An Algorithm for Improving System Safety via Software Fault Trees

Jones, Sean A. Needham, Donald M.(Advisor)

USNA-CS-TR-2005-05

May 26, 2005

An Algorithm for Improving System Safety via Software Fault Trees

S. Jones and D. Needham
Computer Science Department,
United States Naval Academy
Annapolis, Maryland 21402 U.S.A.
Ph: 1.410.293.6809 Fax: 1.410.293.2686
sean.jones@acm.org
needham@usna.edu

Abstract

Analysis of software fault trees exposes hardware and software failure events that can lead to unsafe system states, and provides insight on improving safety throughout each phase of a system's development. Although fault trees can be pruned for low severity and low probability nodes, few techniques exist for systematically improving system safety by focusing on cost analysis of a system's fault tree nodes.

In this paper, we present an algorithm for system failure mitigation, supportive of continuous software evolution, based on the reduction of a fault tree into a polynomial expression of degree g, where g is the number of inputs. We combine cost functions that model the expense of improving component reliability into a vector field which provides a measurement of the degree of difficulty of system improvement. The gradient of the vector field is evaluated for vectors providing steep assent towards the area of greatest safety improvement, which in turn provides guidance on improving design time system safety. We provide an example application of our improvement algorithm, and examine improvement verification of the resulting system modifications.

1. Introduction

In 1955 only 10 percent of weapons systems required software, but by 1981 over 80 percent of such systems required software support [6]. 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. In such systems, the use of software has been justified through the large number of benefits such as flexible programming, increased capability, greater reusability, and decreased deployment costs. The only alternative is an inflexible hardware-only system which must be redesigned for each new system or major system change and tends to be expensive. Another concern with the increased role of software control systems is that the fast paced environments in which they run precludes manual intervention and require automatic failover to backup systems [6]. 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.

2. Background

In this section we review work related to our research area, discuss system safety versus system reliability, examine cost functions as a tool for predicting the cost of system improvement, and explore fault tree concepts relevant to our safety improvement algorithm.

2.1. 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 [5, 1, 4]. Leveson's approach is similar to our work in its focus on the composite causes leading to a system failure, but unlike our approach, focuses on post-accident analysis rather than design time safety improvement.

Our work is similar to Sullivan's approach to converting fault trees into combinatorial representations [9]. 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 our approach, 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, our work is similar to Coppit's effort [2] on evaluating the cost effectiveness of developing and validating complex safety-critical systems through dynamic fault tree analysis of fault-tolerant systems [3]. However, where Coppit focuses on the specification ambiguity regarding the simultaneous occurrence of dependent failure events, our 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 [11].

2.2. Safety vs. Reliability

Safety and reliability are two closely related terms when considering software systems. While it is often true that a safe system is also a reliable system, the inverse does not necessarily hold since system specifications dictate the difference between a safe system and a reliable system. If the specifications include the necessary safety requirements, then a system which is reliable will also be safe. Since this is often not true, the differentiation of safe and reliable is required [7].

The reliability, rather than safety, of individual components of a system are typically considered, because it is often assumed that components are simple enough to be viewed as a basic input. If a component is too complicated, or could be a safety concern, then it is no longer viewed as a black box. In this case, the component's fault tree is integrated into the system fault tree forming a composite tree that can accommodate failure probabilities for both the hardware and software of a system.

2.3. 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 result in an over-designed, probably over-budget system. Although cost functions can combine all possible costs (such as retooling costs, redesign costs) in terms of some quality, we consider cost functions exclusively from the perspective of failure rate quality. It should be noted that the quality chosen impacts 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 $x = \infty$. Figure 1 shows a representative cost function for failure rate. The x-axis of Figure 1 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 such as that shown in Figure 1 can be used as a comparison with other component cost functions.

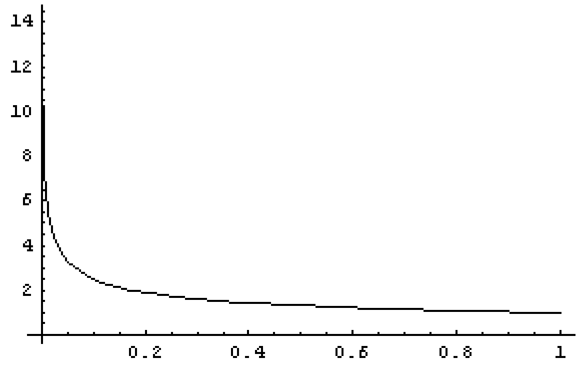

Figure 1. Representative Cost Function

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 measurement unit, 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 [8]. For companies without such references, a less precise approximation or educated guess may be substituted with a corresponding reduction in confidence in 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. An example of how the conversion works can be found in section 4.3. Figure 2 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.

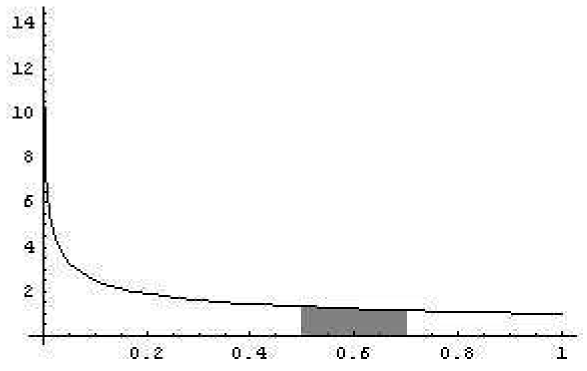

Figure 2. Example Calibration Cost Function

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 [10]. Fault trees are composites of fundamental entities known as events and gates, which form the nodes and connections of the tree. The root of a fault tree is the pre-defined hazard event which the designer of the tree seeks to either prevent or minimize the probability of occurring. A hazard event is any event in a system that can cause 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 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. 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.

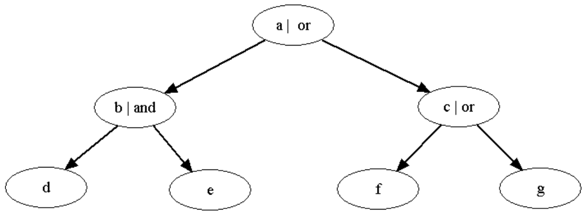

Figure 3. Example Fault Tree

In Figure 3, the leaf nodes are labeled d, e, f, and g; and the internal nodes are a, b, and c. 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 failure condition. When node a is in a failure condition, the hazard described by the fault tree occurs.

In compute the probability of a fault tree entering a hazard state, we must consider equations for AND or OR node systems. Equation 1 is for an AND node system such as in Figure 4 and the equation 2 is for an OR node system such as in Figure 5. In equation 1 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 [10].

$$a = bc \tag{1}$$

$$a = 1 - (1 - b)(1 - c) \tag{2}$$

3. Improvement Algorithm

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 requires: that a fault tree exists to describe how a hazard can occur, the failure probability of every component, and the goal failure probability, G, are known or can be approximated. The

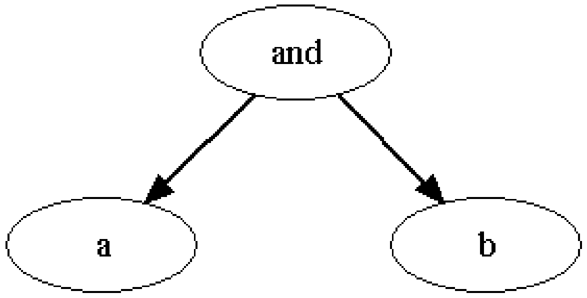

Figure 4. AND probability relationship

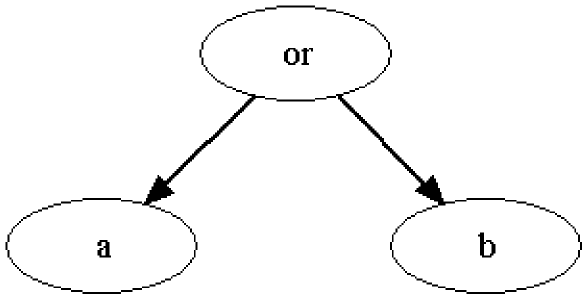

Figure 5. OR probability relationship

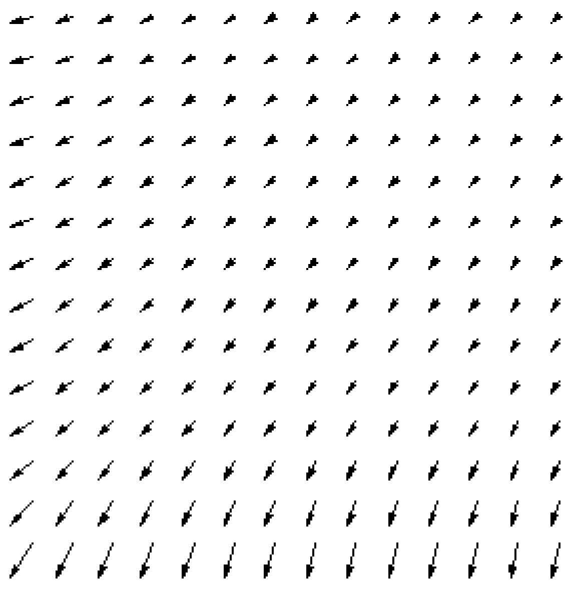

Figure 6. Sample Vector Field

improvement algorithm is based on the reduction of any fault tree into a polynomial expression of degree g, where g is the number of inputs, by a post order traversal of the fault tree. For any one value of P there are an infinite number of sets of inputs. The objective of our 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 cost functions, 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 our 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 $< c0, c1, c2, c3, c4, c5, c6 >$. The vector field is negated in order for the direction to point towards the area of greatest safety rather than least safety. Figure 6 is a vector field formed from combining two cost functions. The arrows point toward increased safety. The vector field described in Figure 6 is $< \frac{1}{x^{0.4}}, \frac{1}{x^{0.6}} >$. The size of each arrow shows relative increase in difficulty of improvement.

The gradient of V is evaluated at a point p (where p is a point in g dimensional space), $-\nabla V(p)$, yields a vector that has the steepest assent towards the area of greatest safety.

The initially known quantities are p_i, $P(p_i)$, $V(p_i)$, 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. We call the new polynomial expression P_k. An example, based on Equation 1, of the substitution is:

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

There will be at most g real roots of P_k. The real root of k that creates the smallest vector $kV(p_i)$ such that $p_i + kV(p_i)$ yields a result with in the domain of $(0 \leq p_f \leq 1)$. 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. For companies that have sufficiently refined metrics such as those found in companies at CMM level 3, the cost functions may be well defined [8]. For other companies, the cost function may be, at best, only an approximation.

4. Application

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

4.1. Example Improvement Scenario

We examine 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 can 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.

4.2. Prerequisites

In order to use the improvement algorithm in this scenario, a few things must be known. First, the probability of each hazard occurring in the initial system is required. This may involve the construction of fault trees and computation of each probability. Second, fault trees need to 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 all components. Table 1 lists the initial failure probabilities of all the variables, the cost functions, and the desired failure probability of the system.

4.3. Verifying the Modified System

In order to verify that the situation is improved through the application of our 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

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 1. Initial Values

and by what amount order to meet the desired failure probability. While the list produced may not be the cheapest possible solution it is not an expensive solution.

The rest of this section examines the improvement process for a single hazard, $h0$. The fault tree in Figure 7 represents all possible ways for the software system to cause $h0$. The cost functions for each of the seven inputs are labeled $c0$ through $c6$. The initial failure probabilities of each component are similarly $p0$ through $p6$. Node 0, for example, would have a cost of c0 and a probability of failure of p0.

The fault tree probability equation, P, is created from Figure 7, and is:

$$\begin{aligned} P(p0,p1,p2,p3,p4,p5,p6) &= \\ 1-(1-(1-(1-p0)(1-(p1)(p2)))) \\ (1-(p3)(1-(1-p4)(1-p5)(1-p6)))) \end{aligned}$$

Once the probability of the hazard is computed it can be compared to the desired probability, G. Since the probability of hazard is greater than G 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 gradient of V at the initial point $(p0,p1,p2,p3,p4,p5,p6)$, provides the direction towards the desired failure probability curve. Once the gradient 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 $k = 0.083321$. 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

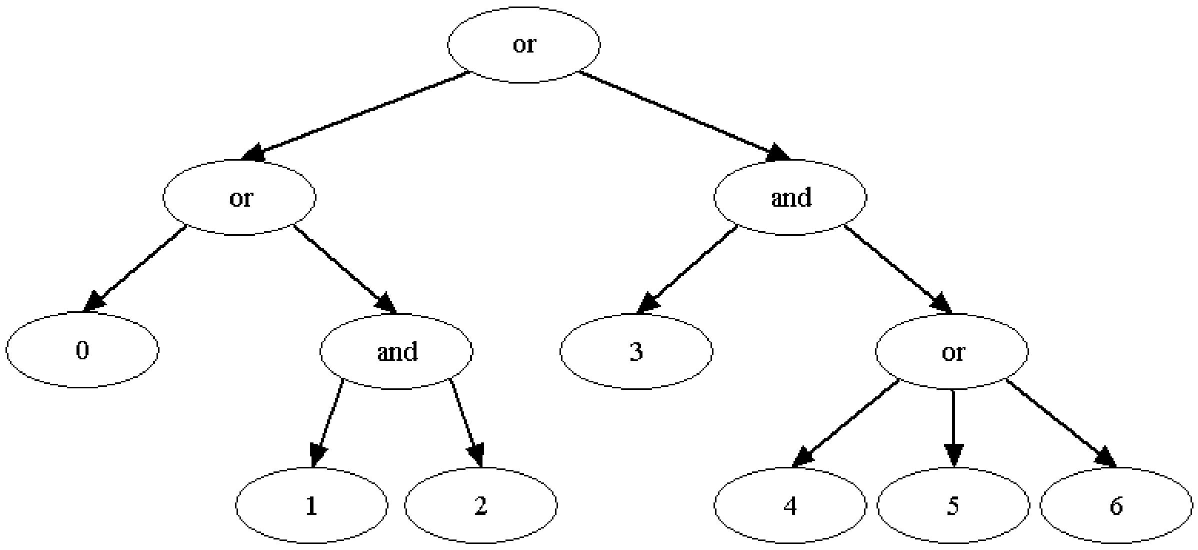

Figure 7. Fault Tree for Hazard h0

variable	value	% improvement	cost
$p0$	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 2. Final Values

required to meet the system requirements is shown in Table 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 100.7 man-hours would be required to improve the failure probability from 0.55 to 0.3.

5. Conclusion

Analysis of software fault trees exposes hardware and software failure events that can lead to unsafe system states, and provides insight on improving safety throughout each phase of a system's development. Our work represents a first step in investigating cost-sensitive methods of verifying safety critical software systems. We presented an algorithm for system failure mitigation based on the reduction of fault trees into polynomial expressions. Cost functions were applied to the nodes of the fault tree to identify nodes where improvement will effectively lead to increased safety of the system represented by the fault tree. We provided an example application of our improvement algorithm, and examined improvement verification of the resulting system modifications.

Future work in this area focuses on the efficiency of the improvement algorithm, which is currently far from optimal. Areas where improvements could be made include the possible use of stream functions to find the cheapest path between the initial and goal fault probabilities and the use of Lagrange multiplies to avoid the scalability issue raised of finding the roots of high degree polynomials. Other future work includes developing the improvement algorithm from a research project into a tool that can be readily used by software developers.

References

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

[2] 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.

[3] 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.

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

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