safety of the modified system be no worse than the safety of the original system. The original system was verified by program proving, however due to cost considerations the progress of program proving is undesirable for the modified version. The improvement algorithm offers an alternative to program proving in such a case.

3.4.2 Prerequisites

In order to use the improvement algorithm in this scenario, four pre-conditions must be met. First, the probability of each hazard occurring in the initial system is required to be known. This may involve the construction of fault trees and the subsequent computation of each probability. Second, fault trees must be constructed for the modified system for each hazard. Third, the present failure probability of each of the components must be known. The fourth, and final prerequisite, is that cost functions be available for the improvement of each component. Table 3.1 gives the initial failure probabilities of all the variables (chosen arbitrarily), cost functions, and desired failure probability, G, of the system.

variable	value	variable	value
$p0$	0.35	$c0$	$\frac{1}{x^{0.4}}$
$p1$	0.29	$c1$	$\frac{1}{x^{0.6}}$
$p2$	0.41	$c2$	$\frac{1}{x^{0.43}}$
$p3$	0.32	$c3$	$\frac{1}{x^{0.56}}$
$p4$	0.49	$c4$	$\frac{1}{x^{0.78}}$
$p5$	0.25	$c5$	$\frac{1}{x^{0.58}}$
$p6$	0.20	$c6$	$\frac{1}{x^{0.35}}$
P	0.55	G	0.3

Table 3.1: Initial Fault Tree Values

3.4.3 Verifying the Modified System

In order to verify that the situation is improved through the application of the improvement algorithm, the probability of each hazard occurring in the modified system is calculated. Based upon these calculations, a set of hazard conditions is created in which the modified system does not meet the safety requirements. The improvement algorithm is then executed on the fault tree for each hazard in the set. This results in a list of components that need to be improved and by what amount in order to meet the desired failure probability.

The rest of this section examines the improvement process for a sample hazard, $h0$. The fault tree in Figure 3.4 represents all possible ways for the software system to cause $h0$. The cost functions for each of the seven inputs, represented by the leaf nodes, are labeled $c0$ through $c6$. The initial failure probabilities of each component are similarly labeled $p0$ through $p6$. Node 0 on Figure 3.4, for example, would have a cost of $c0$ and a probability of failure of $p0$.

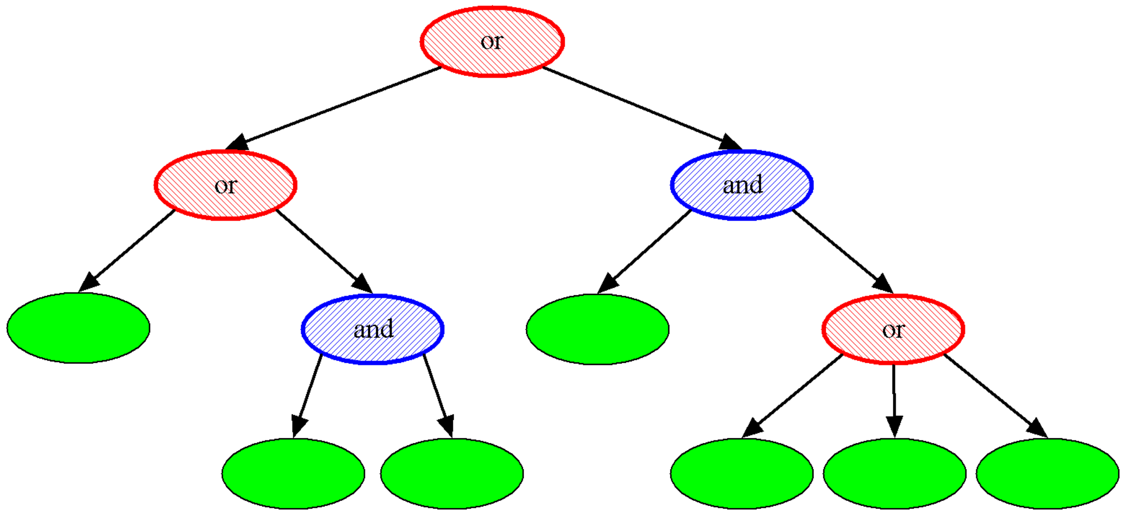

Figure 3.4: Fault Tree for Sample Hazard

The fault tree probability equation, P, is constructed via a post order traversal of the tree in Figure 3.4, resulting in:

$$\begin{aligned}
P(p0, p1) &= (p0)(p1) \\
P_a &= P(p0, p1) \\
P(p4, p5, p6) &= 1 - (1 - p4)(1 - p5)(1 - p6) \\
P_b &= P(p4, p5, p6) \\
P(p0, P_a) &= 1 - (1 - p0)(1 - P_a) \\
P_d &= P(p3, P_b) \\
P(p3, P_b) &= (p3)(P_b) \\
P_d &= P(p3, P_b) \\
P(P_c, P_d) &= 1 - (1 - P_c)(1 - P_d) \\
P_e &= P(P_c, P_d) \\
P_e &= P(p0, p1, p2, p3, p4, p5, p6) \\
P(p0, p1, p2, p3, p4, p5, p6) &= 1 - (1 - (1 - (1 - p0)(1 - (p1)(p2)))) \\
&\quad (1 - (p3)(1 - (1 - p4)(1 - p5)(1 - p6))))
\end{aligned}$$

After substituting the values from Table 3.1 into this equation, the initial probability of hazard is computed to be 0.55. Since a failure probability of 0.55 is worse than the desired value of 0.3 the next step of the algorithm is executed in which a vector field is created from the seven cost functions such that:

$$\begin{aligned}
V(c0, c1, c2, c3, c4, c5, c6) &= \\
< \frac{1}{x^{0.4}}, \frac{1}{x^{0.6}}, \frac{1}{x^{0.43}}, \frac{1}{x^{0.56}}, \frac{1}{x^{0.78}}, \frac{1}{x^{0.58}}, \frac{1}{x^{0.35}} >
\end{aligned}$$

The value of V at the initial point $(p0, p1, p2, p3, p4, p5, p6)$, provides a vector whose direction is towards the desired failure probability curve. Once the vector is known, it is combined with P to create an equation only in terms of the scaling factor k. The scaling factor k, is the smallest positive root of P_k; in this case the smallest positive root is

0.083321. The scaling factor is then substituted into individual component equations to find the failure probability of each component necessary to have the desired failure probability for the system. The failure probability of each component required to meet the system requirements is shown in Table 3.2.

Assuming that the calibrated cost function is the cost function $c0$ and that the integral of $c0$ from 0.7 to 0.5 is equal to 10 man-hours, the cost to improve the system can be predicted by using this value to convert the integrals of the other cost functions. In the case of this example, as shown in Table 3.2, 100.7 man-hours would be required to improve the failure probability from 0.55 to 0.3.

variable	value	% improvement	cost (man-hours)
$p0$	0.22	37.1	8.8
$p1$	0.11	62.1	19.9
$p2$	0.29	29.3	7.7
$p3$	0.16	50.0	14.7
$p4$	0.34	30.6	12.2
$p5$	0.06	76.0	24.4
$p6$	0.05	75.0	13
P	0.3	45.5	100.7

Table 3.2: Improved Fault Tree Values

Chapter 4

Experimental Results

Chapter 4 discusses the experiments that were conducted using the metric and improvement algorithm. The first part of the chapter discusses the work done to evaluate the Key Node Safety Metric. The rest of the chapter is dedicated to the discussion of the validation of the improvement algorithm.

4.1 Validation of Key Node Metric

The discussion of validating the Key Node Metric is divided in to two parts. The first part discusses the procedure used to validate the metric. The second part presents a summary of the results of the validation effort.

4.1.1 Procedure

The Key Node Metric has been tested by comparing 70 fault trees organized into ten sets of seven trees. Each set consists of an initial fault tree, which is the set baseline, and then six trees which are mutations of the initial tree. Three of the mutations were designed to improve the safety of the system represented by the respective fault tree by randomly converting an OR node to an AND node, while the other three focused on decreasing safety by converting an AND node into an OR node. The addition or removal of an AND node will increase or decrease system safety respectively, because AND nodes represents points of fault tolerance or redundancy. For each of the ten sets, the metric was first run on the base tree and then on the remaining trees in the set. After the metric was run on each set, the results were compiled and analyzed to see if the metric properly discerned which trees were the improved trees and which were the degraded trees. A correct metric will properly classify each tree as improved or degraded when compared to the initial tree. The characteristics of the initial fault trees are summarized in Table 4.1. Some of the characteristics include lack of balance in the tree, ratio of key nodes to total number of nodes, and ratio of key nodes to internal nodes. Figures of all 70 test cases are in Appendix B.

	Total Nodes	Key Nodes	Internal Nodes	Max Depth	Min Depth
Set 1	12	2	5	3	2
Set 2	14	3	6	3	2
Set 3	14	3	6	3	2
Set 4	19	4	8	5	2
Set 5	24	3	10	4	3
Set 6	22	5	9	4	2
Set 7	19	3	8	3	2
Set 8	18	3	7	4	2
Set 9	27	4	12	5	2
Set 10	37	5	16	5	2

Table 4.1: Summary of Metric Test Case Initial Trees

4.1.2 Results

The Key Node Metric was able to analyze the entire test case and to differentiate which trees were the positive mutations (improve safety) and which were the negative mutations (decrease safety). The metric had a 100% success rate in completing this operation. The numerical results are summarized in Table 4.2. Table 4.3 displays relative improvement compared to the initial tree of each set.

	Degraded Trees				Improved Trees		
	Tree 4	Tree 5	Tree 6	Initial Tree	Tree 1	Tree 2	Tree 3
Set 1	0.07	0.02	0.00	0.18	0.43	0.35	0.69
Set 2	0.07	0.11	0.12	0.22	0.46	0.35	1.36
Set 3	0.17	0.05	0.17	0.30	0.45	0.56	0.77
Set 4	0.14	0.22	0.24	0.38	0.93	0.53	0.56
Set 5	0.18	0.09	0.12	0.29	0.43	0.41	0.45
Set 6	0.13	0.34	0.33	0.47	0.69	0.60	0.67
Set 7	0.06	0.04	0.07	0.12	0.30	0.21	0.19
Set 8	0.04	0.11	0.11	0.19	0.41	0.35	0.34
Set 9	0.05	0.12	0.10	0.17	0.48	0.25	0.37
Set 10	0.16	0.19	0.19	0.31	0.55	0.43	0.40

Table 4.2: S-values from all Metric test cases

	Degraded Trees			Improved Trees		
	Tree 4	Tree 5	Tree 6	Tree 1	Tree 2	Tree 3
Set 1	−0.106	−0.157	−0.176	0.255	0.171	0.509
Set 2	−0.150	−0.127	−0.124	0.234	0.115	0.083
Set 3	−0.156	−0.116	−0.102	0.238	0.129	1.136
Set 4	−0.126	−0.241	−0.126	0.153	0.262	0.469
Set 5	−0.245	−0.158	−0.145	0.553	0.145	0.179
Set 6	−0.111	−0.202	−0.176	0.132	0.115	0.156
Set 7	−0.342	−0.125	−0.135	0.218	0.135	0.197
Set 8	−0.063	−0.085	−0.055	0.174	0.085	0.070
Set 9	−0.157	−0.080	−0.085	0.219	0.157	0.147
Set 10	−0.116	−0.052	−0.066	0.309	0.083	0.206

Table 4.3: Difference Relative to the Initial Tree of each Set

4.2 Validation of Improvement Algorithm

The discussion of validating the improvement algorithm is divided into three sections. The first section discusses the procedure of how the improvement algorithm was validated. The second section describes the test cases used to in validating the improvement algorithm. Finally the third section discusses the results of the validation effort.

4.2.1 Procedure

The validation of the improvement algorithm is a two step process: the first step verifies the correctness of the algorithm, and the second step checks whether the algorithm is faster than two types of iteration methods. While the run time is not a major consideration for fault trees the size of the one examined here, it is a consideration for trees with hundreds of leaves. The first iteration method, termed the random selection method, uses a random number generator to select which input will be improved. The second method or "Worst Node selection" method selects the input with the largest failure probability for improvement. The second experiment of tests used the same input as the first experiment. The first step of validation for the improvement algorithm consisted of verifying that there exists a solution for the scaling factor, k, that provides an x_f and a y_f that are within the domain while being a solution to $P(x_f, y_f) = G$. The second segment of the test procedure required evaluating the fault tree fifty times for each selection method and recording the time to evaluate the tree each time. The average of each set of timing runs was evaluated to determine which selection method provided the most accurate solution in the minimum amount of time. The average of each data set was taken in order to compensate for variances in the data. A better selection method would have a lower average execution time; however, if it also had a large variance it would demonstrate that the selection method was better suited for some data sets than others.

4.2.2 Test Cases

The validation test verifies that the improvement algorithm can produce a set of final input probabilities to create a system with the desired safety from a set of initial conditions and desired safety. It is not an attempt to formally prove that the algorithm will always produce a correct output, but instead is a demonstration that the algorithm works. A formal proof of the algorithm is considered in the future work section. The test input is based on the fault tree in Figure 4.1 and the input values and cost functions found in Table 4.4:

The second part of the test plan consisted of determining if the improvement algorithm

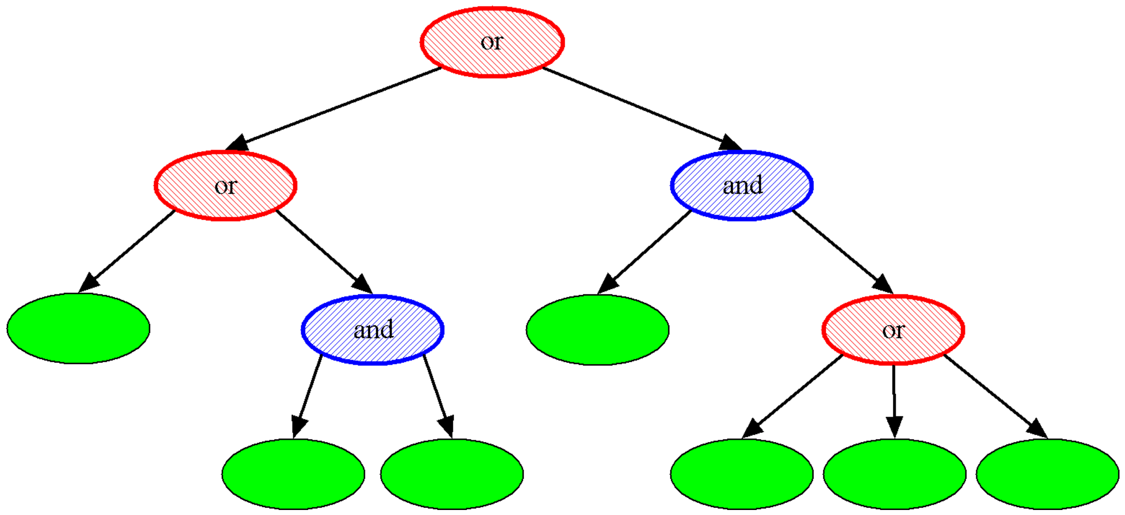

Figure 4.1: Fault Tree for Test Hazard

variable	value	variable	value
$p0$	0.35	$c0$	$\frac{1}{x^{0.4}}$
$p1$	0.29	$c1$	$\frac{1}{x^{0.6}}$
$p2$	0.41	$c2$	$\frac{1}{x^{0.43}}$
$p3$	0.32	$c3$	$\frac{1}{x^{0.56}}$
$p4$	0.49	$c4$	$\frac{1}{x^{0.78}}$
$p5$	0.25	$c5$	$\frac{1}{x^{0.58}}$
$p6$	0.20	$c6$	$\frac{1}{x^{0.35}}$
P	0.55	G	0.3

Table 4.4: Initial Fault Tree Values

would be faster than the Random and Worst Node Selection methods while maintaining the desired accuracy.

4.2.3 Results

Table 4.5 contains the results of the validation test of the improvement algorithm. For comparison reasons the table also contains the initial values of the inputs and calculations of change. The results of the second part of the experiment can be found in Table 4.6. The S1 and S2 designations in the table signify that the data was collected with a step value of 0.999 or 0.9999 respectively.

variable	value	% improvement
$p0$	0.22	37.1
$p1$	0.11	62.1
$p2$	0.29	29.3
$p3$	0.16	50.0
$p4$	0.34	30.6
$p5$	0.06	76.0
$p6$	0.05	75.0
P	0.3	45.5

Table 4.5: Improved Fault Tree Values

	Avg Runtime (msec)	Avg Final Probability	% Error
Improvement Algorithm	< 0	0.3	0
Worst Node S1	12	0.299921	0.026
Random S1	17	0.299947	0.018
Worst Node S2	1220	0.299992	0.0027
Random S2	1798	0.299996	0.0013

Table 4.6: Comparison of Improvement Algorithm with Iterative Methods

Chapter 5

Analysis of Results

This chapter will evaluate the results to the experiments presented in the previous chapter. Section 5.1 provides an analysis of data presented in Section 4.1.2. Included in this analysis is a discussion on the quality of the data sets used in the experiment. Section 5.2 analyzes the data from Section 4.2.3 to determine if the results verify the improvement algorithm.

5.1 Key Node Metric Analysis

Based on the evidence in Table 4.3, it is clear the metric was able to discern which mutations were improvements and which were degradations. There are, however, several anomalies in the data. The largest is from tree 3 of set 2, which has an S value from the Key Node Metric of 1.36. Rather than dismiss the value as an abnormality, since it is the only value over 1, this tree was investigated further and found to be unique since it is the only test tree in which the root is also a key node. Upon examination of the metric equation, it is apparent that the only way for an S value to be greater than 1 is to have the root be a key node. While the range of S is from 0 to ∞, any value over 1 suggests that the fault tree in question is not an accurate model of its system due to the implications of the root being a key node. With the root node being a key node, the implication is that the hazard can only be caused one way and that one way requires multiple components to fail simultaneously. While it is not impossible to have such a system, it would not be expected that a complex system would have one and only one possible way for a hazard to occur. However, this would be a possibility if the fault tree reflected only a single subsystem rather than then entire system as a whole.

Another issue with the data sets is a result of the small number of internal nodes in sets 1, 2, and 3. The problem is that it is not possible to perform six mutations that yield unique trees without any double mutations due to the insufficient number of internal nodes. A double mutation is where two nodes are simultaneously changed from ORs to ANDs or vice versa. Of the three trees affected by this mutation, only two, trees 1 and 2, show any obvious impact on the S value. The S value changes dramatically between the base and trees 3 and 6 of those two trees. While these small sets allowed the testing of two special cases (no key nodes and the root as a key node) they did not provide the same quality of data as other sets in discerning the relative safety of the trees. In sets 4 through 10, in which each tree has an adequate numbers of interior nodes, the difference between the S values of the initial tree and the mutated tree are smaller. This is expected because the mutation of a single node should generally not dramatically change the S value.

5.2 Improvement Algorithm Analysis

While the first set of experiment results is not a proof that the improvement algorithm will always work, it does demonstrate that it can work. The characteristics of the scaling factor found by the algorithm depend on the characteristics of the cost function used. In the case of the functions used in this report k will always be positive. This is a result of the fact that the magnitude of the $V(p)$ will always be positive. Since the general equation in which the values are substituted already has the substituted values subtracted from the initial value, all that is required is to reduce or enlarge the substituted value till the result satisfies the equality.

As shown in Table 4.6 the improvement algorithm is the fastest of the three methods. Comparing the iterative based algorithms to the improvement algorithm shows that the improvement algorithm gives an exact solution every time in under a millisecond whereas the iterative methods take between 12 and 20 times longer to solve the problem to within three hundredths of a percent. A test to increase accuracy of the iterative methods to within three thousands of a percent required over 1000 times as long as using the improvement algorithm. An interesting note is that while the Worst Node Selection algorithm was significantly faster it did tend to be farther from the desired value than the random selection.

While both the Random and Worst Node selection algorithms did not find exact answers, the approximations were universally better than the desired value. While on the surface this may seem good, it actually implies that if the system was constructed using these decisions, the system would be over-engineered and cost more than it should have. In a very large project many small budget overruns could cause the project to be greatly over budget or canceled.

A problem with the data is that the run time of the improvement algorithm was on average less than 1 millisecond, which is the finest granularity of timing available in Mathematica. This does leave some ambiguity with the run time of the improvement algorithm. It may be possible to attain finer granularity of timing in a UNIX environment.

Chapter 6

Conclusions

In today's military, unsafe software can cause mission-critical failures such as the inadvertent release of weapons, loss of vehicle control, and propulsion failure. An understanding of how a software system.s component structure impacts the potential of the system to enter unsafe states is critical for such applications. This research presented two novel ways to assist software engineers seeking to increase the safety of software systems. To assist in designing safer software, a key node safety metric based on key nodes and fault trees was presented. Additionally, a safety improvement algorithm was developed to assist in the process of implementing and then maintaining software with improved safety.

A key node safety metric was proposed as a method of predicting the relative safety between different versions of software modules. The metric was developed from a heuristic analysis of fault tree structure, and calculates a value based on inherent fault tree properties including key node height, size of key node sub-trees, and the number of key nodes. The experiments used to evaluate the metric demonstrated that the metric can correctly predict which of several design variants is safer. The only requirements for using the key node safety metric are that the fault trees being compared must have the same root hazard, and that the fault trees consist internally of AND and OR nodes.

A safety improvement algorithm was presented for situations in which leaf node failure probabilities, and the cost to improve these failure probabilities, are either known or can be estimated. The safety improvement algorithm focuses on mitigating system failure by reducing fault trees into polynomial expressions. Cost improvement functions are applied to identify nodes where improvement will effectively lead to increased system safety, and results in a deterministic solution to achieving a desired safety improvement goal. Experiments were conducted comparing the improvement algorithm to two iterative approaches. The first iterative approach used randomly selected nodes for improvement, and the second used the nave approach of selecting nodes with the largest failure probability. The results of these experiments demonstrated that the safety improvement algorithm is significantly faster than the selected iteration methods. Further, the improvement algorithm provides an exact solution whereas the iterative approaches tend to overshoot the safety goal resulting in wasted improvement costs.

Areas of future work center around extending both the safety metric and the safety improvement algorithm to include XOR relationships in order to increased the utility of these tools since currently only AND and OR relationships are supported. Another area focuses on increasing the efficiency of the improvement algorithm through the use of stream functions to find the optimal path between the initial and goal failure probabilities. Additionally, the use of Lagrange multipliers are being considered to avoid the scalability issues raised when finding the roots of high degree polynomials. Other future work includes developing the improvement algorithm from a research project into an integrated tool that can be readily used by software developers. This will involve integrating the safety metric and the improvement algorithm into computer aided design tools so that safety analysis can be made an automated part of the software development process.

Bibliography

[1] F. P. Brooks, V. Basili, B. Boehm, E. Bond, N. Eastman, A. K. Jones, M. Shaw, and C. A. Zraket. *Report of the Defense Science Board Task Force on Military Software*. Department of Defense, Office of the Under Secretary of Defense for Acquisition, Washington, DC, Sep 1987.

[2] P. Checkland. *Systems Thinking, Systems Practice*. John Wiley & Sons, New York, 1981.

[3] D. Coppit and K. Sullivan. Sound methods and effective tools for engineering modeling and analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 198–207, Portland, Oregon, 2003.

[4] J. B. Dugan, S. Bavuso, and M. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, Sep 1992.

[5] N. Fenton and N. Martin. Software metrics: Roadmap. In *Future of Software Engineering*, pages 357–370, Limerick, Ireland, 2000.

[6] M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability.* John Wiley & Sons, New York, 1995.

[7] F. R. Frola and C.O. Miller. *System Safety in Aircraft Management.* Logistics Management Institute, Washington D.C., Jan 1984.

[8] M.H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[9] J.C. Huang. An approach to program testing. *ACM Computing Surveys*, 8(3):113–128, Sep 1975.

[10] J. Leplat. Occupational accident research and systems approach. In J. Rasmussen, K. Duncan, and J. Leplat, editors, *New Technology and Human Error*, pages 181–191. John Wiley & Sons, New York, 1987.

[11] N. Leveson. A systems-theoretic approach to safety in software-intensive systems. *IEEE Transactions on Dependable and Secure Computing*, Jan 2005.

[12] N. G. Leveson. Software safety: Why, what, and how. *ACM Computing Surveys*, 18(2):125–163, Jun 1986.

[13] N. G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, Feb 1991.

[14] R. Lutz. Software engineering for safety: A roadmap. In *Future of Software Engineering*, pages 213–224, Limerick, Ireland, 2000.

[15] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(12), 1976.

[16] H. D. Mills. *On the statistical validation of computer programs.*, volume Red. 72-6015. IBM Federal Systems Division, Gaithersburg, MD, 1972.

[17] P. Naur and B. Randell, editors. *Software Engineering*, Brussels, 1969. Scientific Affairs Division, NATO.

[18] M. C. Paulk, M. B. Curtis, B. Chrissis, and C. Weber. *Capability Maturity Model for Software, Version 1.1.* Software Engineering Institute, 1993.

[19] Stephen R. Schach. *Object-Oriented and Classical Software Engineering.* McGraw Hill, Boston, 2002.

[20] K. Sullivan and et al. Software assurance by bounded exhaustive testing. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 133142, Boston, 2004.

[21] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook.* U.S. Nuclear Regulatory Commission, Washington D.C., 1981.

[22] J. M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, New York, 1998.

[23] G. Weinberg. *An Intorduction to General Systems Thinking.* John Wiley & Sons, New York, 1975.

[24] M. Young and R. N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the 11th International Conference on Software Engineering*, pages 53–62, Pittsburgh, Pensylvania, 1989.

Appendix A

Glossary

AND Boolean function that is only true when all inputs are true.

Calibrated Cost Function Cost function that a known amount of work is required to move from an initial point to a final point.

Cost Function Function that estimates the relative difficulty to improve an object relative to other objects.

Failure Probability The probability that a component will fail in a given amount of time.

Fault Tree A system's engineering construct that maps the relationship between a hazard and its basic causes.

Goal probability The desired failure probability of a system.

Hazard An event that causes loss of life, loss of functionality, or other catastrophe that would impair system performance.

Improvement Algorithm Algorithm presented in this report to objectively improve the safety of a system.

Key Node A node of a fault tree that prevents multiple failure events from propagating higher in the tree.

Key Node Metric Metric that uses key nodes to predict the relative safety of a fault tree.

Man-hours Unit of work that describes how much work a person can do in one hour.

Metric A software tool used to make a prediction about a quality of the software from internal characteristics. Often used to predict reliability.

Mutation An alteration to a structure to create a new unique structure.

OR Boolean function that is true if any input is true.

Random Selection Method A selection method used in the report where an input is selected at random for improvement.

Safety Metric A metric that is used to predict safety.

Worst Node Selection A selection method used in this report where the node with the worst failure probability was selected for improvement.

XOR Boolean function that is true when only one input is true.

Appendix B

Key Node Analysis

The analysis of the fundamental relationships of AND, OR, and XOR to find which best describes a key node is based on the following definition:

Definition B.1. A key node is a node in a fault tree that will only allow a failure to propagate to the next higher level in the tree if and only if multiple failure conditions exist.

Using this definition and the truth tables of each relationship (Tables B.1, B.2, and B.3) it is clear that the OR relationship does not qualify as a key node. This is because if any input enters a failure condition (logical true), then the failure will propagate the the next level. XOR is less clear. From the truth table it is apparent that if a single failure condition does occur then the failure will propagate; however, if multiple simultaneous failures occur they would be blocked. While this behavior does not meet the strict definition of a key node, it does not completely rule out XORs being key nodes. Building on the XOR analysis, the analysis of the AND relationship is is simple. As long as there is a single input that is not in a failure condition, then the failure will not propagate. This behavior fulfills the definition of a key node.

0	0	F
0	1	T
1	0	T
1	1	T

Table B.1: OR Truth Table

0	0	F
0	1	T
1	0	T
1	1	F

Table B.2: XOR Truth Table

0	0	F
0	1	F
1	0	F
1	1	T

Table B.3: AND Truth Table

Appendix C

Metric Test Cases

This section contains all 70 of the fault trees in the Key Node Metric test set. Each page represents a set of ten related trees. To the left of the initial tree in the center are the negatively mutated trees. To the right are the positively mutated trees. Under each tree is the value produced by the Key Node Metric when run on that tree.

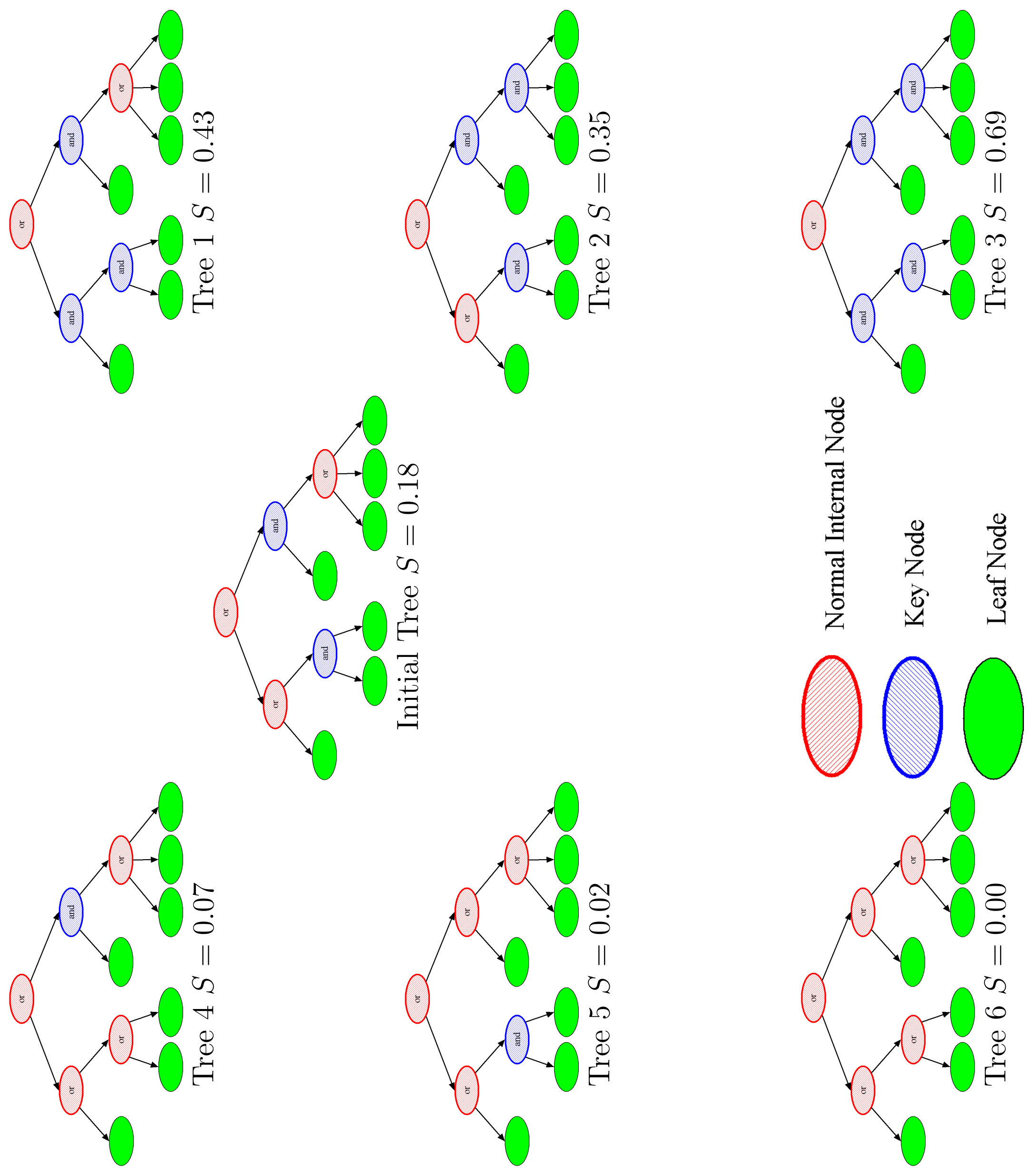

Figure C.1: Set 1

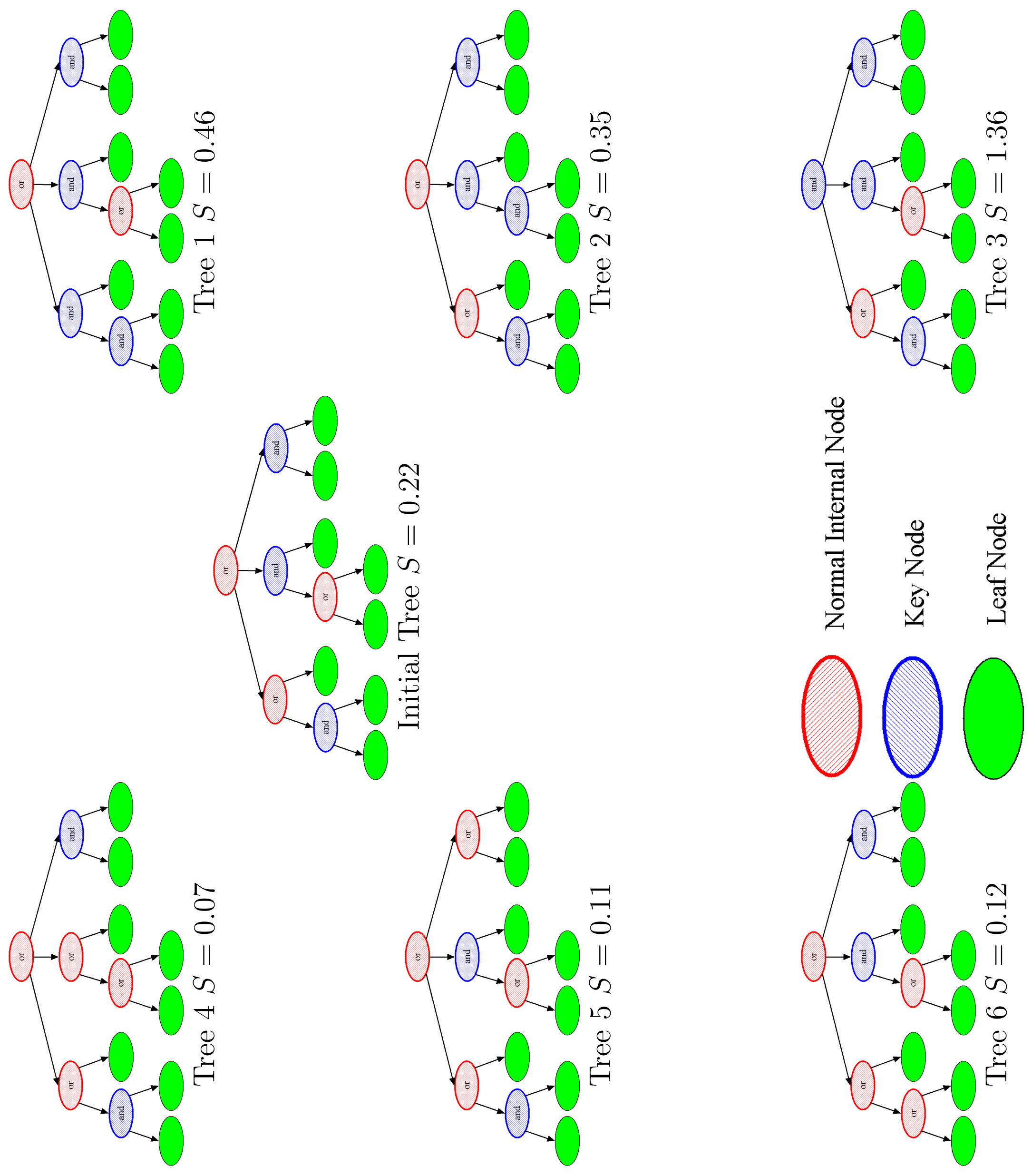

Figure C.2: Set 2

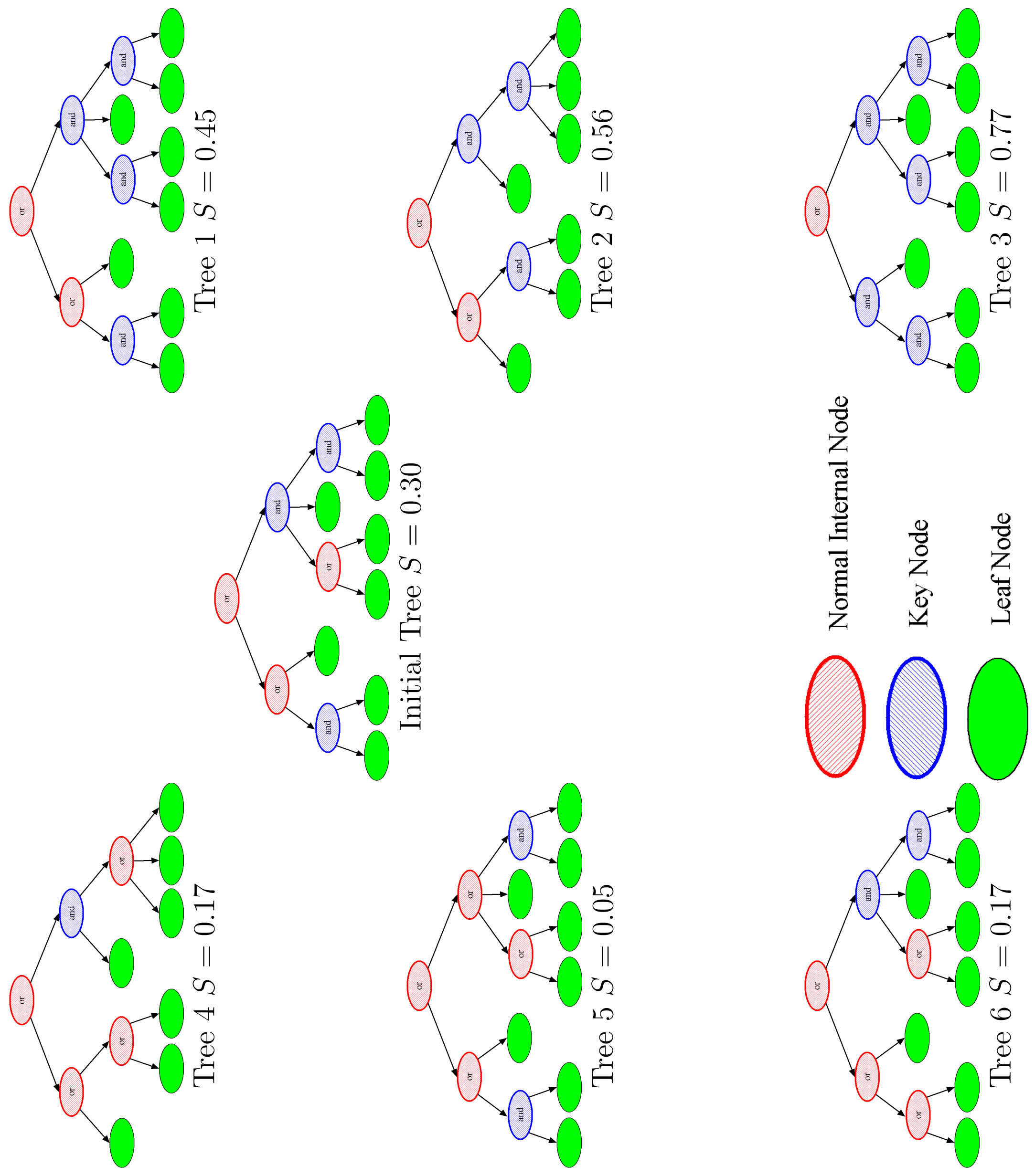

Figure C.3: Set 3

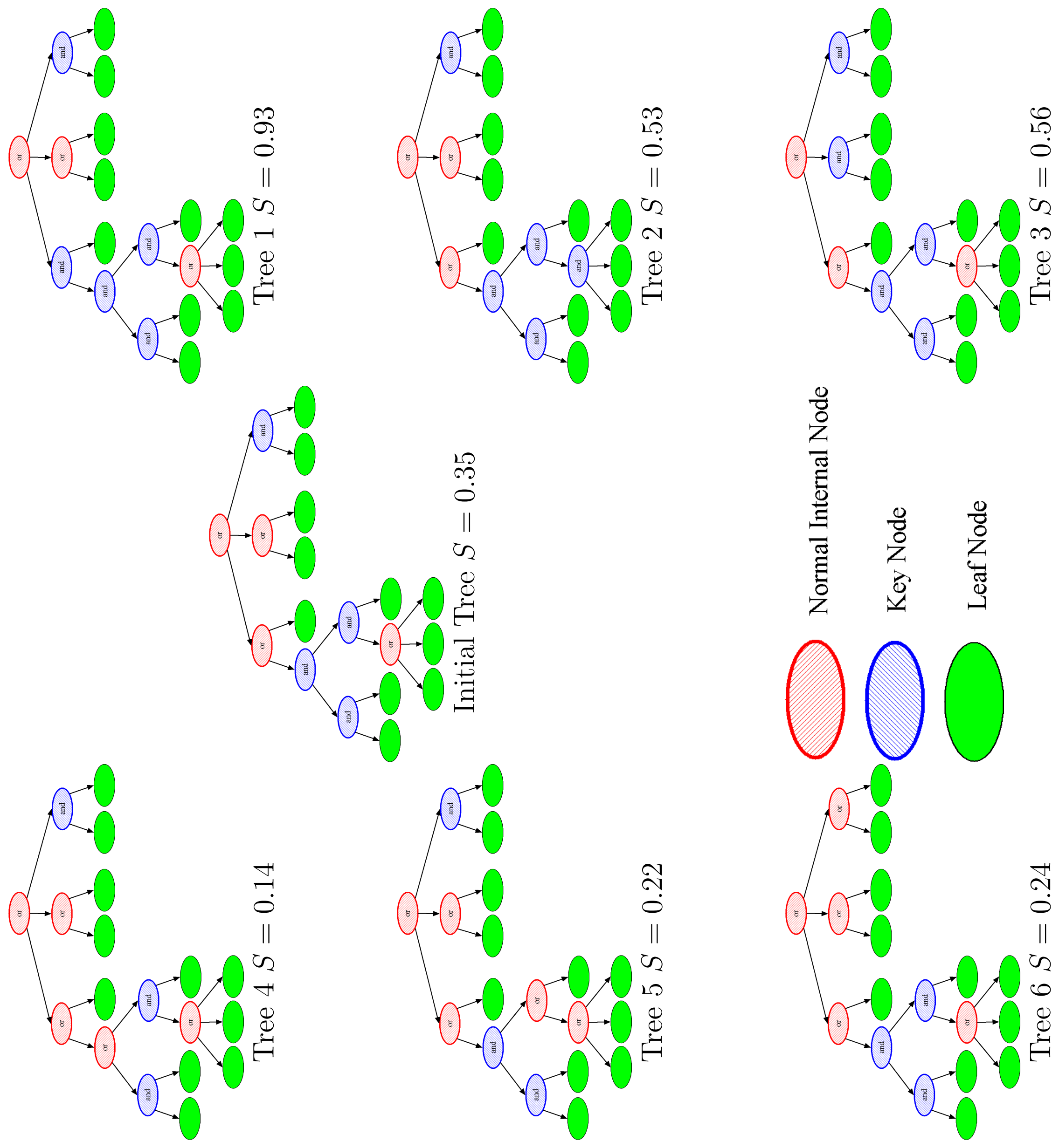

Figure C.4: Set 3

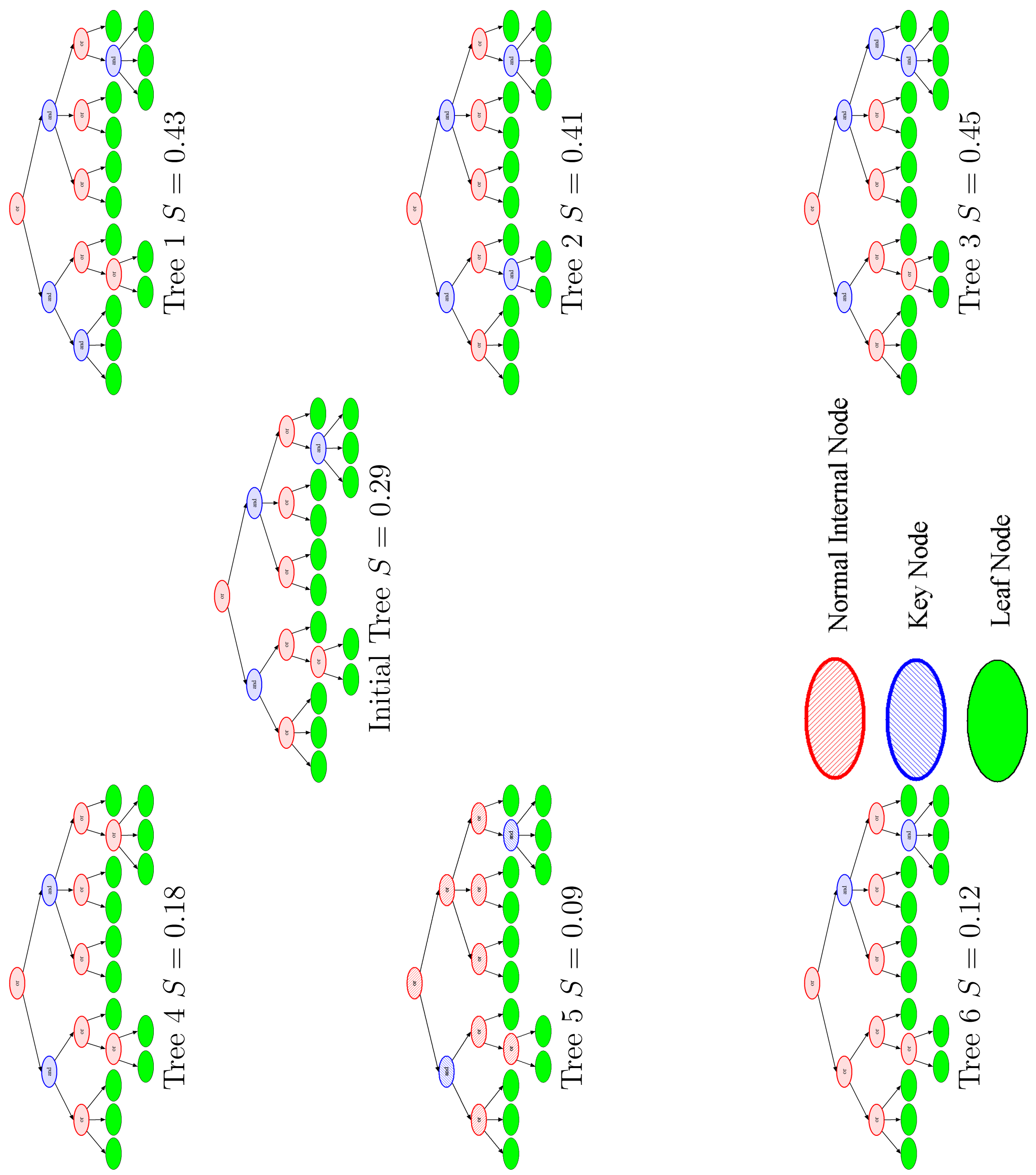

Figure C.5: Set 5

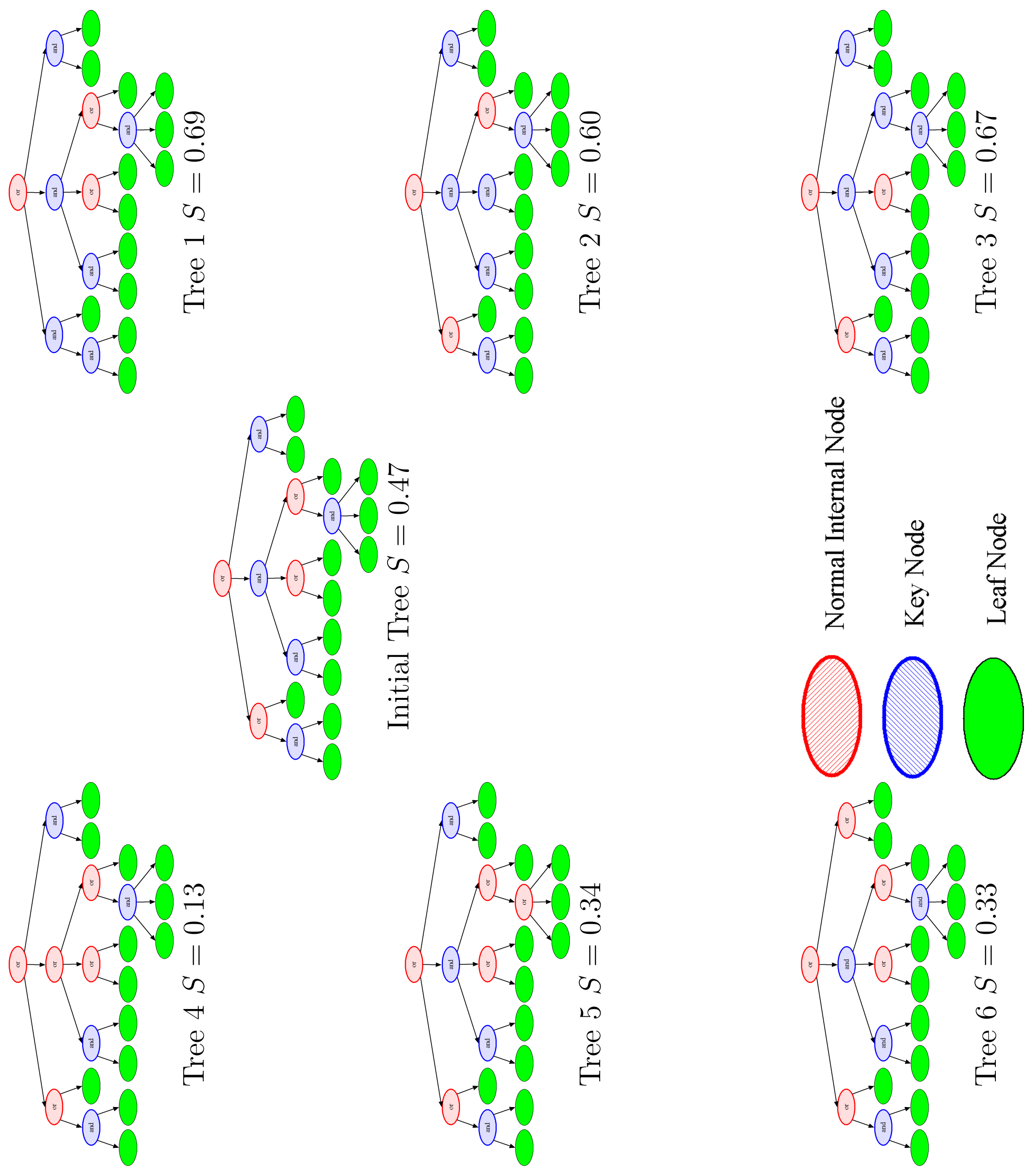

Figure C.6: Set 6

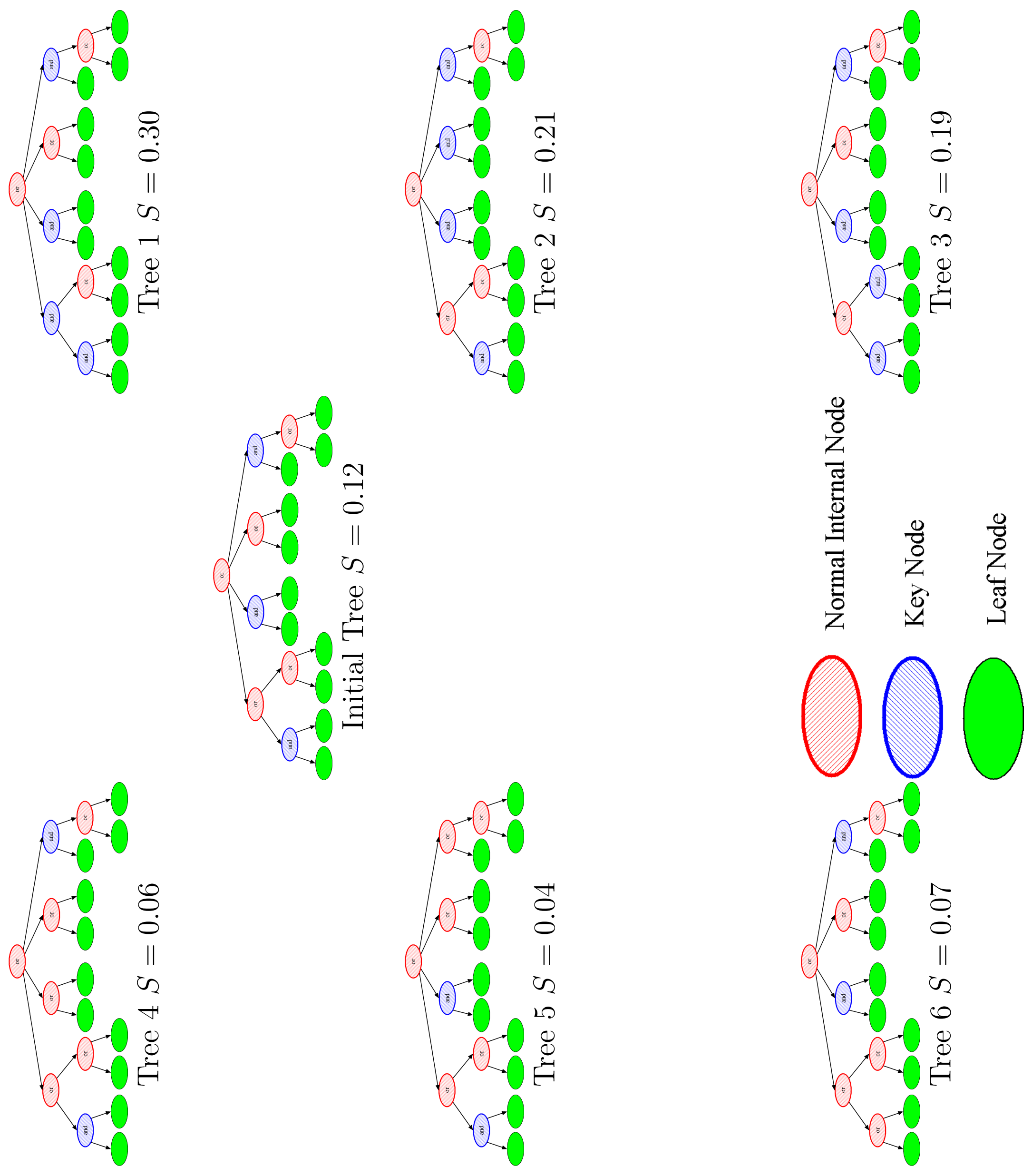

Figure C.7: Set 7

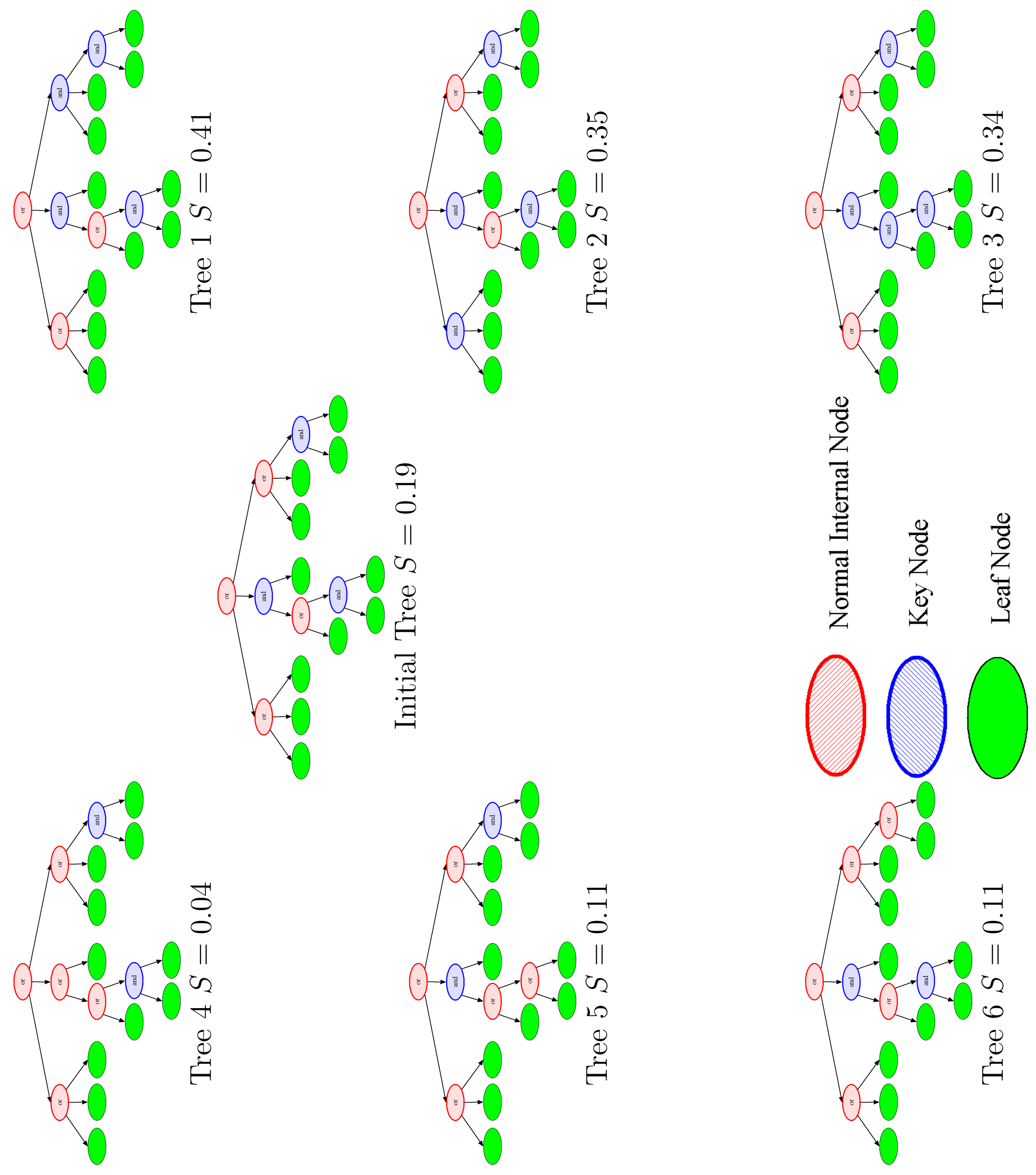

Figure C.8: Set 8

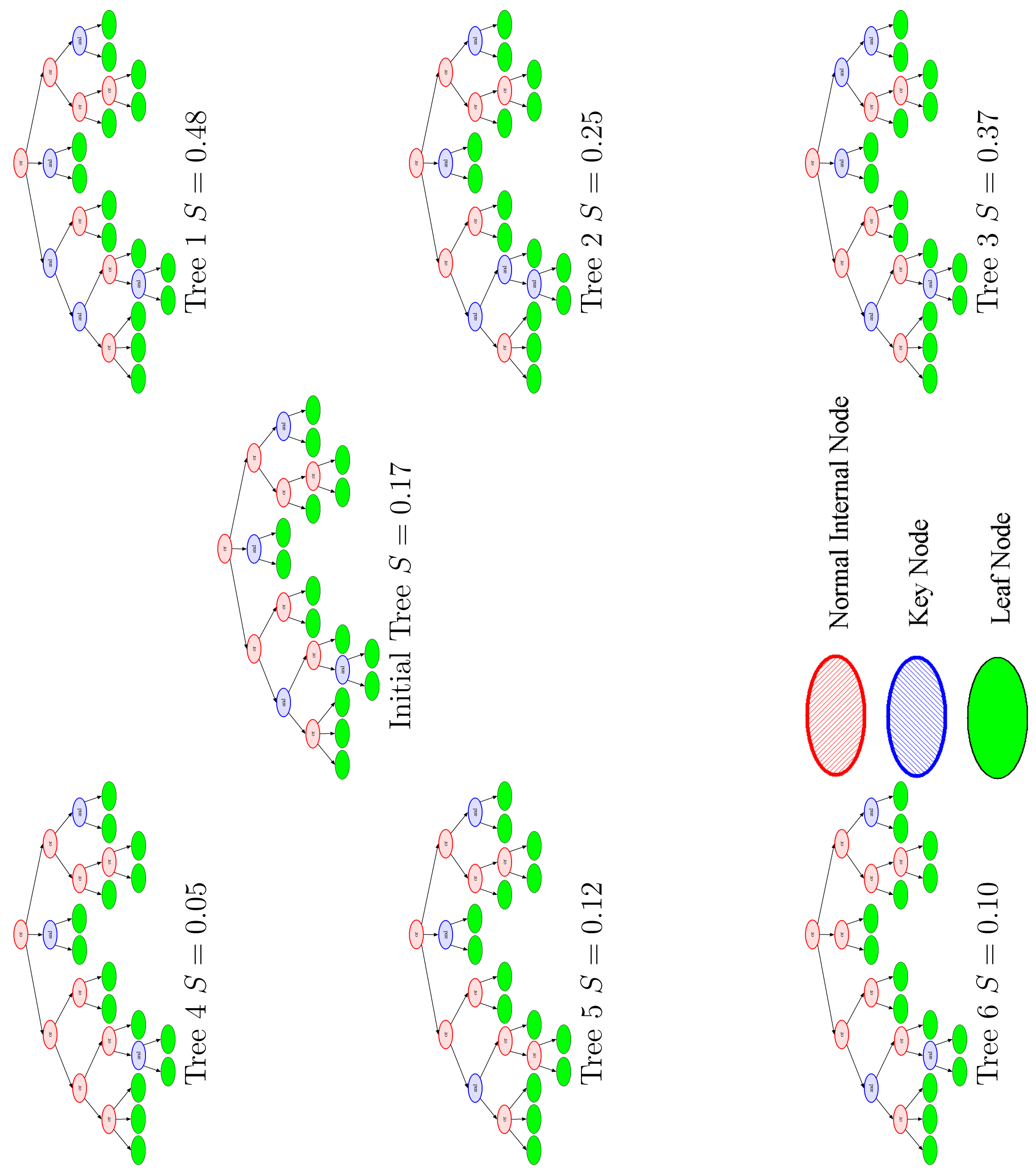

Figure C.9: Set 9

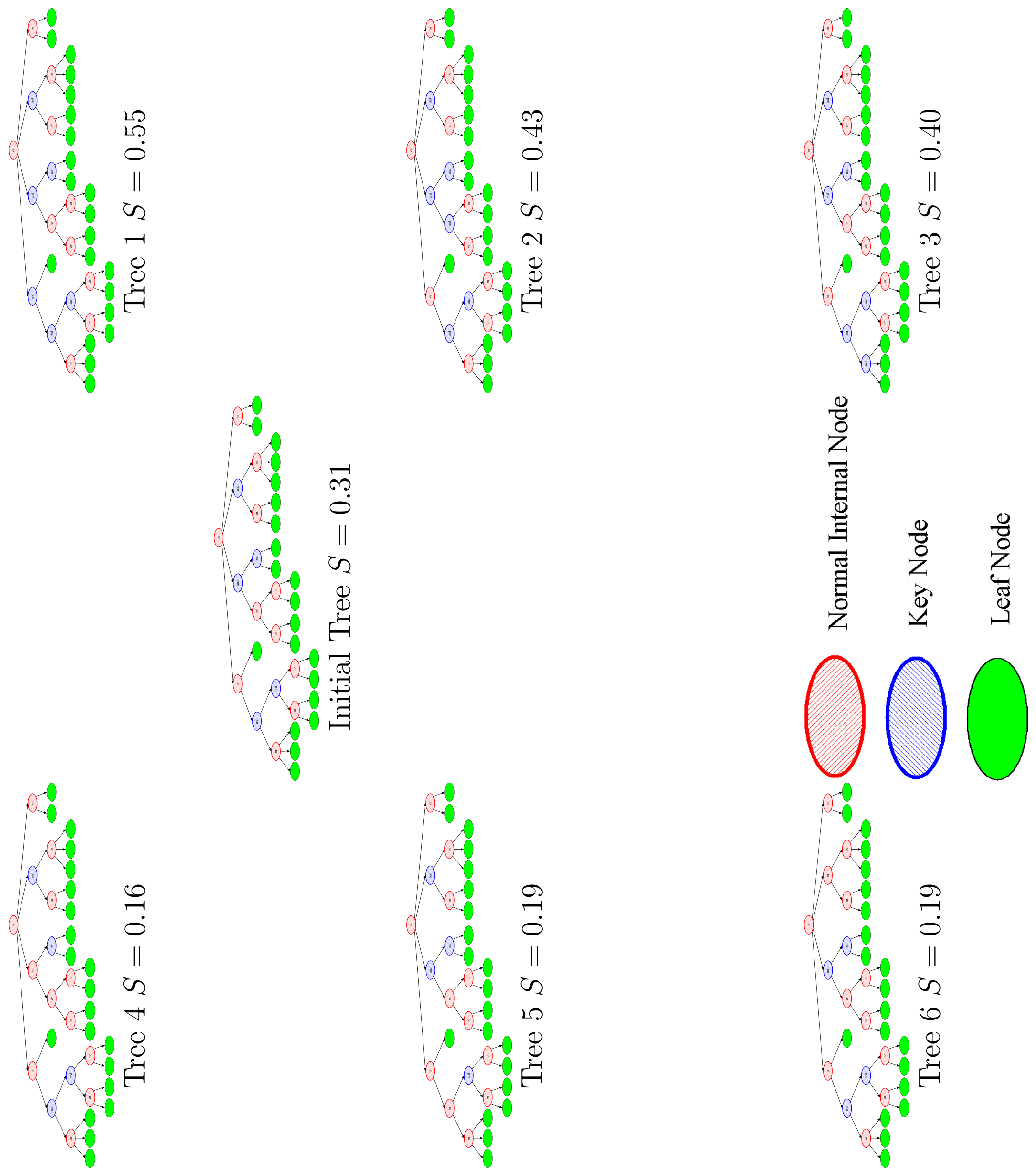

Figure C.10: Set 10

Appendix D

Source Code

The following is the Mathematica code used to generate all the timing and accuracy data in the report. It was written in Mathematica 5.1.

D.1 Improvement Algorithm

```
goal = 0.3;
n6 := 1 - (1 - n9)(1 - n10)(1 - n11);
n4 := n7*n8;
n2 := n5 * n6;
n1 := 1 - (1 - n3)(1 - n4);
n0 := 1 - (1 - n1)(1 - n2);
For[i = 0, i < 50, i++,
  x = 0.35;
  y = 0.29;
  z = 0.41;
  w = 0.32;
  v = 0.49;
  t = 0.25;
  s = 0.20;
  v0 = 1/x^.4;
  v1 = 1/y^.6;
  v2 = 1/z^.43;
  v3 = 1/w^.56;
  v4 = 1/v^.78;
  v5 = 1/t^.58;
  v6 = 1/s^.35;
  f0 = 1;
  f1 = 1;
  f2 = 1;
  f3 = 1;
  f4 = 1;
  f5 = 1;
  f6 = 1;
  n3 = x - v0*f0*k;
  n7 = y - v1*f1*k;
  n8 = z - v2*f2*k;
  n5 = w - v3*f3*k;
  n9 = v - v4*f4*k;
  n10 = t - v5*f5*k;
  n11 = s - v6*f6*k;
  Clear[k];
  a = Timing[b = Solve[goal == n0, k]];
  ]
```

D.2 Worst Node Selection Method

```
n6 := 1 - (1 - n9)(1 - n10)(1 - n11);
n4 := n7*n8;
```

```
n2 := n5 * n6;
n1 := 1 - (1 - n3)(1 - n4);
n0 := 1 - (1 - n1)(1 - n2);
For[i = 0, i < 50, i++,
  n11 = 0.20;
  n10 = 0.25;
  n9 = 0.49;
  n8 = 0.41;
  n7 = 0.29;
  n5 = 0.32;
  n3 = 0.35;
  goal = 0.3;
  step = .999;
  a = Timing[While[n0 = goal,
        If[n3 > n7,
          If[n3 > n8,
             If[n3 > n5,
              If[n3 > n9,
                If[n3 > n10,
                  If[n3 > n11, n3 *= step, n11 *= step], If[n10 > n11,
n10 *= step, n11 *= step]
                  ],
                 If[n9 > n10,
                  If[n9 > n11, n9 *= step,
        n11 *= step], If[n10 > n11, n10 *= step, n11 *= step]
                  ]
                ],
               If[n5 > n9,
                If[n5 > n10,
                  If[n5 > n11, n5 *= step, n11 *= step], If[n10 >
                 n11, n10 *= step, n11 *= step]
                  ],
                If[n9 > n10,
                  If[n9 > n11, n9 *= step, n11 *= step], If[n10 > n11,
n10 *= step, n11 *= step]
                  ]
                ]
              ],
            If[n8 > n5,
              If[n8 > n9,
                If[n8 > n10,

            If[n8 > n11, n8 *= step, n11 *= step], If[n10 > n11,
              n10 *= step, n11 *= step]
                  ],
                If[n9 > n10,
                   If[n9 > n11, n9 *= step, n11 *= step],
                  If[n10 > n11, n10 *= step, n11 *= step]
                  ]
                ],
               If[n5 > n9,
                If[n5 > n10,
                  If[n5 > n11, n5 *= step, n11 *= step], If[n10 > n11,
                 n10 *= step, n11 *= step]
                  ],
                If[n9 > n10,
                   If[n9 > n11, n9 *= step, n11 *= step],
                  If[n10 > n11, n10 *= step, n11 *= step]
                  ]
                ]
              ]
            ] ,
          If[n7 > n8,
             If[n7 > n5,
              If[n7 > n9,
                If[n7 > n10,
                  If[n7 >
                    n11, n7 *= step, n11 *= step], If[
                      n10 > n11, n10 *= step, n11 *= step]
                  ],
                 If[n9 > n10,
                  If[n9 > n11, n9 *= step, n11 *= step], If[n10 > n11,
n10 *= step, n11 *= step]
                  ]
                ],
               If[n5 > n9,
                If[n5 > n10,
                  If[n5 >
                  n11, n5 *= step, n11 *= step], If[n10 > n11,
n10 *= step, n11 *= step]
                  ],
                If[n9 > n10,
                  If[n9 > n11, n9 *= step, n11 *= step], If[n10 > n11,
              n10 *= step, n11 *= step]
                  ]
                ]
              ],
            If[n8 > n5,
              If[n8 > n9,
                If[n8 > n10,
                  If[n8 > n11, n8 *= step, n11 *= step], If[n10 > n11,
n10 *= step, n11 *= step]
                  ],
                If[n9 > n10,
                   If[n9 > n11, n9 *= step, n11 *= step],
                  If[n10 > n11, n10 *= step, n11 *= step]
                  ]
                ],
               If[n5 > n9,
                If[n5 > n10,

                If[n5 > n11, n5 *= step, n11 *= step], If[n10 > n11,
n10 *= step, n11 *= step]
                  ],
                If[n9 > n10,
                   If[n9 > n11, n9 *= step, n11 *= step],
                  If[n10 > n11, n10 *= step, n11 *= step]
                  ]
                ]
              ]
            ]
          ]
        ];
      ]
  ]
```

D.3 Random Selection Method

```
n6 := 1 - (1 - n9)(1 - n10)(1 - n11);
n4 := n7*n8;
n2 := n5 * n6;
n1 := 1 - (1 - n3)(1 - n4);
n0 := 1 - (1 - n1)(1 - n2);
For[i = 0, i < 50, i++,
  n3 = 0.35;
  n7 = 0.29;
  n8 = 0.41;
  n5 = 0.32;
  n9 = 0.49;
  n10 = 0.25;
  n11 = 0.20;
  v0 = Random[Integer, {0, 6}];
  v1 = Random[Integer, {0, 6}];
  While[v0 == v1, v1 = Random[Integer, {0, 6}]];
  v2 = Random[Integer, {0, 6}];
  While[(v0 == v2) || (v1 == v2), v2 = Random[Integer, {0, 6}]];
  v3 = Random[Integer, {0, 6}];
  While[(v0 == v3) || (v1 == v3) || (v2 == v3),
v3 = Random[Integer, {0,6}]];
  v4 = Random[Integer, {0, 6}];
  While[(v0 == v4) || (v1 == v4) || (v2 == v4) || (v3 == v4),
v4 = Random[Integer, {0, 6}]];
  v5 = Random[Integer, {0, 6}];
  While[(v0 == v5) || (v1 == v5) || (v2 == v5) || (v3 == v5) || (v4 == v5),
v5 = Random[Integer, {0, 6}]];
```

```
  v6 = Random[Integer, {0, 6}];
  While[(v0 == v6) || (v1 == v6) || (v2 == v6) || (v3 == v6)  ||
(v4 == v6) || (v5 == v6), v6 = Random[Integer, {0, 6}]];
  goal = 0.3;
  step = .999;
  sel := Function[tmp, Which[
    tmp == 0,  n3 *= step, tmp == 1, n7 *= step, tmp == 2, n8 *= step,
tmp == 3, n5 *= step, tmp == 4, n9 *= step, tmp == 5, n10 *= step,
tmp == 6, n11 *= step]];
  a = TimeConstrained[Timing[
        While[(J = n0) = goal,
            rand = Random[Integer, {0, 6}];
            Which[rand == 0,  sel[v0], rand == 1, sel[v1],
        rand == 2, sel [v2], rand == 3,  sel[
  v3], rand == 4, sel[v4], rand == 5, sel[v5], rand == 6, sel[v6]];
            ];
        ], 1, {{Infinity}}];
  ]
```