The safety of industrially-based controllers incorporating software

Thesis

THE SAFETY OF INDUSTRIALLY-BASED CONTROLLERS INCORPORATING SOFTWARE

A thesis submitted for the degree of

Doctor of Philosophy

in the Electronics Discipline,

Faculty of Technology, The Open University

Milton Keynes

by

PHILIP ANTHONY BENNETT

September, 1984

Author's number : HDN 2025

Date of submission : September 1984

Date of award : 24 January 1985

ABSTRACT

This thesis is concerned with the safety of industrial controllers which incorporate software. Software safety is compared with software reliability as a means of discussing the special concerns of safety. Definitions are given for the terms hazard, risk, danger and safe. A relationship between these terms has been attempted and the philosophy of safety is discussed. A formal definition of software safety is given. The factors influencing the development of software are examined. The subjectivity of safety is discussed in the context of safety measurement being a conjoint measurement. Methods of assessing the risk resulting from the use of software are described along with a discussion on the impracticability of using state transition diagrams to isolate catastrophic failure conditions. Categories of danger are discussed and three categories are advanced. The structuring of the software for safety is discussed and the principle of using safety modules and integrity locks is proposed. In discussing the reasons for errors remaining present in the software after testing two methods of measurement are suggested; Plexus and Fallibility Index. The need to declare variables is discussed.

An experiment involving 119 volunteers was conducted to examine the influence of the length of variable names on the correct usage. It was found that variables with a character length of 7 have a better probability of correct interpretation than others.

The methods of assessing safety are discussed and the measurements proposed were applied to a commercially available product in the form of a Software Safety Audit.

It is concluded that some aspects of the safety of controllers incorporating software can be quantified and that further research is needed.

CONTENTS

Acknowledgements

The original idea for this research came whilst working for the British Steel Corporation. It was as a result of discussions with my colleagues that the idea of entering University to conduct the research came about.

The research project has only been made possible through the generosity of many people whom I wish to thank; Professor A.S.Douglas of the London School of Economics and Political Science, Professor I.C.Pyle of the University of York and the manufacturer of the medical product assessed as part of the research. Special thanks are due to Professor J.Monk of The Open University for being generous with his time and guiding me through the research and, of course, to my family.

The research was only made possible because of the facilities provided by The Open University.

CHAPTER 1

Introduction

Previous research has attempted to isolate those factors of software production which influence the incidence of software errors. However, previous research was reviewed for this thesis and it was found not to be concerned with the safety of industrial-based systems. Previous research has been mainly concerned with the non-industrial applications of computing though some has relevance to this thesis.

The research for this thesis was concerned with studying the safety of industrial-based controllers which also incorporate software. The study was concerned with the development cycle of the software from the specification and the development environment to the programming language, testing and maintenance. As a part of the research a set of metrics for assessing various features of the software have been developed to give some guidance on the structuring of such systems. The metrics are intended to allow comparisons to be made between different software development procedures.

Since errors in the software can be considered as a risk and the combination of a risk and a hazard implies danger then it is asserted that software errors are dangerous.

In this thesis an attempt has been made at providing guidelines for the production of safe software based on the research.

In Chapter 2 a survey is made of the current state of knowledge of various factors considered to influence software.

Chapter 3 examines the difficulty of assessing safety from the basis of structural elements and develops some methods of quantification based on certain features of the software.

Chapter 4 discusses the reasons for it being impracticable to

remove all errors and indicates where some of the errors arise.

Chapter 5 reports on work undertaken on a real product to assess the safety of the product and makes observations as to where the difficulties lay in conducting such a safety assessment.

Chapter 6 includes the conclusions and recommendations for further work. It is concluded that no ultimate solution was found during the research and there is a considerable amount of work left to done.

1.1 Problem Definition

The reduction in the cost of computers has meant a corresponding increase in the use of computers for industrial-based control engineering applications. In such applications the consequences of an error are reflected in new risks to capital equipment, human life and the natural environment. The new risks are a consequence of implementing the control strategy in software when replacing existing technologies. In particular, microprocessors introduce new kinds of risk.

The application of microprocessors in industrial-based control systems makes it necessary to be able to assess application programs according to some specified safety standards, though no method of measuring safety exists and a safety standard has not been formulated.

In Great Britain there are legal considerations when applying industrial controllers to hazard-related processes; there is a contractual obligation of the User to inform the Supplier of safety requirements and, conversely, the Supplier has an obligation to inform the User of any safety related issues that have been identified in the controller. Within the framework of commercial activity due regard must also be given to the statutory instruments, such as the "Health and Safety at Work etc Act, 1974".

The Health and Safety at Work Act is administered by the Health and Safety Commission through its Health and Safety Executive (HSE) which has six Inspectorates, three of which are directly relevant to this thesis; the Nuclear Installations Inspectorate, the Mines and Quarries Inspectorate and the Factory Inspectorate.

In hazard-related industrial control systems there is still a need to establish a method for assessing the safety of software once it has been developed and before it becomes operational with respect

to the plant that it will control. Some software test procedures check the software for correct operation within a limited range of test data sets. These checks may be unsatisfactory for industrial controllers incorporating software and which may be used in a hazardous application. No standard testing method exists to demonstrate the safety of software in such a situation. Yet a complete industrial control system may comprise software packages from different sources of supply and sometimes developed for a different range of computers to the target computer under test.

Safety of industrial control software should consider the software, the run-time environment that the software is expected to work in and the function of the software. Measurement of the software safety should indicate the extent to which the software can be confidently expected to work safely: both safe and consistent in operation when controlling equipment.

Since software errors are hazardous, the containment of the hazard within acceptable limits is called software safety in this thesis.

The safety of software is an area of research where there is little published evidence of research.

1.1.1 Definitions

Throughout the thesis the terms 'hazard', 'risk', 'dangerous' and 'safe' are used and to avoid confusion over the terms a definition has been placed on each of the terms;

- 'Hazard' describes a condition with the potential to cause harm; to capital equipment, people or the natural environment
- 'Risk' is used to describe the probability of a hazard materialising
- 'Dangerous' is used to describe a situation where the level of risk of a particular hazard is considered to be unacceptable

- 'Safe' is used to describe a situation where the level of risk is judged to be acceptable.

In all cases people need to be present to transform a hazard into a dangerous state.

Additionally, the term 'software' is used to refer to computer programs written to meet a specific industrial control application.

1.2 Safety

'Reliability' and 'safety' are sometimes considered to be synonymous but in this thesis they are held to be related subjects with different goals.

Reliability is often associated with the term 'reliance' to mean the dependence a user places on a system, when reliability is a measure of the success of achieving a desired operation.

Safety is an emotive topic and the assessment of safety is a subjective judgement but 'safe' intuitively suggests some absolute measure that the risk is 'acceptable' or does not exist. The use of the term 'acceptable' must consider costs, benefits and to whom the risk is considered acceptable; the supplier, the procurer or the user. Therefore, acceptable should be used sparingly to express some agreement between the parties exposed to the risk of the costs and benefits.

Since hazard is used to describe a condition with the potential to cause harm then it follows that for a hazard to materialise then the risk needs to approach unity. As the risk increases the threshold of acceptability will be crossed at a cusp point and the state will be considered to be unsafe which implies, if people are present, that the state is dangerous.

Individual thresholds of danger will vary but it is possible to postulate a set of thresholds which categorise danger according to three levels of danger; serious, major and minor. Placing any state into one of these categories suggests that the level of danger can be expressed by such a term as

$$\text{Level of Danger} = P(r) \cdot Hn$$

where P(r) is the probability of the hazard materialising, risk

Hn is some subjectively assessed number associated with the hazard.

P(r) is the sum of all the events in the event space which can cause the hazard and P(r) is made up of a number of sets of events. The relationship of these sets of events to P(r) can be shown by using Venn diagrams. As an example, consider some computer-controlled machinery.

In the simplest case, Figure 1.2.1. a), the set of events associated with the machinery alone being a hazard are given as P1 and the event space is considered to be the universe of events unbounded. Since no people are present then, by definition, there is no danger.

When people are involved then there is a set of events associated only with the environment being the hazard, P2'. The set of states intersecting P1' and P2', given as P12', are those events associated with the machinery and the environment. When the machinery is being operated by an Operator without the aid of a computer then the event space can be considered to be bounded to include only those events associated with the operation of the machinery, Figure 1.2.1. b).

When the control of the machinery includes some form of computer control the boundary event space changes to include a set of events associated with the computer control alone being the hazard, P3''. The intersection of P1'', P2'' and P3'' is given as P123'' and is the set of events associated with the machinery and the environment and the computer control causing the hazard, Figure 1.2.1. c).

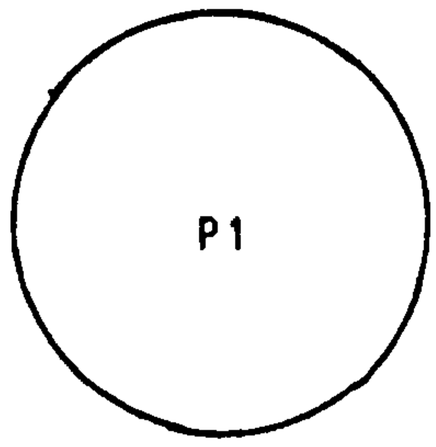
The probability of events in the intersection of P2'' and P3'' causing a hazard is low and the set of events in the intersection of P1'' and P3'' does not involve people so, by definition, cannot be considered as dangerous.

By including computer control on the machinery new risks are introduced. However, the introduction of computer control, P3''',

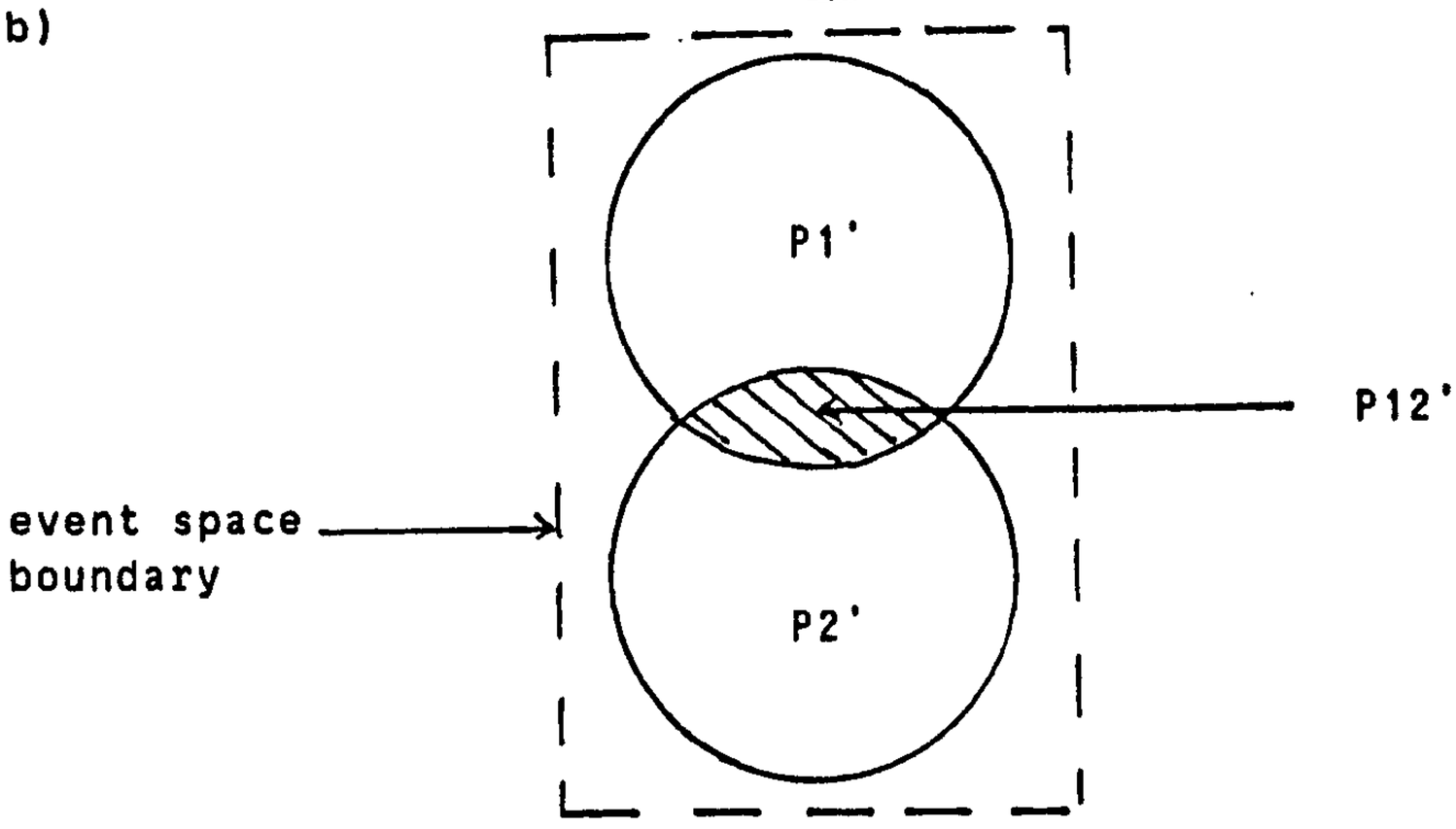
may also reduce the set of events associated with the machinery alone, P1''', and so distort the boundary of the event space, Figure 1.2.1. d).

Figure 1.2.1 Venn Diagrams for Computer-Controlled Machinery

a)

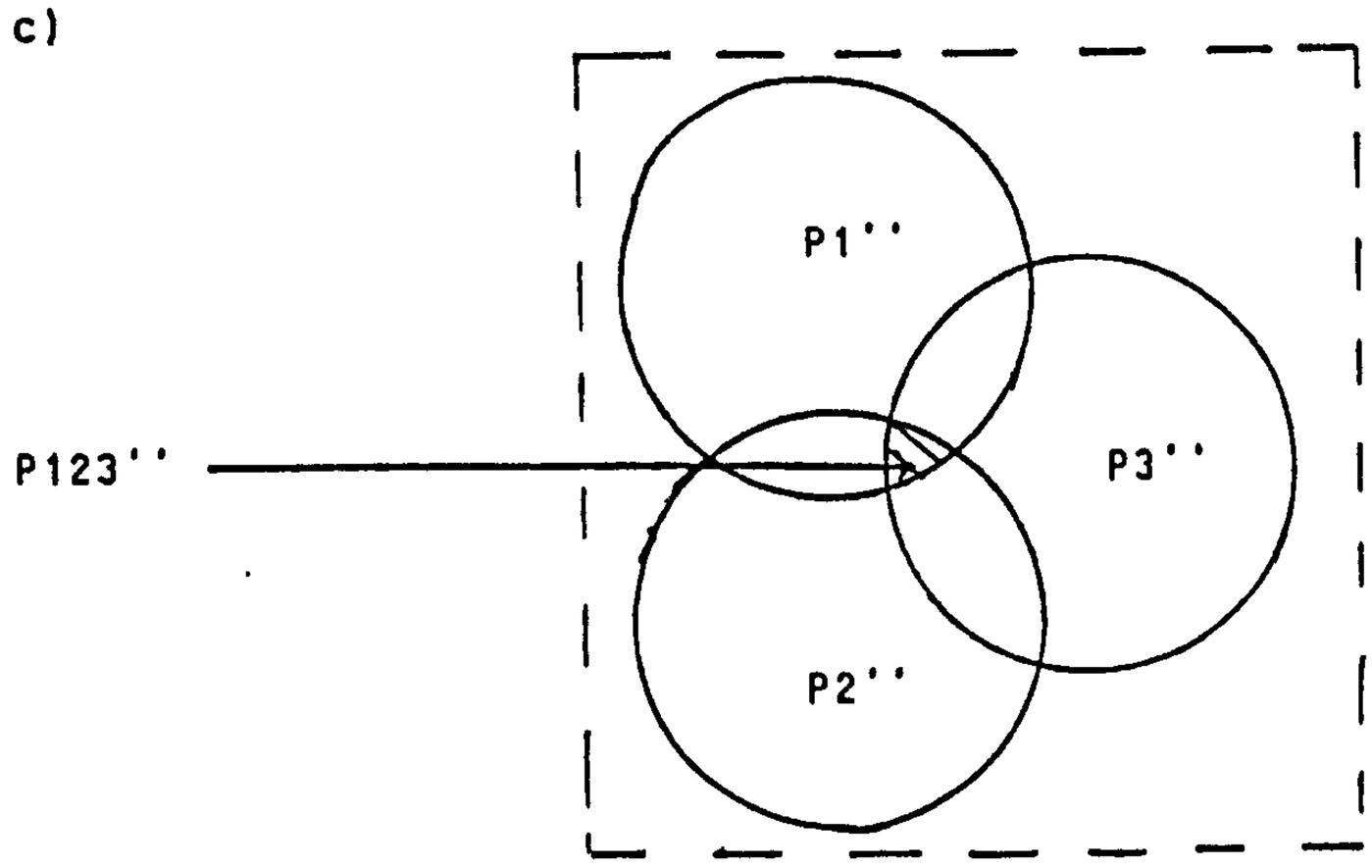


b)



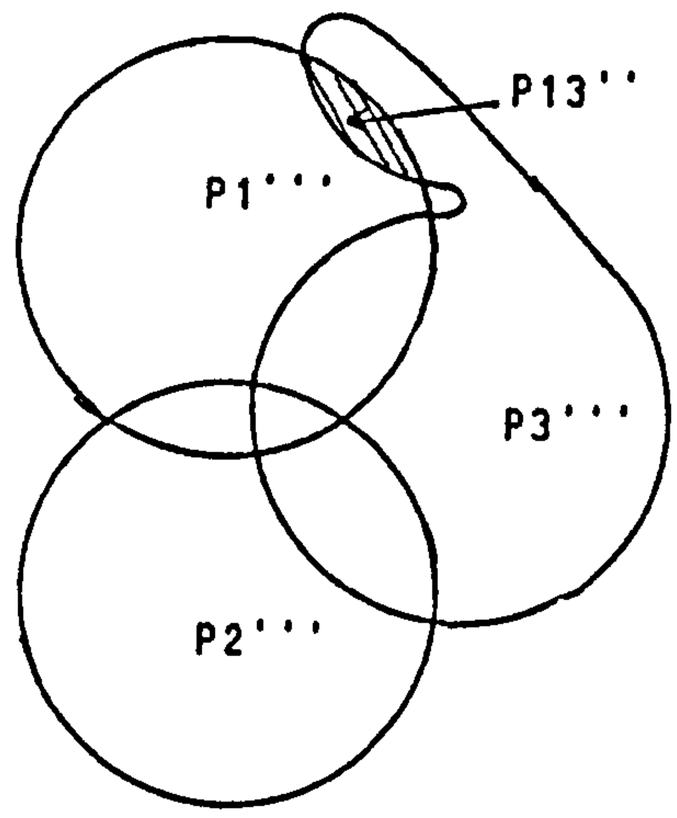
$$P(r) = P1' + P2' + P12'$$

c)



$$P(r) = P1'' + P2'' + P3'' + P123''$$

d)



$$P(r) = P1''' + P2''' + P3''' + P123''' - P13'''$$

1.3 Software Reliability

Having established what is meant by the term safety and that software errors can affect safety then it is necessary to appreciate the concepts of software reliability before discussing what is meant by software safety.

Definitions of software reliability range from an assessment of the correctness of a program with respect to the requirements specification through to a count of the number of programming errors in sample programs.

The assessment of a program's reliability necessitates some knowledge of the programs requirements but the requirements specification can only be regarded as a necessary design document. A requirement specification will state what the software is required to do and may not include statements on safety although it may be the case that safety is to be maintained even when the software is abused. A requirement specification is insufficient to instil confidence in the correct and safe working of a program or to ensure that the program satisfies the requirements.

1.4 Software Safety

The published material on software reliability presented models for indicating the number of software errors found and aimed to predict how many errors still existed. The models also consider that all errors present have the same hazard. The concern of this thesis is the extent to which a risk exists when an error is experienced. The Oxford English Dictionary defines safety as being the noun of the adjective safe: "out of danger: not involving risk: cautious: .....". The concern was with the maintenance of an operational condition which, whilst being reliable, is also free from danger, therefore, it is conjectured that the research for this thesis was on Software Safety.

Though the terms 'reliability' and 'safety' are frequently interchanged they have different interpretations. An item of software may perform in an unintended manner and yet be safe in operation. Equally, software can be unsafe whilst functioning as intended in the specification. Reliability is concerned with all failures. Safety is concerned with the consequences of failures which may result in human or economic cost. Some failures incur more economic-social costs than others and so some errors are considered to be more serious than others.

There has been much research into Software Reliability concerned with the intended function of the software but little specifically on Software Safety.

Current software reliability theory attempts to quantify errors by predicting the number of errors expected to exist. The theories give equal weight to each error predicted to exist. By contrast, in safety assessment it is the intention to qualify errors by weighting them according to the resultant economic-social cost.

Safety and reliability have different goals due to the differing

emphasis. Decision making in a safety-related system involves moral, ethical and economic factors and requires a knowledge of the difference in emphasis between reliability and safety. If this difference is not taken into account then less information will be available on which to base the decision. Therefore, software safety should be dealt with as a related but separate issue from software reliability.

In this thesis software safety is defined as:

"The confidence that a given program will, for a given run-time environment, perform its function in a controlled and reproducible manner within an acceptable evaluation of risk."

The term 'acceptable evaluation of risk' in the definition recognises that safety is a subjective judgement of which software is safe and which is unsafe. The subjectiveness in evaluating risk is a value judgement on the damage that could arise in possible situations and was reflected in the Report of the Court of Inquiry into the Flixborough Disaster [1], Para. 197.

Using propositional logic it is possible to formally state the definition of software safety such that there are three conditions to be satisfied;

1. When the current state, si, is contained in a set of safe states S, there is a function F that will transform the current state to the next state, sj, which is also contained in the set of safe states,

$$\forall s_i \in S, (F(s_i) = s_j \supset s_j \quad S)$$

2. When the current state is contained in a set of unsafe states, U, there is a function that will transform the current state to the next state, which is contained in the set of safe states,

$$\forall s_i \in U, (F(s_i) = s_j \supset s_j \quad S)$$

3. When the current state is contained in a set of unsafe states and there does not exist a set of safe states for the current state to be transformed to, then there is a function that will transform the current state to the next state with the lowest risk

$$\forall s_i \in U, \text{ if } \nexists\, s_j \in S \text{ such that } F(s_i) = s_j \text{ then}$$

$$F(s_i) = s_k \supset Risk_{(sk)} < Risk_{(sj)}$$

where S is a set of states judged to be 'safe'

U is a set of states judged to be 'unsafe',

$\forall$ is the universal quantifier

$\nexists$ is the existential operator

In asserting that $Risk_{(sk)} < Risk_{(sj)}$ consideration needs to be given to the time taken for the system to achieve state sk.

There at least two strategies that can be adopted when considering the consequences of time. If $Risk_{(sk)}$ is considered to be lower than $Risk_{(sl)}$, yet more time is required for the system to achieve state sk than state sl, a judgement can be made whether safety is best served by achieving state sk with a low risk in a longer time than state sl. State sl has a higher risk than state sk but a lower risk than state sj and can be achieved in a short time.

Therefore, condition 3 can be qualified;

3a. $$\forall s_i \in U, \text{ if } \nexists\, s_j \in S \text{ such that } F(s_i) = s_j \text{ then}$$

$$F(s_i) = s_k \supset Risk_{(sk)} < Risk_{(sj)},$$

$$\text{iff } Risk_{(sk)} \supset T_{(sk)} < T_{(sl)} \wedge (Risk_{(sk)} < Risk_{(sj)} > Risk_{(sl)})$$

where T is the time required to achieve a particular state from the current state, si. This strategy is appropriate to those instances when it is only possible to one state ahead.

If it is possible to look ahead more than one state an alternative strategy might be to achieve a state with a higher risk

for a short time in the knowledge that a state with a lower risk will be achieved ultimately. In this strategy safety could be expressed as the integral of the level of danger against time

$$\text{Safety} = \int P(r) \cdot Hn \, dt$$

The value of Hn is subjectively judged and could influence which strategy to adopt due to the level of confidence in the judgement.

The definition of safety can be formally stated but what is considered to be a set of 'safe' states or 'unsafe' states depends on a subjective judgement based on knowledge, experience, emotion and legislation of what is acceptable at the time.

1.5 References

[1] Dept. of Employment, The Flixborough Disaster: Report of the Court of Inquiry, H.M.S.O., 1975

CHAPTER 2

Factors Affecting Software Safety

In this chapter an argument is advanced that the assessment of safety is subjective and will remain so until some method of measurement can be found which is not a conjoint measurement. Factors found during a literature survey, which have been considered by other researchers to influence software, are also examined in the context of each stage of the software development process before implementation and following implementation.

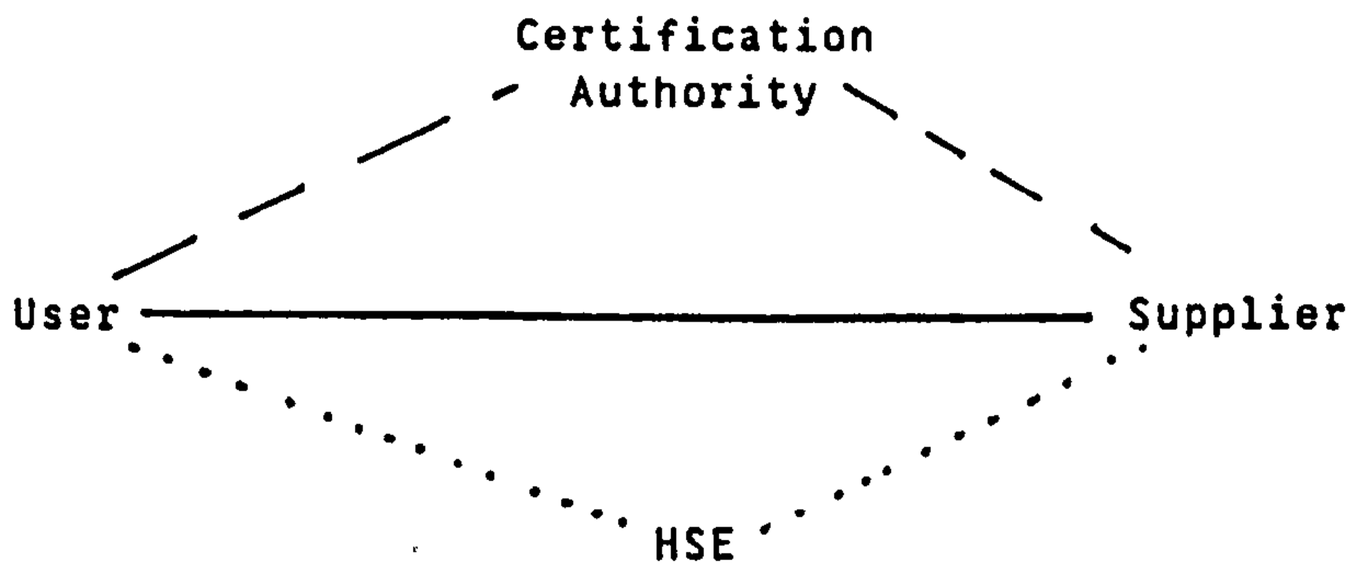
There has been a substantial amount of material published attempting to establish those factors having an influence on the production of software. Since there has been a substantial amount of relevant material published only selected material has been identified and referenced.

2.1 The Subjectivity of Safety

Software can be adversely affected by certain factors and it follows that the safety of an industrial process resulting from the use of the software can also be affected by these same factors. The factors considered in this chapter are those asserted by the respective researchers to have an undesirable influence on software. These factors are considered with the emphasis being on the safety of the software. The term 'safety of the software' is used to mean the safety of the system as a result of using software rather than to mean an assurance that the software is itself 'safe' from errors.

To confidentally install a controller incorporating software it is necessary for some checks to be carried out leading to certification of the software for use in safety-related systems. In a safety-conscious industry it will be normal practice for these checks to be undertaken by a third party, separate from the User or Supplier, who is also aware of the requirements imposed through legislation or common by 'best world practice'. The relationships between the Supplier, the User, the Certification Authority and the Health and Safety Executive are represented diagramatically in Figure 2.1.1 with the solidity of the line reflecting the strength of the relationship.

Figure 2.1.1 Relationship between the parties in the certification process.



The form that the certification takes will depend on the experience and knowledge of the User, of the Supplier, of the

Certification Authority and on the current legal requirements. The form of the certification will also be influenced by the state of knowledge of the certification methods.

The certification of the design of any equipment needs to be comprehensive. The difficulty of undertaking a comprehensive design study was raised in the Court of Inquiry into the Flixborough Disaster, [14] paras. 191-193, when it discussed in general terms the probabilities of the "8-inch hypothesis" and the "20-inch hypothesis". The Court decided to refer to a special committee, paras. 217-219, the concern of a major disaster resulting from the design of process plant and equipment.

The Danish organisation Elektronik Centralen, [17], have issued a draft directive on the testing of software used in control and surveillance systems. The level of safety is determined by assessing observable actions called 'qualities' which are considered to influence the safety of software. It is difficult to use the assessment as a comparison between two dissimilar systems since there are no quantitative measurements. Since these qualities are subjective observations made by the assessor of the software at the end of the whole development they have not been used as the framework for this Chapter.

Software production factors such as the choice of programming language and data structures, programming methodology, quality assurance and project standards are sometimes asserted to influence software production.

One researcher, Rault [59], surveyed the published work on the production of high quality software and concluded that there was a need for research into what he called "quality control" and listed some of the factors to be considered; 'complexity, comprehensibility, usability, modularity, reusability, adaptability, testability,

sortability, linkability, robustness with respect to user mis-operations, and so forth,' and suggested some methods of measuring these factors. These factors are vague and concerned with assessing the end-product, the software, against a set of qualities which are not observable.

At each stage of the development of software there is a choice to be made between competing methods and techniques and in each case there are some good and some bad ones. The choice of method or technique to use will influence the safety of the software but, as the literature survey will show, there has been competing assertions.

Following the literature survey, the factors in this thesis have been grouped into four sections covering the stages of the development process before implementation and after implementation. The sections are:

Specification and Design

Programming Language and Programming Structure

Support Environment and Testing Strategy

Operational and Psychological Factors

Table 1 contains an analysis of the frequency of occurrence of these factors (and their subsets) by application area. The totals are for each set of factors found in the surveyed works.

The sections of the survey assume that undesirable influences can introduce unsafety at each stage of the software development process and that each factor can affect the outcome. For safety to be assessed according to any one factor, the factor must be observable and quantifiable and not subjective which implies that it is possible to attach a numerical value to safety and that safety can be absolute. It is the desire to quantify safety that has led many people to relate safety and reliability, when reliability is a quantitative measure and a method has not yet been found of

quantifying safety.

Calculations of software reliability related to the coding of a program do not constitute a definition of software reliability but can be categorised as a software metric. The assumption is that a well structured program will be more reliable than a badly structured one due to the clarity of expression of the logic in the program. The assumption takes account of the possibility that the logic could be incomplete and also assumes that good structuring is always possible.

One definition based on error rates, from Rault and Bouissou [60] states that software reliability is;

"the probability that a program works without error during a given time span on the machine for which it has been intended and under specified conditions".

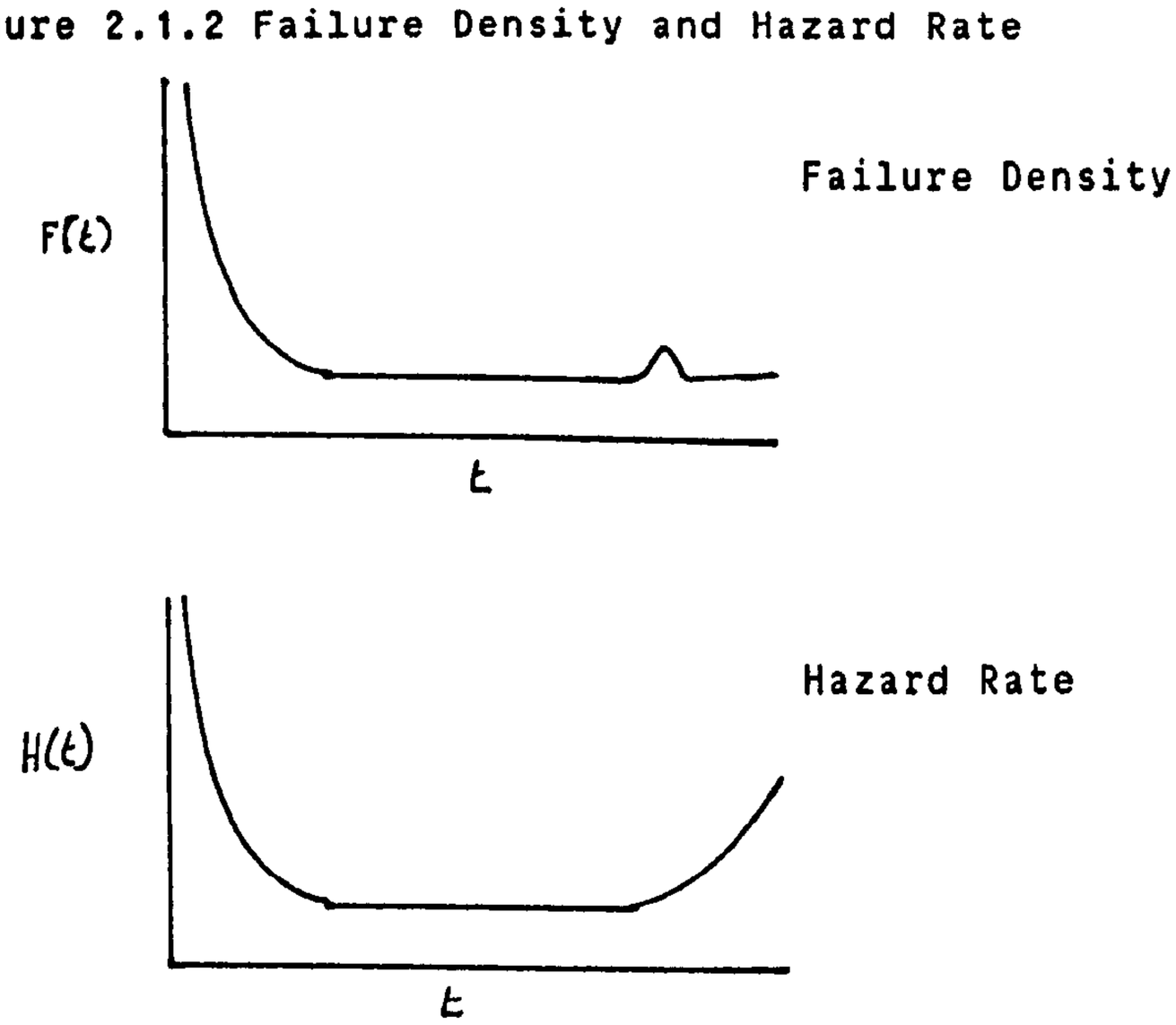
Here the concern is with a statistical probability of failure calculated from a count of the number of errors detected and corrected over a specified period of time. A problem arises in the use of elapsed time as a parameter since failure to function reliably is dependent on the occurrence of a specific condition. The history of the rate of reduction of programming errors will influence over-confidence in the software if it shows a rapid reduction. Conversely, if the number of outstanding errors is reduced at a slow rate, the confidence in that software would be accordingly low.

If it is assumed that the failure density reduces exponentially then as the failure density reduces so will the hazard rate accordingly and demonstrate a steady state operational life. The effects of wear and aging of mechanical equipment causes random failures to be seen in the failure density giving a corresponding change in the hazard rate. The hazard rate will then be reminiscent of a bath-tub, which is where the term "bath-tub curve" comes from.

In software reliability it is assumed that the detection and correction of software errors reflects an exponential function and that a point can be reached where an 'acceptable' number of errors are considered to exist and after which time continued stable operation can be expected since software is not affected by aging or wear-out.

Many industrial control systems will not be changed from their initial operating status during the life of the system. Many will be subjected to change after some period of stability to reflect the revised operational requirements. The changes may cause some new errors to be introduced causing a transient increase in the failure density and a consequent rise in the hazard rate. Modelling of the failure rate of software using the bath-tub curve is useful if changes to the system are anticipated.

Figure 2.1.2 Failure Density and Hazard Rate



When the failed unit is repaired and returned to service, a measure of the reliability of a unit is the term Mean Time Between Failure, and when the failed unit is not repaired the term Mean Time To Failure is used. It has become accepted practice to use MTTF when measuring software reliability since any correction applied to

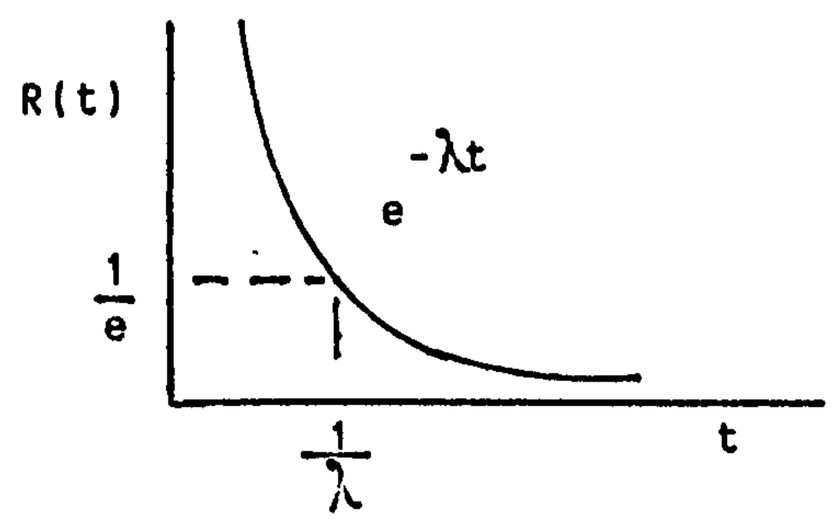
the program will change its characteristics and can therefore be regarded as a new instance of the program rather than the erroneous one being returned to service after repair.

In using MTTF it is assumed that an exponential reliability function with a constant hazard rate, λ, applies. However, MTTF equates to the reciprocal of λ and the number of failures experienced will be at least half of all failures, Figure 2.1.3.

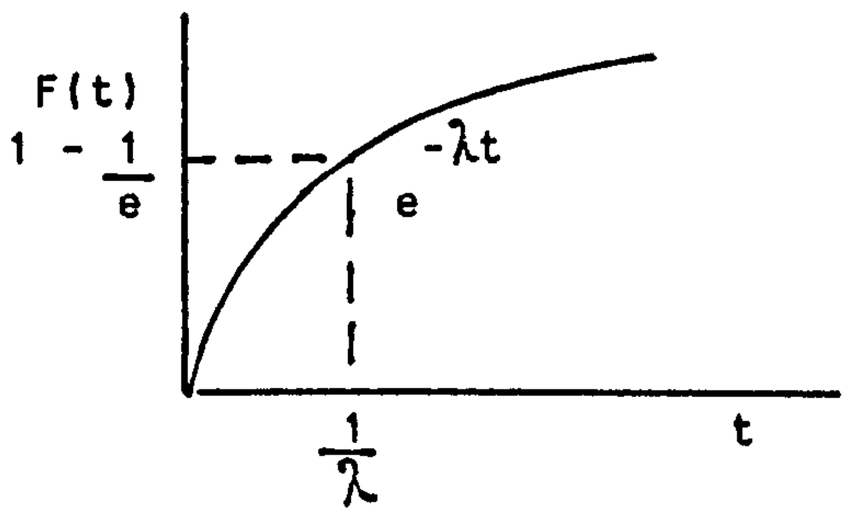
Reliability models can be grouped into two main types: Deterministic and Bayesian.

Figure 2.1.3 Mean Time To Failure

a) in terms of reliability



b) in terms of failure rate



In the Deterministic group of models the Jelinski-Moranda and Musa models dominate the published material, [49]. The assumption in these models is that the times between detection of errors, T, are independent random variables, V, and that time, t, is conditionally exponential, so

$$pdf(t \mid V) = \lambda e^{-\lambda t}$$

and

$$\lambda = (N+1)\Phi$$

where N = initial number of faults

Φ = contribution to failure rate from each fault

The execution time model of Musa is becoming more widely evident in published literature and is based on the Jelinski-Moranda Model. In the Musa model the expected number of errors, n, is given by

$$n = No\ [\ 1 - \exp(\ -Ct/NoTo\ )]$$

where No is the inherent number of errors

To is the MTTF at start of testing

C is the 'testing compression factor' and a ratio of equivalent operating time in the target environment to the actual operating time in the test environment.

The present MTTF is given by

$$T = To \exp\left(\frac{Ct}{NoTo}\right)$$

giving

$$R(t) = Pr\{\text{ no failure in } (t,t+1)\} = \exp(-t/T)$$

To improve the MTTF from T to T'

$$\Delta n = NoTo\ \left(\frac{1}{T} - \frac{1}{T'}\right)$$

and the execution time to achieve this change is

$$\Delta t = \frac{MoTo}{C}\ Ln\ \left(\frac{T'}{T}\right)$$

Littlewood [41] discounts the use of Mean Time To Failure (MTTF) and Mean Time Between Failure (MTBF) in the context of software as elapsed time can only be used when a regular pattern of use can be demonstrated.

The Littlewood-Verrall model, [42], dominates the Bayesian group of models in the published material and also assumes exponential reliability growth;

$$pdf(t \mid \lambda) = \lambda\, e^{-\lambda t}$$

where / is the program failure rate with a gamma distribution: that the failures do not occur at a constant rate but depends upon program usage.

The model also assumes that different program errors have different probabilities of failure. The failure rate is given as

$$PDF(\lambda) = \frac{[\Psi(i)]^{a}\,\lambda^{a-1}\,e^{-\Psi(i)\lambda}}{\Gamma(a)}$$

where a = a-th failure

λ = failure rate

$\Gamma(a)$ = gamma function

Ψ = linear function

giving

$$F(t) = 1 - \left[(\Psi(i)) / (t + \Psi(i)) \right]^{a}$$

$$R(t) = \left[(\Psi(i)) / (t + \Psi(i)) \right]^{a}$$

and

$$MTTF = \Psi(i) / a-1$$

The models discussed above are concerned with the operational performance of the software, the amount of testing needed and the software error-rate. There is no indication of the seriousness of the errors estimated to exist or which errors would create a catastrophic operational malfunction. The rate of detection and correction of errors does not indicate the risk associated with the usage of the software.

In chemical plant design studies it is common practice for the design to be subjected to a range of techniques known by the generic term 'Risk Analysis', [3], in order to determine the risk associated with the design. The approach is to examine product flow routes and to ask the question 'would it be nasty if ...'. Probabilities of

component failure in each route are assessed and submitted to probabilistic analysis to establish the risk for the total plant.

An analogy between the risks in a chemical plant and the risks in software is to be seen by viewing the data flow of the software in a similar manner to the material flow of an chemical plant. An assumption is made that the risk from the software is independent of the risk from the supporting hardware. Such an assumption is similar to that applied to other industrial plant when it is assumed that the risk from each nut and bolt or individual component is acceptable. In both instances, the resulting risk analysis is subject to external events on each component.

Once a risk analysis has been done then it is a simple task to repeat the original risk analysis following modifications. The investigations reported by Taylor, [66], indicate that it may be possible to apply some risk analysis techniques to software following detailed examination of the functional specification.

In the Report of the Court of Inquiry into the Flixborough Disaster, [14], the following comment is made in para. 196;

"No plant can be made absolutely safe any more than a car, aeroplane, or home can be made absolutely safe. It is important that this is recognised for if it is not, plant, which complies with whatever may be the requirements of the day tends to be regarded as absolutely safe and the measure of alertness to risk is thereby reduced".

Both 'risk' and 'hazard' can be quantified and the combination of risk and hazard is called 'danger'. Since 'safe' has been defined as being a situation where the level of risk is judged to be acceptable then it is desirable that safety should also be expressed quantitatively. The word 'safety' is often associated in peoples minds with the word 'dangerous' which describes a situation which,

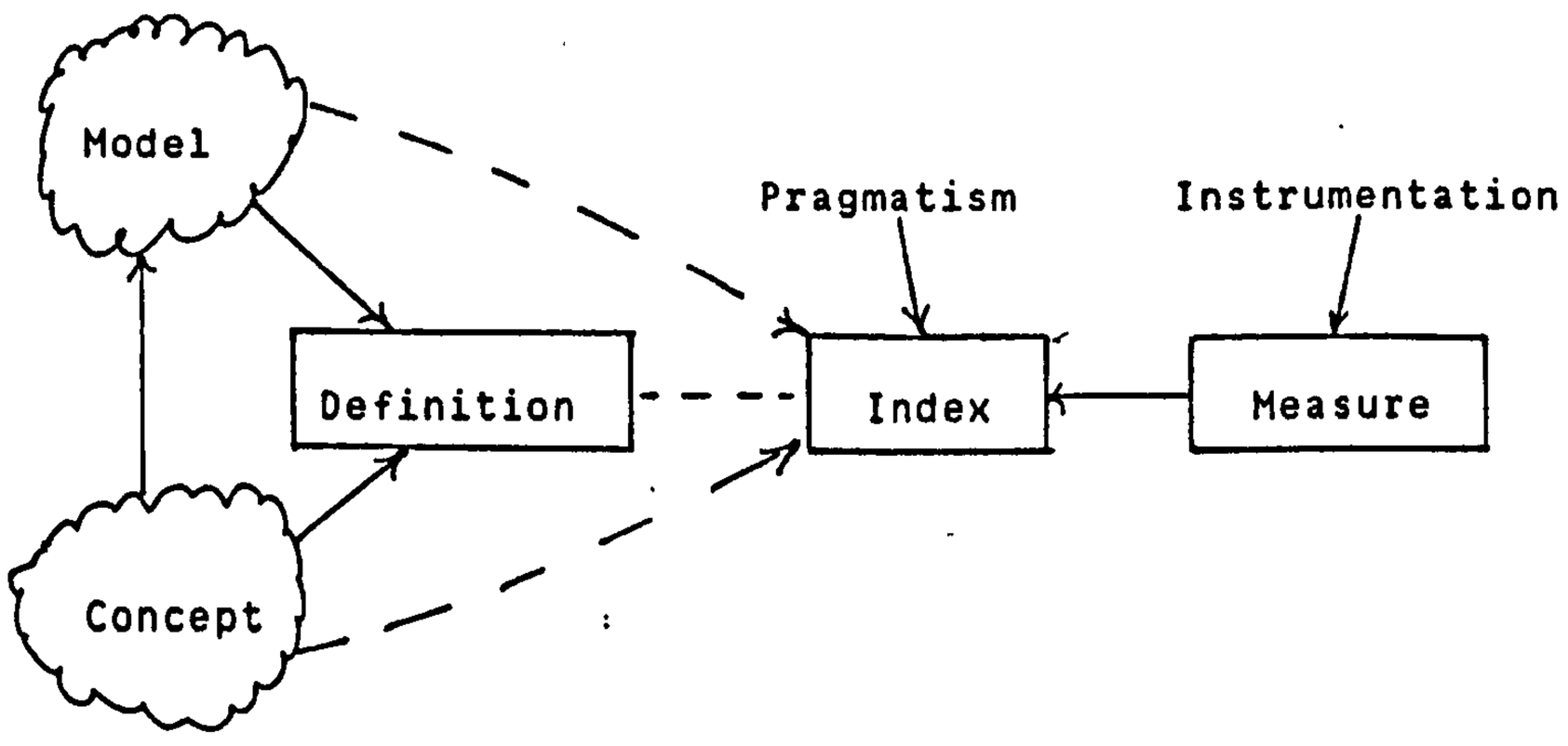
though safe to one observer, may be considered to be unsafe to another. Dangerous has been defined in this thesis to mean a situation where the level of risk of a hazard materialising is unacceptable and will have an undesirable consequence on either capital equipment or people.

Safety refers to the subjective judgement of potential hazards within the safety criteria and is based on personal experience supported by limited measurements of measurable parameters. This is currently not possible.

To understand why safety cannot be expressed in terms of a quantitative measurement the epistemological and logical foundations of measurements need to be examined. But before examining the principles of measurements it is necessary to have a picture of the problem of fitting any scale of measurement to safety.

On the one side of the picture there is a notion of safety comprising a conception of what is 'safe' and what is 'unsafe', a model of how safety relates to the world and a definition based on both of these. On the other side of the picture there is some method of instrumentation providing a measure which, through pragmatism, is ordered in to some index. The ordering of the index is not influenced by pragmatism alone but is also subject to influence by the model and the concept.

Figure 2.1.4 The Measurement of Safety.



To be able to have some measurement of safety according to the definition it is necessary to have a mapping function between the definition and the index. Finkelstein, [19], formally defines measurement as being a set of mathematical entities Q and a set numbers N with a mapping function M between the sets such that M maps the mathematical entities Q onto the set of numbers, M:Q -> N. A scale of measurement S is given as the triplet S = {Q,N,M}.

If the class of entities Q is considered to be the definition of safety shown in Figure 2.1.4 and the set of numbers N as being the index, then in order for safety to be expressed in terms of a measurement a mapping function M which maps the definition D into the index I, M:D -> I, is needed. Such a mapping function may be considered as a conjoint measurement which, according to Finkelstein, relates to a set of measurements having the capacity to assign a measure to the object and order the measure in a set of measures. Conjoint measurements, then, are some form of ordering according to empirical observations (subjective judgement) not rankings which Finkelstein describes as a comparison against defined standards. Since defined standards do not exist for the safety of software it is asserted that the safety assessment of software is a subjective assessment.

Safety, then, is some subjective judgement about the risk of a hazard materialising and that the risk is acceptable in the social climate prevailing at the time of the judgement. The Flixborough Inquiry, [14] para. 197, comments on the acceptability of risks;

"When Mr Marshall refers to risks exceeding a specific value we understand him to refer to risks which exceed what at a given time is regarded as socially tolerable, for what is or is not acceptable depends in the end upon current social tolerance and what is regarded as tolerable at one time may well be regarded as intolerable at

another. Nowhere perhaps is this more apparent than in the field of road transport where the construction and use regulations have, over the years, become even more stringent".

The acceptability of risk is dependant on personal experience and an appreciation of the probability issues involved. For example Knox, [37], has suggested that a probability of 100,000:1 is considered to be safe whilst Starr and colleagues, [64], found that risks between 10,000,000:1 and 10,000:1 are considered by the general public as being acceptable since "their likelihood are no more than being struck by lightning". Caution must be expressed when thinking of acceptable risks in terms of ratios otherwise the wrong inferrence may be drawn.

If it is subjectively assessed that activity X is "safer" than activity Y then the judgement may accurately reflect a comparison of some characteristic of the entities. However, the utility of the safeness of entity Y may be greater than that for entity X, since the social-economic consequences of entity Y being unsafe may be greater than for entity X. Therefore an assessment of safety must not be considered in isolation from the economic-social costs of being unsafe.

As an example of the subjective nature of safety it can be said that the accumulation of explosive materials above that licensed by the Local Authority is not safe. Yet at the Inquiry into the Flixborough Disaster it emerged that the site had a licence under the Petroleum (Consolidation) Act 1928 to hold 8,500 gallons of explosive material yet the management, including the Safety Officer, had allowed 367,850 gallons of explosive material to be stock-piled. The storage of explosive material at a level of 43 times that licensed was not considered by the Court of Inquiry as unsafe. To the contrary, the Report commends the management in three paragraphs

(paras 201, 202 and 206) for being "safety conscious".

2.2 Specification and Design

The earliest stage of the development of software is the specification and design. A specification should be an unambiguous statement of the intended properties (characteristics) of a program. The unambiguous property of the specification applies whether it is a formal requirements specification or an informal functional specification and is a guide to the designer about the requirements of the system. The designer undertakes the design according to his understanding of the specification.

The produced design is the designer's interpretation of the specification which he will consider to have understood in detail and yet may have produced a design which does not satisfy the specification. Such a design may be unsafe in operation due to the designer not having appreciated the safety requirements contained in the specification.

Basili and Perricone, [4], examined two large software systems. One of the systems was for satellite planning studies and comprised approximately 90,000 lines of Fortran. The second system, a 'ground-support' system, was programmed by the same organisation as the first but the length of the code and the programming language used were not reported. After analysis Basili and Perricone reported that on the 'ground-support' system only 8% of errors were attributed to specification errors yet on the satellite system 48% of all errors were "... attributed to incorrect or misinterpreted functional specifications or requirements".

2.2.1 Formal Methods

Formal Methods of specification have been developed using Finite State Machines, Directed Graph, Control Flow Graphs, Modal Maths and Denotational Semantics. The published works have been largely concerned with the description of formal methods rather than specific

application examples.

Berg et al, [5], suggest that formal methods are an aid to software production by stating precisely the requirements and objectives that the program is to satisfy.

When a formal specification is used the designer may consider there to be minimal specification ambiguities. Acts of faith by the designer in the infallibility of the specification may lead to a lower awareness by the Programmer of the requirements and a consequential increase in human error during the software production.

2.2.2 Functional Specifications

A functional specification is a statement to the customer about the way the software is expected to react on a given input sequence. However, the action to be taken following an unanticipated and unspecified combination of inputs or events is not specified. A consequence of failing to make a statement about actions following unexpected events may mean that the designer fails to establish the unsafe conditions.

Functional specifications have become common in many industrial installations over the years and are sufficiently detailed for many purposes according to Kopetz, [38]. The format of a Functional Specification varies according to the project standards in use within the organisation and no standard approach to its compilation exists. Non-standardisation of a specification format may create a situation where a specification is misinterpreted by a designer who is familiar with one format of specification and is being requested to prepare a design against an unfamiliar specification format. Consequently the designer may overlook some of the safety features of the design.

Pyle, [57], suggests that the requesting authority for the design may be a Plant Manager who has a deep understanding of his process requirements but may not have a similar grasp of formal

specification methods. In such a situation functional specifications aid the requesting authority to have confidence that the overall safety of his plant will be maintained and aids him to appreciate complex concepts, particularly about the scope of the proposed solution.

When Functional Specifications are used decisions on algorithms and their implementation are deliberately not taken at an early stage in the design process so that unnecessary constraints are not imposed on the designer. Henry, [28], warns of the dangers of "overspecification" hindering the conception process through limiting the set of possible solutions.

With functional specifications a confusion may arise about the precise nature of the function to be performed and lead to the omission of some aspect of the design aimed at ensuring safety. Some functional specifications include a separate section on the safety aspects of the system.

2.2.3 Specification Languages

Ramamoorthy and Ho, [58], state that there is an urgent need for specification languages in which system requirements can be unambiguously stated and validated.

The belief that specification languages can improve the consistency of the software design has led to the use of program-like languages to specify the design. A program written in a high level programming language describes the means of achieving a given transitional state without explicitly expressing the effect of the transition. For a specification to be meaningful to the requesting authority the effect of the transition needs to be expressed not the means. Pyle, [57], rejects the use of a specification language to formally specify a design since it is usual for such a specification to be useful solely to the designer and not understandable to the

requesting authority.

Software specified by a specification language can be submitted for verification using formal methods but the use of a specification language does not implicitly ensure safety. Berg et al, [5], report that no major systems have been specified or verified using a specification language.

2.2.4 Structured Design

Structured design is a software interpretation of the specification.

Structured design is the arrangement of functional modules into a conceptual hierarchy of modules comprising the system. To construct the hierarchy a technique known as stepwise refinement, Wirth [69], can be used to develop a description of the system and its data structures. At each step in the refinement process a consistency check is made to ensure adherence by the design to the specification and that each development stage reflects the specification of the previous stage plus revisions.

In order to construct the hierarchy two approaches are common; bottom-up design and top-down design.

Bottom-up design is a method used by many designers when designing individual modules and arranging the interconnection of the modules until they meet the requirement. Top-down design examines the requirement and divides it into designs which are definable portions of the total requirement. The designs can be further divided until a number of modules have been identified.

Step-wise refinement can induce errors when following either a purely top-down or purely bottom-up design because of the oversight of common functions. There is a secondary effect, that of creating a poor design because of the design being fragmented. These effects can be assessed qualitatively but not quantitatively.

2.3 Programming Language and Programming Structure

2.3.1 Programming Language

Whilst the design method influences the system structure and interactions, the main production tool in software development is the programming language. The choice of programming language influences the amount that errors can be introduced into the resulting code.

Young, [71], gives a comprehensive review of languages for industrial control systems and makes recommendations on those languages he considers to be suited for the purpose. Young sets out six basic criteria for the design of a real-time language; "security, readability, flexibility, simplicity, portability and efficiency".

Security of a language is some measure of the extent to which errors in the program will be detected by either the compiler or the run-time support system. Readability concerns the choice of variable names and legibility such that a conceptual understanding of the software can be gained by reading the program listing without recourse to further documentation. Flexibility of a language is the richness of choice available to a programmer using the language.

Simplicity reflects the time and cost required to train a programmer in the language and also the reduction of programming errors caused by misinterpretation of the language. Portability is the ease with which a program written in a particular language is able to be moved from one computer to another computer without being dependent on the supporting hardware of either computer. Efficiency is some measure of the computational throughput compared with the constraints imposed by the control system and some measure of the predictable overheads, such as data manipulation. Young suggests that of the six criteria security and readability are vital in safety-related systems.

It was found during the survey of published literature that some

languages are considered to be more secure than others, due to their syntax. Horning, [31], gives an insight into some of the problems of using particular languages.

For high-level programming languages there are fewer errors for a given function then would be the case with a low-level programming language. Therefore, the programming language has an effect on software safety.

Comments have been made by Young, [71], and Horning, [31], about the type of language to be used for different tasks. Rzevski, [61], reported on experiments which he asserts show that expert FORTRAN programmers write equally reliable and safe programs as expert PASCAL programmers. Rzevski also reported that he has found it easier for novices to learn to write reliable programs in PASCAL than in FORTRAN and attributed the findings to the structuring of the language.

Gannon, [21], suggested that a programming language for real-time use needs to be secure and cites the implementation of data typing as an example of language security. A data type specifies the set of operations that can be applied to objects of that type and the range of values an object of that type may have. The method that a programmer adopts to ensure the security of data is a safety concern since data corruption can lead to incorrect functioning of programs.

Another aspect of programming languages which Horning, [31], considers to be unsafe regardless of the task, is the control structure. The control structure is influenced by the amount of code indentation and in an experiment conducted to examine the effects of the indentation of code Miara et al, [47], found that indentation significantly influences the comprehension of the program by programmers and concluded that the experiment coincided with the earlier work of Kerninghan and Plauger, [36].

Reference was found to faulty instructions not being detected during testing and resulting in unsafe operation. One such report concerned the early termination of a French meteorological experiment caused by a controlling satellite issuing the 'abort' command instead of the 'read' command and destroying 72 of the 114 weather balloons, Anderson and Lee [2], and Myers, [50].

In some languages, notably BASIC and FORTRAN, the declaration of variables is not required and new variables can be implicitly declared within the body of the program. The alternative strategy is to require that all variables be defined in a declaration block at the beginning of a program. Languages which do not require declarations may be considered as unsafe when used for industrial control since the declaration of variables within the body of the program promotes ambiguity.

An example of the risks of not requiring declarations of variables is the reported loss of a space mission to Venus, Mariner I, Myers [50], which was reportedly due to an error in a program written in FORTRAN of the type:

```
DO 3 I = 1.3
```

Because Fortran is a context-dependent language, the statement was treated as an assignment of the value 1.3 to a variable and allocated DO3I to that variable rather than correctly executing a DO loop.

Reported losses of equipment through software errors have caused expensive losses of equipment. There have been no published reports of incidents endangering human life. Reports such as these demonstrate the risk of not declaring variables.

The ability to handle non-standard input-output devices such as Analogue-Digital Convertors and Digital-Analogue Convertors, is important to the control strategy and needs to be considered when

selecting a particular language for control. In industrial control systems programs need to be able to control low-level devices and problems have been recorded with low-level languages, Pyle [56]. It is considered by Pyle to be preferrable to use a high-level language for such occasions yet there needs to be a capability of programming low-level device hardware, without needing to resort to machine code. The language 'C' has many low-level features as part of the language.

In industrial control systems it is common for an external stimulus to execute more than one program simultaneously whilst maintaining synchronism. Such a requirement is called concurrency and with the development of multi-tasking languages like Ada, [11], it will be possible to operate concurrent tasks at the program level rather than through the Operating System. With multi-tasking languages special problems arise in validating the software for safety but no evidence of these problems has yet been published.

The mechanism for handling exception conditions in high-level languages in a safe way is important and with the development of the language Ada exception handling is becoming a feature incorporated within the language rather than being a feature of the operating system.

2.3.2 Program Structure

According to Ramamoorthy and Ho, [58], safety of the software can be improved by using a high-level language and structured programming techniques.

Understanding the problem that the program is attempting to satisfy is important in reducing the extent of errors and may also ease the task of testing. If each module specification states the internal and external program interfaces the possibility of a mismatch between, and with, other modules is reduced.

Modularisation of the system allows the programmer to become

familiar with the module to be worked on and to comprehend the detail. But if the program is badly modularised it is probable that the programmer will not be able to appraise the objective of the program. However, with modularisation there is the need to maintain standard inter-module interfaces and dependencies.

Unless the programmer has an intimate knowledge of the program he will not be adequately equipped to test it. If the program is not tested to the best of the programmer's ability then only a limited amount of reliance can be placed on the program.

Structured programming was defined by Dijkstra, [16], as being a set of rules for programming to meet just such a requirement. Since Dijkstra's initial paper on structured programming there have been many definitions including the definition by Myers, [50] p.130, where structured programming is defined as "the attitude of writing code with the intent of communicating with people instead of machines". The Infotech Report, [34], singles out one definition of structured programming as; "the task of organising one's thoughts in a way that leads, in a reasonable time, to an understandable and correct expression of a computing task".

In structured programming functions are structured into distinct units which may be subsequently interpretted into program blocks, procedures or function calls depending on their purpose within the program, Young, [71]. Statements are arranged in a manner that will reflect the logical execution of the program. An example is the interpretation of the general statement "if x obtains then do y otherwise do z" into the program statement "IF x THEN y ELSE z". Reduction of abstract function statements into a structured program removes the need to use GOTO statements but makes use of the basic control statements; sequence, IF..THEN..ELSE.., WHILE..DO.., REPEAT..UNTIL.. and CASE..OF...

Pyle, [55], suggests that structured programming techniques are not sufficient for industrial control systems without improving on existing techniques. Pyle bases his argument on the significant differences he has observed between control software and conventional sequential programs. The main difference being the need for control software to respond to external stimulii and for the control software to be not only correct but also safe. Pyle's argument is more significant as concurrent software becomes accepted in control systems.

Structuring of the total system, which may consist of many programs, influences safety in a positive way according to Allworth, [1]. If a computer has the facility for interrupts and priority levels then the commonly used structure in industrial control computing is to put the frequently run and time-critical programs, like alarms, on the higher priority levels and the less critical programs, like reports, on the lower priority levels. The interrupt facility can then be retained for activating those programs which must be run without delay from the scheduler of the run-time support system.

The language chosen for the given task, the style of programming used and the availability of programming constructs which reflect the problem structure can result in errors in interpreting a specification of a program.

2.3.3 Programming Methodologies

As part of the extensive range of work being undertaken on the language Ada two comprehensive studies have been made into programming methodologies for embedded computer systems.

In the first study, Pickett et al, [10], a range of formalised methods of programming methodologies were examined. The study aimed to determine which, if any, existing programming methodologies would

be most appropriate for British Industry when programming embedded systems using the language Ada. The complete study described 21 methodologies and outlined a further 15. In the analysis of methodological features Pickett observed the difficulty of supplying a reliable system when the requirements may change between the inception of the project and its completion. Further, the study determined that the objectives of a programming methodology suitable for embedded systems are rigorous checking of the requirements with the produced system, formality of specification, rapid prototyping of the system and automation of as many parts of the software production cycle as practical, without a reference to ensuring safe operation of the software developed on the methodology.

The study concluded that whilst many of the methodologies provided some of the required features none of the methodologies fully met the study objectives. Methodologies such as CCS, HDM, JSD, SARA and VDM were considered to provide most features.

The second study, Wasserman and Freeman, [67], examined 24 methodologies. The study, known as "Methodman" complemented the earlier "Steelman" [13], and "Stoneman", [12], documents. The emphasis was on the software issues rather than on the more general issues of systems engineering. Concern was expressed by the authors of the study that inadequate analysis is "virtually certain to lead to project failure" because of a resulting poor specification.

In support of functional specifications the "Methodman" study asserts that functional specifications are "the basis against which validation is performed .... whether by acceptance testing or through formal proof of program correctness."

Twelve requirements are listed, p.7, as being essential for a methodology but none specifically refers to the need to ensure a safe software product. However, in the constraints, p.9, the effects of

the program on the development cycle are acknowledged as coming from the severe constraints often placed on embedded systems, for instance real-time responses and memory usage.

The 'technical characteristics' of a methodology, p.13, for embedded systems are considered to include "reliability - the absence of errors that lead to system failure" and "safety - the avoidance of run-time failures which could lead to the loss of life or the occurrence of other catastrophic consequences". Yet these two issues were not addressed either in the questionnaire or the evaluations of the responses to the questionnaire.

Both studies, [10], and, [67], acknowledge the requirement to ensure safe and reliable programs but failed to determine which, if any, methodology addressed the requirement of safety and the developers of the methodologies failed to indicate that the safety requirement had been addressed in their methodology.

2.4 Support Environment and Testing Strategy

The provision of software tools, compilers and editors, is the role of the programming support environment. The programming support environment may also be an aid to the programmer in the production of programs by maintaining a single project database of approved interface standards, common modules and standard testing facilities.

2.4.1 Support Environment

Some published material conflicted on precisely what was the correct view of a programming support environment but Lehman, [40], makes a contribution to this conflict of views when he declared that the programming process is "the transformation of a computer-application concept into an operational system and the subsequent evolution of that system to maintain it satisfactory and effective in its changing operational environment".

Degano and Levi, in [8] pp.251-264, assert that by making full use of the resources of the programming support environment the programmer is able to construct a program compatible with those of the rest of the project and the programmer is able to test his program in a consistent manner. Although the production of software with a programming support environment is more efficient in terms of costs, has a more consistent structure and it is more probable that testing will have been conducted within a better framework, there is no evidence to suggest that the safety of resulting programs is any better.

No published material was found to demonstrate how a programming support environment will influence a program's ability to meet the required safety criteria.

2.4.2 Test Strategy

Rushby, in Meek [46] p.87, states that "it is program testing, rather than debugging, which is the central feature of the final stage in the creation of a program. The objective of testing is to verify that the program functions as it should, that it conforms with its specification, and solves the right problem in the real world". Rushby developed his argument until concluding that it is reasonable "to stop error hunting when only a relatively small number of errors are left and the costs of finding any more are not justified". Assuming that a program contains any number of errors without some method of measuring the number of errors remaining, or their effect, infers that the remaining errors are benign.

Zweben, in [8] pp.3-12, states that no single test strategy is sufficient to satisfy all test conditions and recommends that a good testing strategy should be capable of determining that errors exist. Miller, in [33] pp.4-16, lists some of the benefits of program testing as being better user acceptance because the software is more reliable, demonstrable history of high-quality performance and confidence in the software product.

The testing strategy adopted is considered by Rushby, in [46], as influencing the production of the software and as a consequence the safety of the software.

In recognition of the need to approach a uniform testing strategy national regulatory bodies are examining ways of assessing various factors concerned with testing, Elektronik Centralen, [17].

2.4.3 Program Proving and Correctness Methods

Ramamoorthy and Ho, [58], demonstrate how a program with only nine paths can have an extremely large number of execution sequences thus making exhaustive testing impractical.

Proofs of correctness decompose the software logic into

axiomatic statements using mathematical notation to develop a mathematical proof. Program proving is a specialised and protracted activity with little evidence of what can be formally proved other than the absence of certain specific hazards, like the output from a variable which has not been assigned following initialisation. Criticism has been expressed by Cho, [7], as to whether a proof can itself be proved to be correct.

According with the view of Cho, [7], is that of Good and London, [23], when they observed that a 433 statement program required 46 pages of formal proof.

Validation and verification techniques abound but there is difficulty in establishing a general definition of the terms. Myers, [50], asserts that validation and verification are similar to correctness proofs, except that validation aims to find errors by running the program in a real environment, whilst verification aims to find errors by running the program in a test environment. Other definitions, Bologna, [6], and Dahll et al, [9], suggest that verification is the testing of a subset of the total program suite and that validation is the testing of the total program suite.

The idea that a software module can be analysed for structuredness by measuring topological features without consideration of the logic it portrays is given in the work of Hennell, [27], on LCSAJ (Linear Code Sequence and Jump) and Tai, [65]. Essentially, the technique relates to the number of crossings of flow or control paths within the program code. Such a measure is called 'knot complexity'. Woodward, [70], compares the knot complexity of 26 programs with McCabe's cyclomatic complexity, V(G), for the same set of programs and found a close correlation.

Huang, [32], gives a comprehensive overview of the most commonly used testing methods.

2.4.4 Software Metrics

Lord Kelvin, [44], is often quoted as having said;

"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."

The view of Lord Kelvin summarises the objective of the work on software metrics.

Testing of software is an aimless task unless some measure is used to indicate the effectiveness of such testing. Software Metrics aim to establish methods of measurement relating to the software.

Halstead, [25], introduced the phrase 'software science' to describe a set of empirically derived measures of the software based on phenomanological aspects of the software. There have been many other researchers in software metrics, notably McCabe, [45], who defined a measure based on a graph theoretic approach and known as the 'cyclomatic number'. Many metrics have been developed and Perlis et al, [54], considered some of these metrics and recommended research into software metrics.

Gilb, [22], presented a set of metrics but little material has been published regarding their derivation or application. Harrison et al, [26], reviewed many metrics concerned with complexity and found that their experiments supported Gilb's assumption that the degree of decision-making logic in the program can be correlated to characteristics of a program such as error proneness, development costs and time. Findings similar to these are reported by Farr and Zagorski, [18], and Sime et al, [63].

Shen et al, [62], critically examined the work of Halstead and the experimental results published by other authors in support of Halstead and concluded that the theory of software science was still evolving. Further, they resolved that researchers should continue to refine Halstead's metrics as there is a need for such measures.

2.4.5 Simulation

The use of software or hardware simulations, designed to test the control software prior to implementation, is not common due to the high costs involved in the development of a simulator.

The method commonly used in Industry is to construct a panel of switches and knobs to allow simulation of the anticipated normal input-output sequence and, to a lesser extent, the known exception paths from a restricted data set. The disadvantages of a hardware simulation are the high costs involved, the time to produce the simulator, the need for manual operation of the simulation and the need for a protracted and accurate repetition of the test causes doubt to be cast on the effectiveness of such simulation tests.

Software simulation has been shown by Nunns, [51], and others to overcome many of the drawbacks experienced by the hardware approach but the cost of developing a software simulator is still high. Using a software simulation it is possible to establish a detailed simulation of the expected plant input sequences that will exercise either separate programs or complete systems. With a software simulator it is possible to run these simulation sequences for protracted periods of time and often at a rate of input sequences greater than or less than those to be expected in real-time operation.

A development of software simulation is the use of simulation monitors which log data from specific application areas, such that when a malfunction occurs in plant operation a simulation model is

available to the Plant Engineer who can in turn request further controlled testing from the Programmer.

One major difficulty with software simulations is ensuring the faithfulness and accuracy of the simulator.

2.5 Operational and Psychological Factors

At all stages of the software production cycle people are involved. Since people are fallible the effects of their failings can be seen in the errors introduced into the software.

Hirsgh, [30], reported that in an experiment to determine the error rate of humans operating a typewriter keyboard a total error rate of 6.175% was encountered out of a sample of 5 million key depressions. An error rate of this magnitude infers that on average one in every 16 keys depressed will be in error and with most programs containing many hundreds of characters a considerable number of characters can be expected to be in error.

2.5.1 Psychological Factors

Green et al, [24], suggest that software production is a design activity and dependent on the mental agility of the programmer. Errors can, therefore, be induced into a program due to psychological factors.

Kopetz, [38], cites Per Brinch Hansen as having said;

> "If the intellectual effort required to understand and test a system increases more than linearly with the size of the system, we shall never be able to build reliable systems beyond a certain complexity."

In the works of Mohanty, [48], and Fitzsimmons and Love, [20], there are references to a Stroud number which is derived from the definition of a 'moment' given by J.M. Stroud;

> "The time required by the human brain to perform the most elementary mental discrimination."

The number of mental discriminations required to understand a software module influences the production of software according to the amount of effort required.

Estimates of an individual's Stroud number (the number of

'moments' in a second) range from 5 to 28 and have been used in experiments concerned with programming rates. Halstead, [25], used a Stroud number of 18 to indicate what he considered to be a reasonable level of mental activity for a concentrating person. The usefulness of measures based on the Stroud number is supported by Basili, in [54], yet discounted by Curtis, in [54]. Researchers such as Mohanty, [48], and Sime, [63], hold the view that the greater the effort required, the more the risk of inducing errors yet there is no consensus view on the usefulness of psychological measures.

Wasserman and Freeman, [67], acknowledged that there is a psychological factor affecting the development of software as a result of what they have called the "physical workplace", that is to say the actual place where the developer undertakes the development. The factors that they refer to include "access to computers, privacy and noise levels, ergonomic considerations of terminals, and availability of reference materials including books and journals". They further suggest that there is little doubt that these factors are significant.

None of the references suggested ways of restricting the influence of the psychological factors which affect the programmer other than methods of detecting and measuring the extent to which there is an effect. Research into the psychology of programming continues but few applications of the findings of such research have been reported.

2.5.2 Operational Factors

The operational factors include the industrial equipment and the process or the plant being controlled by the software.

The safety of control systems incorporating software is influenced by the activities of external factors, for example, the need for safe operation of the software to be maintained when the

equipment malfunctions. In such cases the equipment will provide erroneous information for the software to interpret but the software should be designed to cater for such events. Though the particular event may have been considered improbable.

Hardware failures may suggest to the observer that the software malfunctioned instead of the equipment whilst the software may have reached a reasonable interpretation of the information. The software may be considered to have failed to meet the safety requirement yet in reality it was the equipment reliability that was suspect.

The operation of the software should not compromise safety because of operational difficulties. Low reliability of equipment will cause an initial low confidence in the safety of the software since the equipment and software are frequently viewed as one.

Longbottom, [43], and Williams, [68], have suggested that hardware failures influence the production of software. Suggestions such as these have led to software reliability being measured in such terms as 'errors per 1,000 hours'.

Anderson and Lee, [2], have investigated the effects of hardware failure on software and the outcome of their investigations are ideas such as fault-tolerant computing. Fault-tolerant computing is an extremely large field of study and in general is more concerned with equipment reliability than safety.

At the final stage of testing many errors will remain in the software. So the provision of satisfactory documentation to enable comprehensive testing should be mandatory for all software projects according to Hewitt, [29].

Hewitt, [29], and Johnson, [35], have suggested that documentation should be built up as the project progresses. They suggest that a poorly documented project will also be subjected to a

restricted set of tests and if the test set is limited by the documentation then it is held that documentation influences software safety.

It is important to document changes to the software. Lawley, [39], developed a scheme, known as HAZOP, for documenting the desirable and undersirable effects of changes proposed for chemical plants. Nunns, [52], and, [53], has shown that a modified HAZOP procedure can be implemented for software.

2.6 Conclusion

Many factors have been advocated as those influencing software but none were found which were claimed to specifically influence the safety of software.

It may be that factors affecting the safety of software can be identified but then there needs to be a knowledge of how to manipulate them, how to measure them and what such measurements mean.

Any set of measurements of factors will need to address three points of issue for each measurement;

1. the relative criticality
2. the relative importance
3. can it be assessed

From the current state of the art consistent opinion is that there are factors influencing software and there is a consensus on their likely effects but there is no evidence to isolate which features these are.

2.7 References

[1] Allworth, S.T.,Introduction to Real-Time Software Design, MacMillan, 1981

[2] Anderson, T., & Lee, P.A.,Fault Tolerance; Principles and Practice, Prentice-Hall, 1981

[3] Andow, P.K., The Numerical Analysis of Hazards & Failures, in Proceedings of the Institution of Chemical Engineers Symposium Number 63

[4] Basili, V.R. & Perricone, B.T., Software Errors and Complexity: An Empirical Investigation, Communications of the ACM, Volume 27, Number 1, January 1984, pp.42-52

[5] Berg, H.K. et al, Formal Methods of Program Verification and Specification, Prentice-Hall,1982

[6] Bologna, S., Guidelines for Verification and Validation of Safety Related Software, European Workshop on Industrial Computing, Technical Committee Number 7, Paper 287, 1982

[7] Cho, C-K.,Software Quality Control, Wiley, 1980

[8] Chandrasekaran, B. & Radicchi, S. editors, Computer Program Testing, North-Holland, 1981

[9] Dahll, G. et al, Techniques for Verification and Validation of Safety Related Software, European Workshop on Industrial Computing, Technical Committee Number 7, Paper 267, 1983

[10] Dept. of Industry, ADA-Based System Development Methodology Study Report, Dept. of Industry, London, 1981

[11] Dept. of Defense, Reference Manual for the Ada Programming Language - Proposed Standard Document, United States Department of Defense, November 1980

[12] Dept. of Defense, Stoneman: Requirements for an Ada Programming Support Environment, February 1980

[13] Dept. of Defense, Steelman: Requirements for a High-Order Language, June 1978

[14] Dept. of Employment, The Flixborough Disaster: Report of the Court of Inquiry, H.M.S.O., 1975

[15] Dijkstra, E.W., Go To Statement Considered Harmful, Communications of the ACM, Volume 11, Number 3, March 1968, pp.147-148

[16] Dijkstra, E.W., Structured Programming, in Software Engineering Techniques, NATO Science Committee, 1969, pp.84-88

[17] Elekronik Centralen, Directive for Testing Software Quality in Control and Surveillance Systems, Denmark, 1983

[18] Farr, L. & Zagorski, H.J., Quantitative Analysis of Programming Cost Factors: A Progress Report, in Economics of Auto Data Processing, Proceedings of ICC Symposium, North Holland, 1965

[19] Finkelstein, L., Fundamental Concepts of Measurement: Definition and Scales, Measurement and Control, Volume 8, March 1975, pp. 105 - 111

[20] Fitzsimmons, A. & Love, T., A Review and Evaluation of Software Science, Association of Computing Machinery Computing Surveys, Volume 10, Number 1, March, 1978, pp. 3-18

[21] Gannon, J., An Experimental Evaluation of Data Type Conventions, Communications of the ACM, August 1977

[22] Gilb, T., Software Metrics, Studentlitteratur, Sweden, 1980

[23] Good, D.I. & London, R.L., Computer Interval Arithmetic: Definition and Proof of Correct Implementation, Journal of the ACM, Volume 17, October 1970, pp. 603-612

[24] Green, T.R.G. et al, The Problems The Programmer Faces, Ergonomics, Volume 23, Number 9, 1980, pp. 893-907

[25] Halstead, M., Elements of Software Science, North-Holland, 1977

[26] Harrison, W. et al, Applying Software Complexity Metrics to Program Maintenance, IEEE Computer, September 1982

[27] Hennell, M.A., Systematic Software Validation: The First Ten Years, University of Liverpool Report

[28] Henry, B., Some Remarks about Systems Requirements for Industrial Programmed Equipment, European Workshop on Industrial Computing, Technical Committee Number 7, Paper 277, 1982

[29] Hewitt, D.J., Editor, Guide to the Quality Assurance of Software, Electronic Engineering Association, 1978

[30] Hirsgh, R.S. Human Factors in Man-Computer Interfaces, IBM Human Factors Centre, USA, 1976

[31] Horning, J.J., Programming Languages in Computing Systems Reliability, Cambridge University Press, 1978, pp. 109-152

[32] Huang, J.C., An Approach to Program Testing, Association of Computing Machinery Computing Surveys, Volume 7, Number 3, September, 1975, pp. 113 - 128

[33] IEEE, Tutorial: Software Testing & Validation Techniques, Institute of Electrical and Electronic Engineers, USA, 1981

[34] Infotech, Structured Programming, Infotech State of the Art Report on Structured Programming, 1976

[35] Johnson, K.S., Editor, Establishing a Quality Assurance Function for Software, Electronic Engineering Association, 1981

[36] Kernighan, B.W. & Plauger, P.J., Programming Style: Examples and Counter Examples, Association of Computing Machinery Computer Surveys, Volume 6, Number 4, 1974

[37] Knox, E.G., Negligible Risks to Health, Community Health, Number 6, 1975, pp. 244-251

[38] Kopetz, H., Software Reliability, MacMillan, 1979

[39] Lawley, H.G., Operability Studies and Hazard Analysis, Chemical Engineering Progress, Volume 70, Number 4, April 1974, pp. 45-56

[40] Lehman, M.M., Research Proposal to Study the Role of Executable Metric Models in the Programming Process, ACM Software Engineering Notes, Volume 7, Number 5, December 1982, pp. 106

[41] Littlewood, B., How to Measure Software Reliability and How Not To, Proceedings of the 3rd International Conference on Software Engineering, 1978, pp. 37-45

[42] Littlewood, B. & Verrall, J.L., Likelihood Function of a Debugging Model for Computer Software Reliability, IEEE Transactions on Reliability, Volume R-30, Number 2, June 1981, pp. 145-148

[43] Longbottom, R., Computer System Reliability, Wiley, 1980

[44] Lord Kelvin, Popular Lectures and Addresses, 1891-1894

[45] McCabe, T., A Complexity Measure, IEEE Transactions on Software Engineering, Volume SE-2, December 1976, pp. 308-320

[46] Meek, B. et al, Guide to Good Programming Practice, Ellis Horwood, 1983

[47] Miara, R.J. et al, Program Indentation and Comprehensibility, Communications of the ACM, Volume 26, Number 1, November 1983, pp. 861-867

[48] Mohanty, S.N., Models & Measurements for Quality Assessment of Software, Association of Computing Computer Surveys, Volume 11, Number 3, September 1979, pp. 251-275

[49] Musa, J.D., Software Reliability Measurement, Journal of Systems and Software, Volume 1, Part 3, North-Holland, 1980, pp. 223-241

[50] Myers, G.J., Software Reliability: Principles & Practices, Wiley, 1976

[51] Nunns, S.R., A Software Simulator - an Aid to Plant Commissioning, European Workshop on Industrial Computing, Technical Committee Number 7, Paper 313, 1982

[52] Nunns, S.R., A Formal Approach to Software Change Control, European Workshop on Industrial Computing, Technical Committee Number 7, Paper 316, 1982

[53] Nunns, S.R., Some Further Examples of Software Change Control, European Workshop on Industrial Computing, Technical Committee Number 7, Paper 328, 1982

[54] Perlis, A.J., et al, Software Metrics, The MIT Press, 1981

[55] Pyle, I.C., Methods of the Design of Control Software, Department of Computer Science Report YCS.12, University of York, 1978

[56] Pyle, I.C., Review of Standards in Software for Real-Time Computing, Real-Time Data Handling & Process Control, North-Holland, 1980

[57] Pyle, I.C., Towards Specifying and Information System, Department of Computer Science Report YCS.43, University of York, 1981

[58] Rammamoorthy, C.V. & Ho, S.F. Testing Large Software with Automated Software Evaluation Systems, IEEE Transactions on Software Engineering, Volume SE-1, Number 1, 1975, pp. 46-58

[59] Rault, J-C., The Many Facets of Quantitative Assessment of Software Reliability, IEEE Workshop on Quantitative Software Models, Kiamesha Lake, New York, October, 1979, pp. 224 - 231

[60] Rault, J-C. & Bouissou, Quantitative Measure for Software Reliability: A Survey, in State of the Art Report on Software Testing, Infotech, 1978, pp.215 - 229

[61] Rzevski, G., Identification of Factors which cause Software Failure, Proceedings of the Annual Reliability and Maintainability Symposium, Los Angeles, 1982

[62] Shen, V.Y. et al, Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support, IEEE Transactions on Software Engineering, Volume SE-9, Number 2, March 1983, pp. 155 - 165

[63] Sime, M.E. et al, Structuring the Programmer's Task, Journal of Occupational Psychology, Volume 50, 1977, pp. 205 - 216

[64] Starr, C., Rudman, R., & Whipple, C., Philosophical Basis for Risk Analysis, Annual Review of Energy, Number 1, Annual Reviews Inc., USA, 1976

[65] Tai, K-C., Program Testing Complexity & Test Criteria, IEEE Transactions on Software Engineering, Volume SE-6, Number 6, November 1980, pp. 531-538

[66] Taylor, J.R., Logical Validation of Safety & Control Specifications Against Plant Models, Report RISO-M-2292, RISO National Laboratory, Denmark, 1981

[67] Wasserman, A.I. & Freeman, P., Ada Methodologies: Concepts and Requirements, Department of Defense, ACM Software Engineering Notes, Volume 8, Number 1, pp. 33 - 50

[68] Williams, J.R., Reliability in a Process Control System, in State of the Art Report on Computer System Reliability, Infotech, 1974, pp. 373 - 388

[69] Wirth, N., Program Development by Stepwise Refinement, Communications of the ACM, Volume 14, Number 4, April 1971, pp. 221 - 227

[70] Woodward, M.R. et al, A Measure of Control Flow Complexity in Program Test, IEEE Transactions on Software Engineering, Volume SE-5, Number 1, January 1977

[71] Young, S.J., Real-Time Languages: Design and Development, Ellis Horwood, 1982

CHAPTER 3

The Structural View

The structure of the software has an influence on the safety of the software. In this Chapter the ways that the software can be structured to ensure a safe operation and methods of analysing that safety will be examined.

The Chapter begins by examining the use of a set of techniques known, generically, by the term Risk Analysis. In particular, the applicability to software of Fault Tree Analysis and Event Tree Analysis is explored.

State Transition Diagrams are sometimes considered to be the means of identifying all possible fault conditions. State Transition Diagrams are examined with particular reference to the interaction between software, the hardware and the system.

Having examined methods that may be applicable in isolating a fault condition an argument is advanced for weighting errors according to three categories of danger.

Finally, the structuring of the software for safety according to the control flow is examined. It is recommended that in safety-related systems the software should be structured into Control Modules, Safety Modules and Arbitrators. It is also suggested that a system of Integrity Locks should be used.

3.1 The Risk Analysis of Software

'Risk Analysis' is a generic term used by Engineers to describe a group of methods used to determine the conditions that will cause a hazardous state to exist and the associated risk. There is a need to assess the risk resulting from the use of computers as controllers in safety-related processes. The principal cause for concern is the possible number of software errors that can exist and the effects of these errors on the system. Since these techniques are used to analyse the risk associated with industrial processes and its hardware, it follows that control software should also be subject to similar analysis.

Risk Analysis comprises a collection of analytical techniques used to examine the design of complex items of equipment within a safety context. The principle risk analysis techniques are Fault Tree Analysis (FTA) and Event Tree Analysis (ETA), [12]. The application of both these risk analysis techniques to software will be discussed in this Chapter.

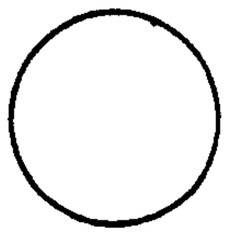
3.1.1 Fault Tree Analysis

Chelson, [4], has shown that fault-trees are constructed by first listing all the possible hazards considered to be present in the system. Once the hazards have been listed the construction of a fault tree begins by assuming that a particular event has caused one of the hazards and then to trace backwards through the logic of the system to find which events could lead to the hazard. Since preceeding events may be the logical combination of other events a set of symbols is used to represent the logical sequence of possible events. As each node in the tree is encountered a decision is made whether further investigation is required. As the investigation continues more symbols are included in the tree until a node is reached where either no further investigation is necessary, called a

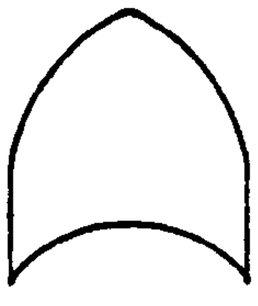
Failure Event, or a Terminal Event, also called a Basic event, is encountered. These symbols are shown in Figure 3.1.1.1.

Figure 3.1.1.1 FTA Symbols

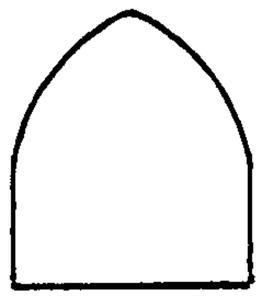
Top Event



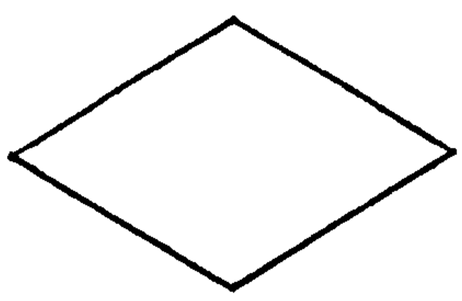
Terminal or Basic event requiring no further investigation



OR gate



AND gate



Failure event, not a basic fault event but one which requires no further investigation

Leveson and Harvey have shown, [8], that Fault Tree Analysis can be applied to software provided that the catastrophic event which is to be considered can be defined in a precise manner. Since FTA was developed for hardware and has now been applied to software, it is possible to link the two sets of analyses to form a complete set

for the total system. Fault Tree Analysis applied to software has been renamed by Leveson and Harvey as Software Fault Tree Analysis (SFTA).

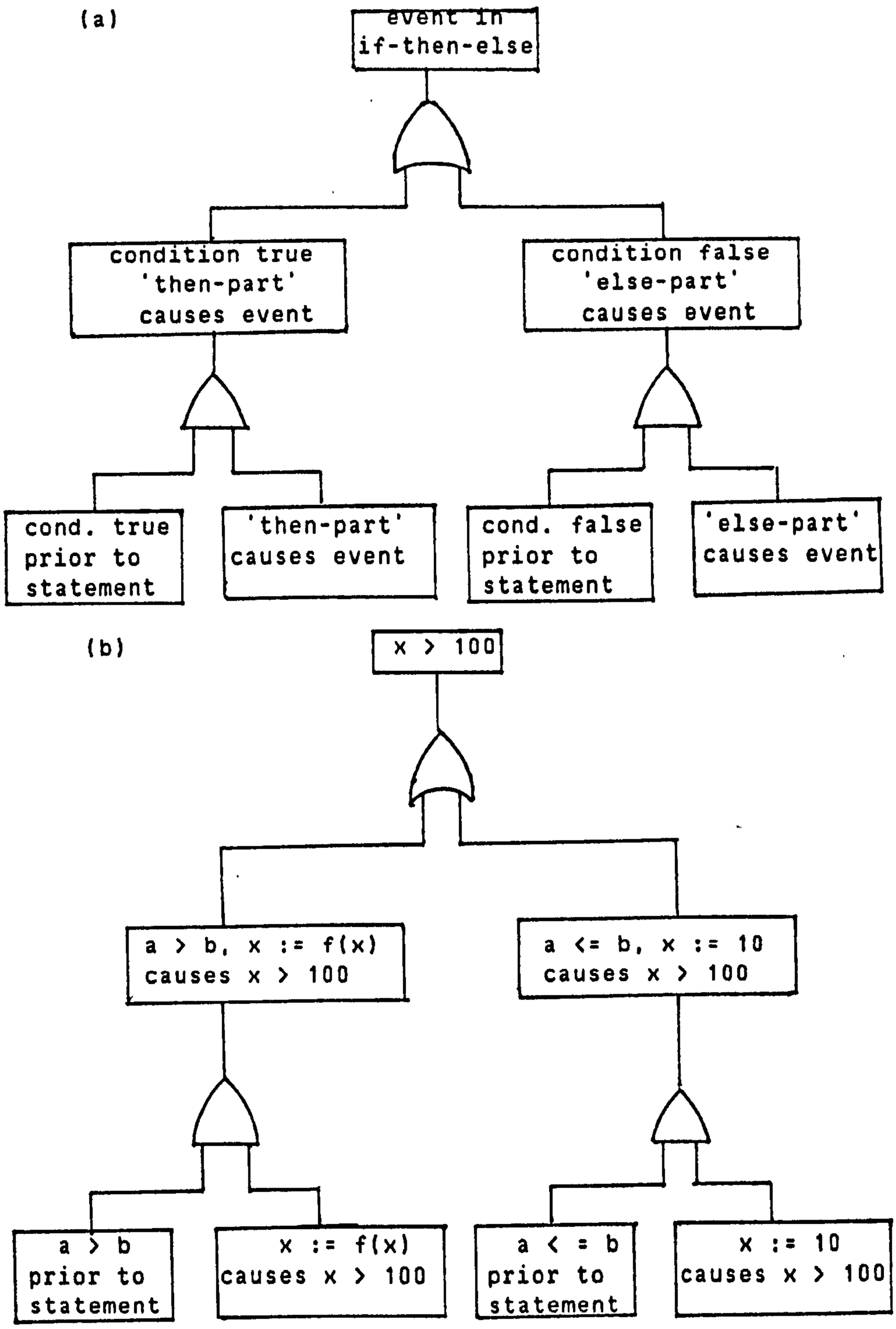
SFTA, in common with hardware FTA, examines the potentially dangerous conditions that could occur, called 'catastrophic events', as a result of 'top events' or 'loss events', and considers all possible actions that could cause the dangerous condition to exist using diagrams which are a variation of those used for hardware FTA.

Leveson and Harvey, [8], have also shown that SFTA can be performed at various levels and stages of software development. The highest level of analysis is the functional description. At the lowest level of investigation SFTA analyses the program code. Leveson and Harvey also suggest that it is possible to construct fault trees from a program design language and that the information derived from the tree during the software development phase can be used. However, SFTA does not cater for the effect of one part of a program influencing another.

In SFTA it is assumed that for a dangerous condition to exist it is necessary for there to be a related output from the computer. Therefore, the starting point for SFTA, when working at the program level, is the section of code responsible for effecting an output. The analysis then proceeds backwards through the code determining both how the program arrived at the section of code and what are the current states of the variables.

Standard forms of symbolism have been proposed by Leveson and Harvey for Pascal-like program statements. The general form for the IF..THEN..ELSE.. statement is shown in Figure 3.1.1.2 (a). The statement " IF a > b THEN x := f(x) ELSE x := 10 " is shown in Figure 3.1.1.2 (b) below when analysed for the event " x > 100 ".

Figure 3.1.1.2 SFTA for IF..THEN..ELSE..



Since the right-most node, stating that x:=10 causes x>100, is clearly nonsense the node can be assigned a zero probability and removed from the tree. Analysing for the top event of x>100 could stop at this point and assertions placed in the code or the preceeding code could be analysed for the events "a > b " and "f(x) > 100 ".

Figure 3.1.1.3 (a) shows the suggested general format for

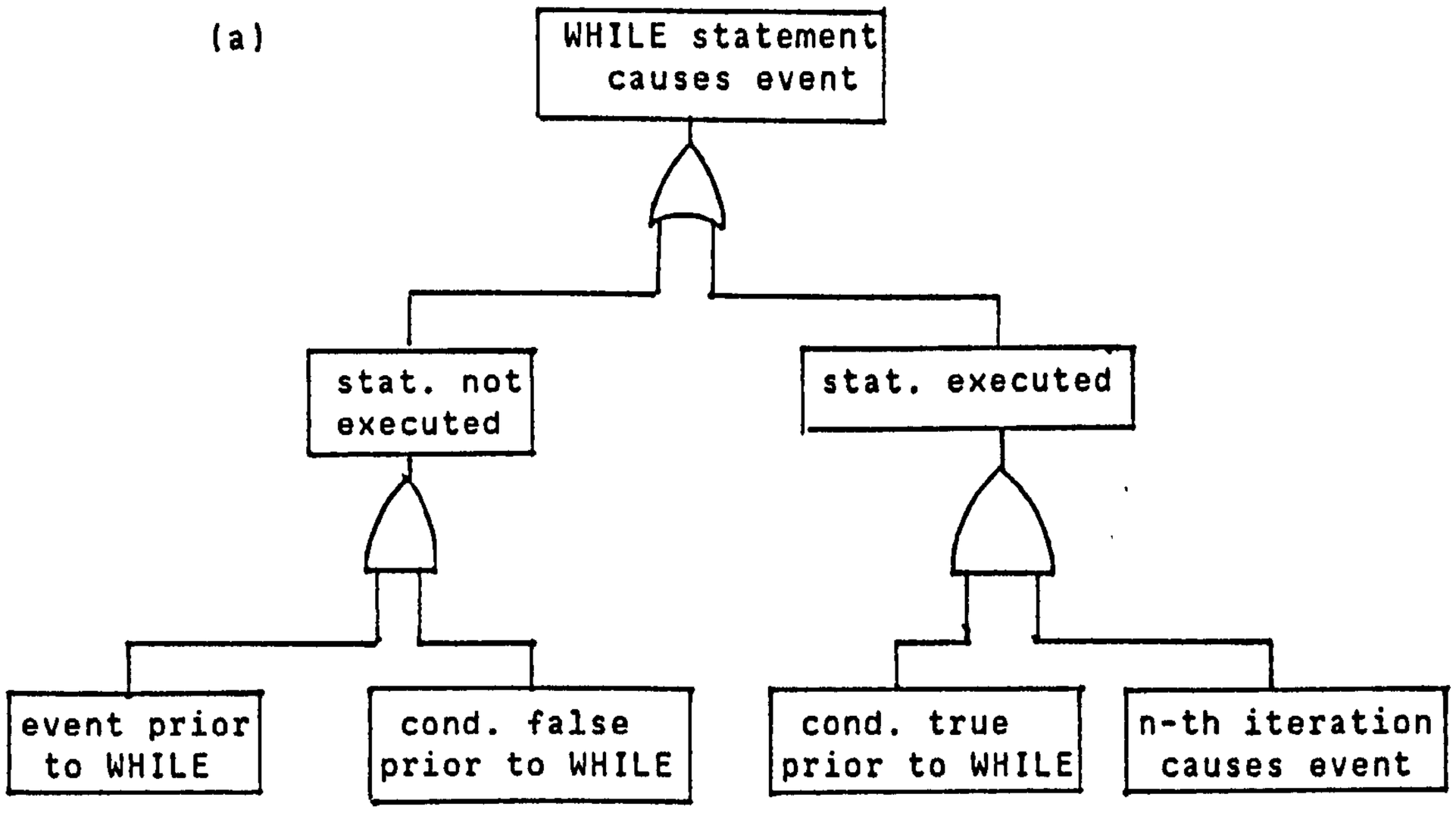
analysing a WHILE..DO statement and Figure 3.1.1.3 (b) shows the analysis for the loop

```
WHILE b > x DO
  BEGIN b:= b - 1;
        z:= z + 10;
  END
```

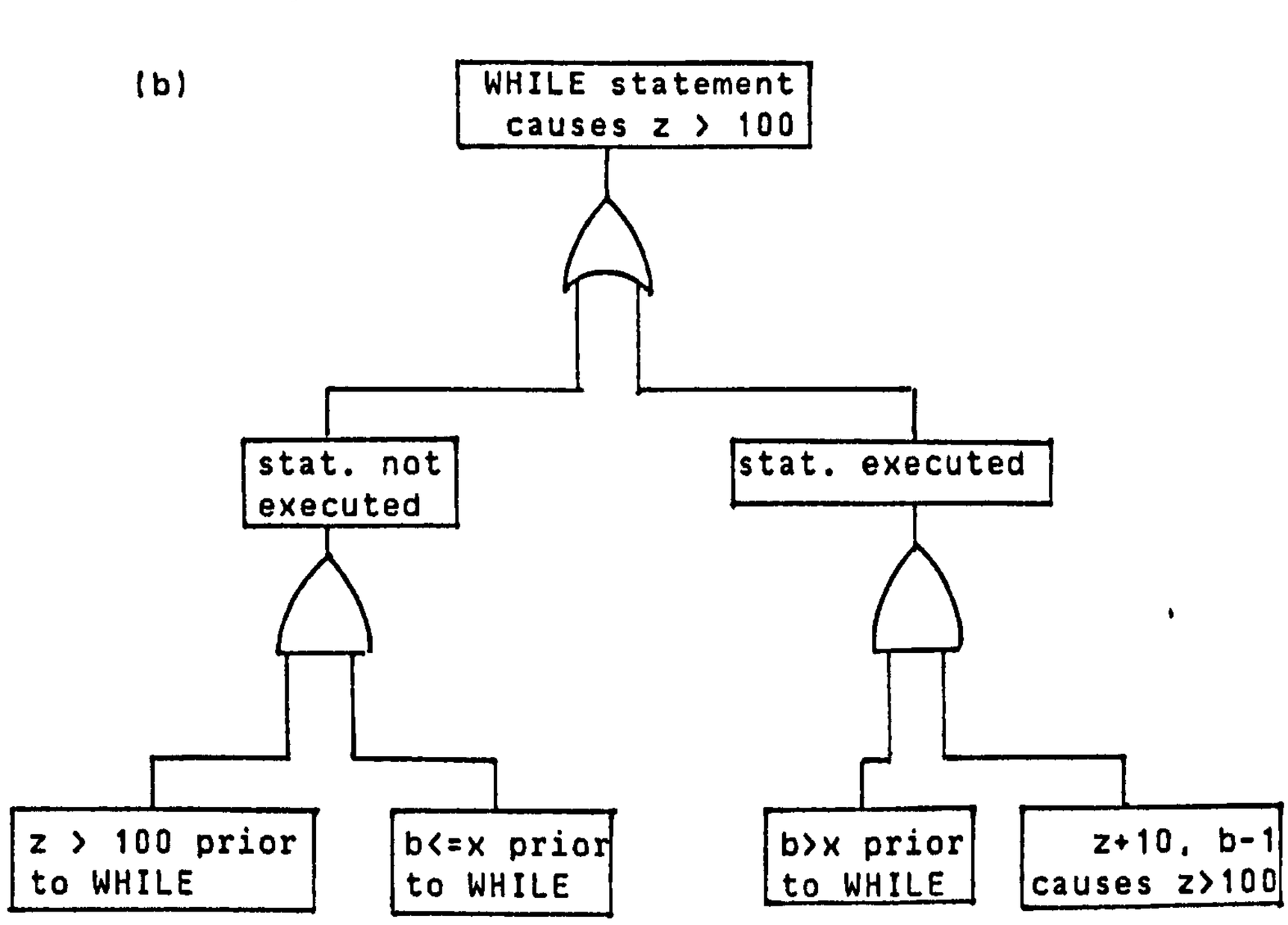
analysed for the top event " z > 100 ".

Figure 3.1.1.3 Example of SFTA

(a)



(b)



Leveson and Stolzy, [9], have suggested that real-time features, like concurrency found in the language Ada, can also be analysed

using SFTA.

A disadvantage with SFTA is the difficulty in determining all possible top events that may arise and assessing their preceeding events, called cut sets, and basic failure events, called minimum cut sets. SFTA is not exhaustive and relies upon the person analysing the system to identify the "top events". Also there is no check to indicate that the analysis is complete.

In the software context, tracing through the data flow of a program and analysing for failure events will identify some hazard situations which can be further analysed using SFTA. One method of tracing the data flow is to use Petri Nets or Event Tree Analysis.

3.1.2 Petri Nets

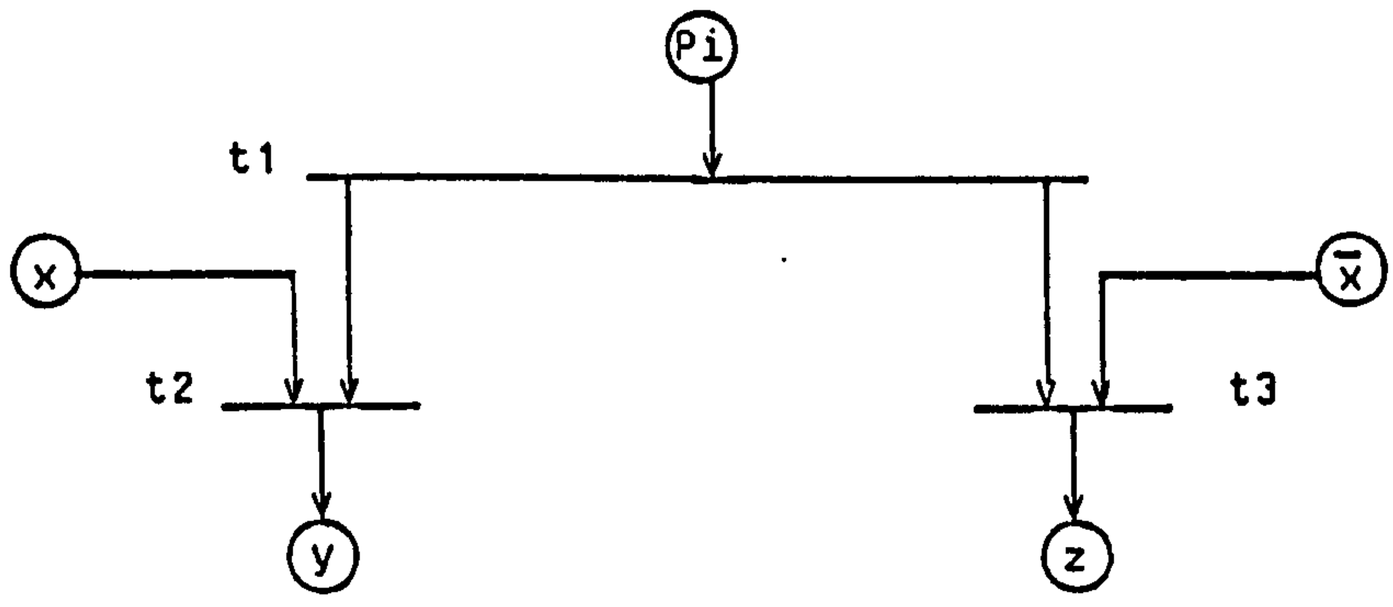
Petri Nets, [10], are formal methods of representing information flow and can be used to illustrate information flow in a program statement. Petri Nets can be used to represent the information flow at the level of the specification or at the level of the actual program.

Petri Nets are bipartite directed graphs consisting of two basic components; a set of places, P, and a set of transitions, T. In addition two functions are created to link transitions to places; the input function, I, and the output function, O. For each transition, tj, there is a set of input places, I(tj) and for each transition, tj, there is a set of output places O(tj). Formally, a Petri Net is made up of a quadruple C=(P,T,I,O). Since each Petri Net has an initial condition, μo, the initial condition needs to be included in the structure giving a quintuple (P,T,I,O,μ). Defining the initial condition of the Petri Net is called "marking" a Petri Net.

Diagramatically the places in a Petri Net are represented by circles and the transitions are represtened by a line crossing the arc joining two places. A transition is said to be enabled to

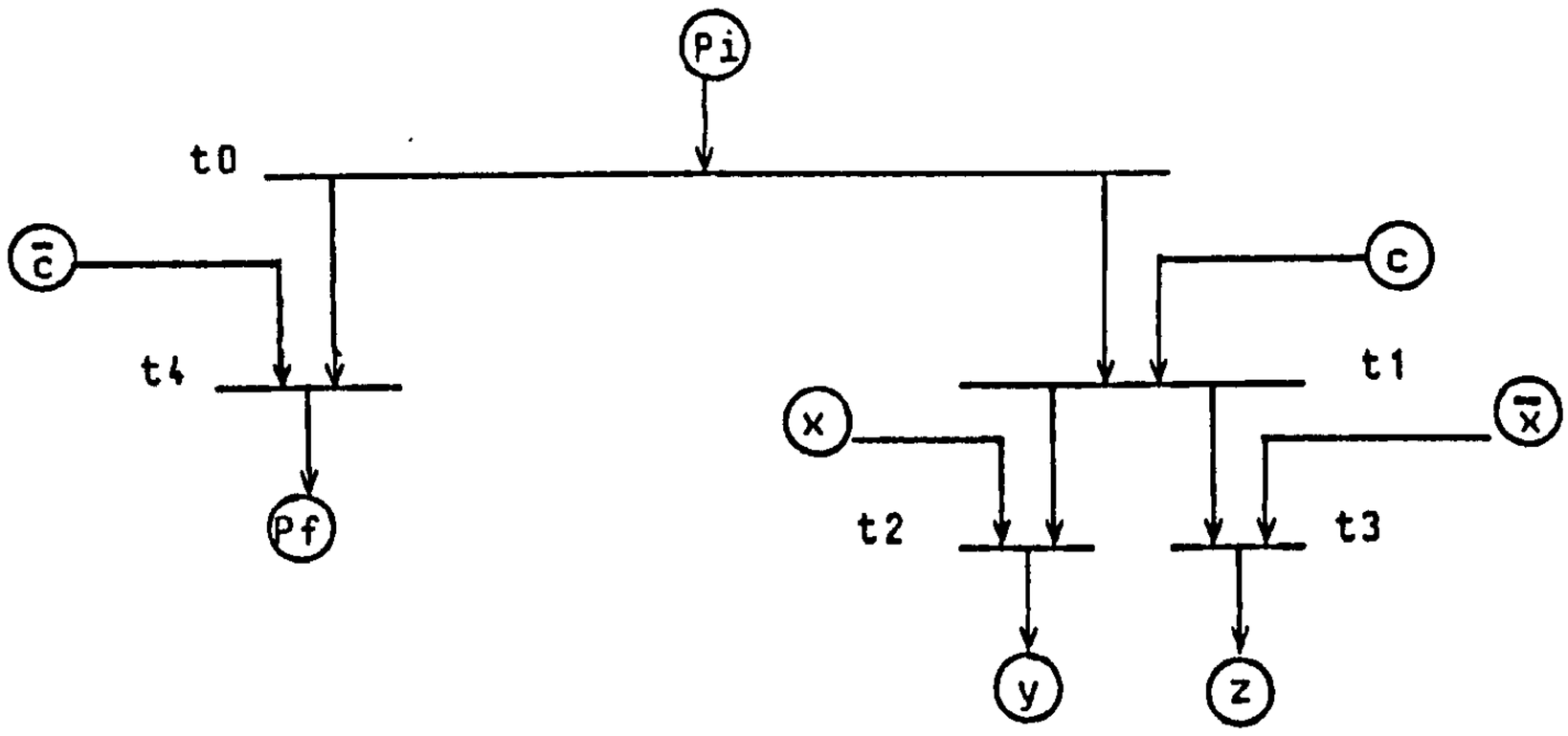
"fire" if and only if all the input tokens for that transition, markings, are satisfied and which allow the token at an input place to be transferred to an output place. The transition of the token is, in the abstract, the transfer of information from one place to another place.

When the statement IF x THEN y ELSE z is executed the control will pass to either y or z according to the truth of x. A Petri Net can be represented graphically for such a statement;



where Pi is the initial input place which fires transition t1. Transition t2 will only be fired and pass a token to y when t1 has fired and place x has a token (x is true). Transition t3 will fire and pass a token to z when t1 has fired and x bar has a token (x is false). For the IF x THEN y ELSE z statement places y and z would be the input places for the following statements.

The firing of transition t1 enables the firing of either t2 or t3 dependent on the logical state of x. However, when considering a programming statement according to a failure criteria it must be considered that the conditional expression, x, may also fail. If the possibility of the conditional statement, c, failing in the statement, IF x THEN y ELSE z, is included then the Petri Net becomes



The transitions t1 and t4 will fire according to the status of the conditional statement, c, and transitions t2 and t3 will fire according to the logical truth of the conditional expression, x. The failure of the conditional statement is called the conditional failure and the logical truth of the conditional expression is called the temporal switch.

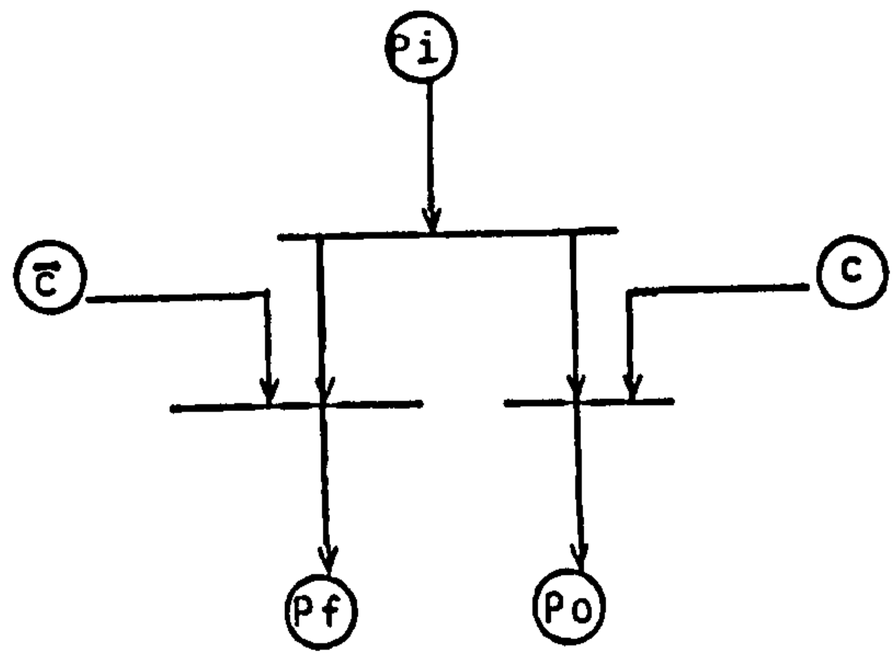
Figure 3.1.2.1 shows the general form for some Pascal-like programming language statements using Petri Net diagrams.

Petri Nets of complete programs become unmanageable and need simplification. One method for simplifying the representation of failure events is to use a Risk Analysis technique called Event Tree Analysis.

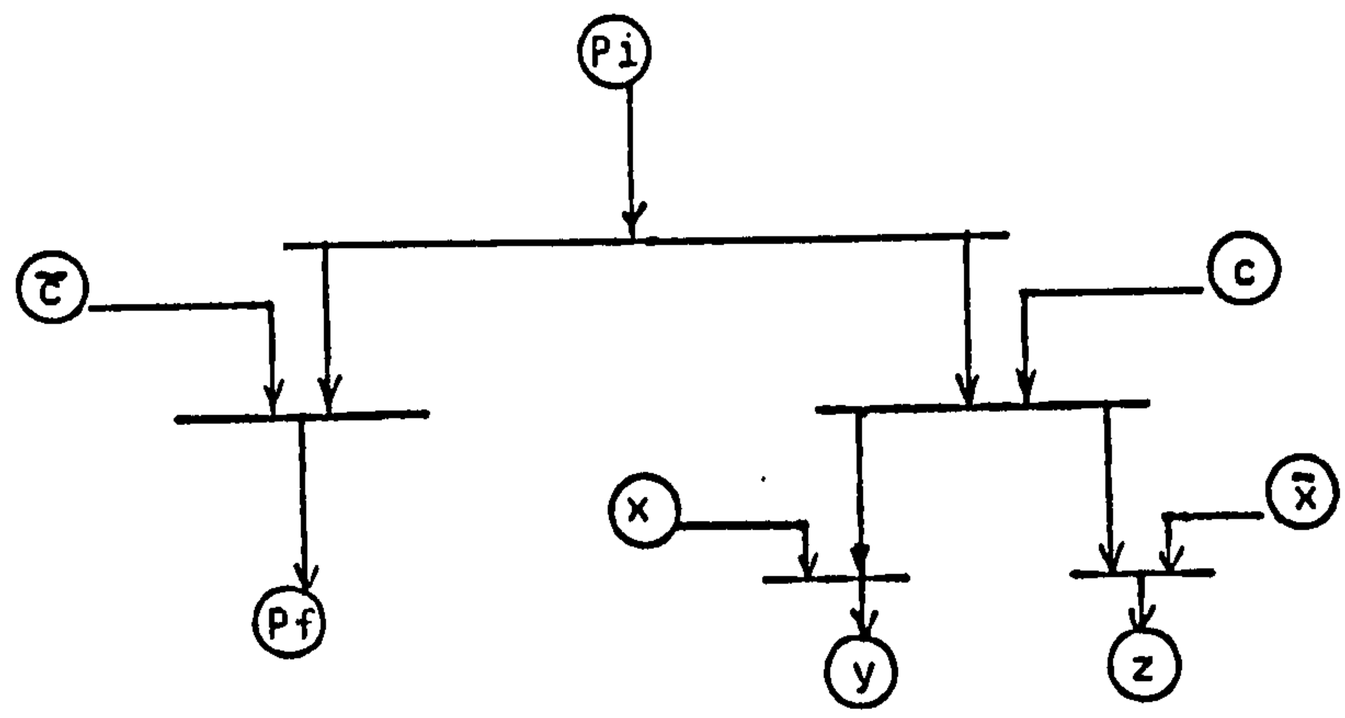
For a graphical representation of an abstract model of information flow to be useful in identifying risks the probabilities of failure for components of the model need to be added. The addition of such probability data to a Petri Net will detract from its function of representing the logical sequence.

Figure 3.1.2.1 Program Statements using Petri Nets

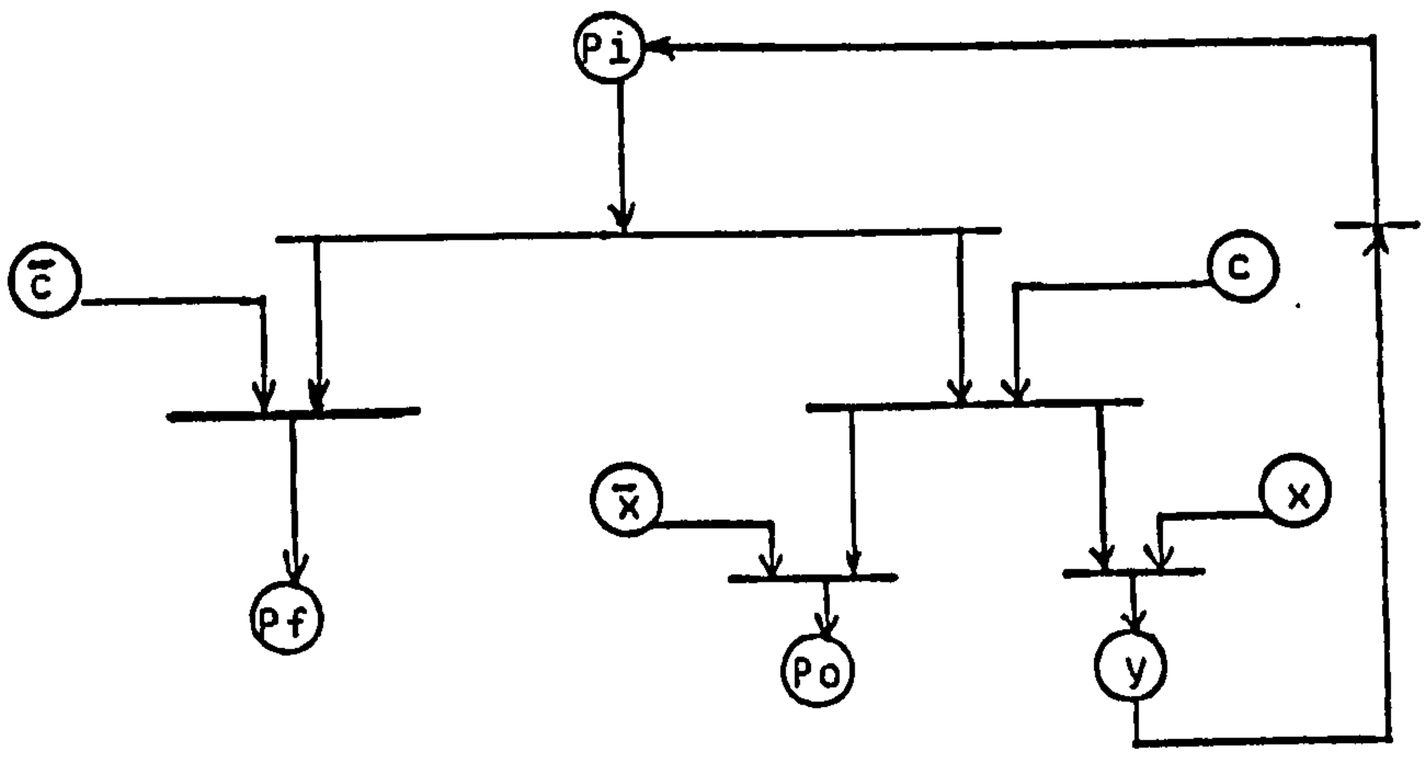
a) assignment



b) IF..THEN..ELSE..



c) WHILE x DO y



3.1.3 Event Tree Analysis

Event Tree Analysis (ETA) is less common than FTA but is becoming more commonly used in Industry, [2] and [3].

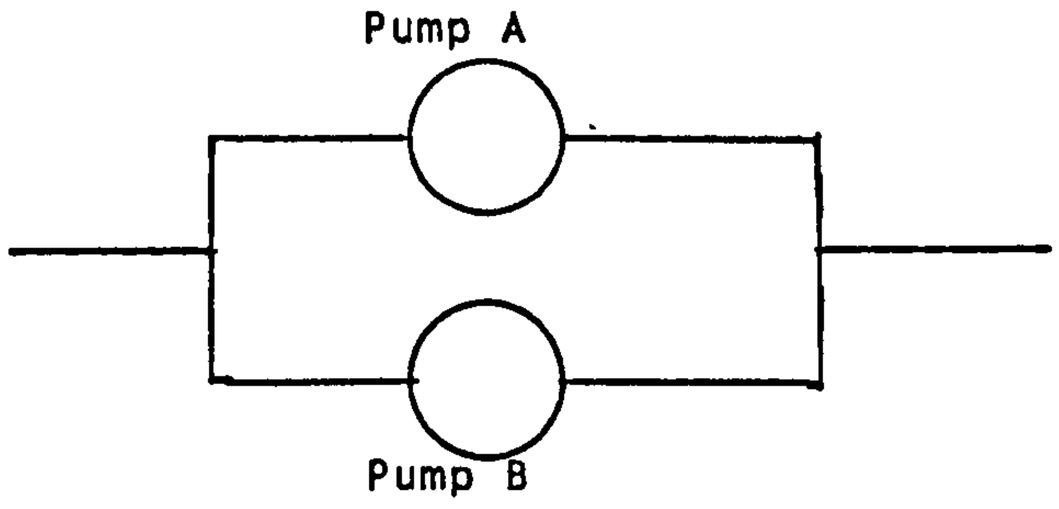
Hardware ETA attempts to identify those events which may cause a sequence of events leading to a dangerous condition and can be considered as an approach to the task of identifying risks from the lowest event towards the 'top event'. FTA starts with the 'top event' and traces back to the lowest event in the sequence examining the causes of events. ETA examines the consequences of possible failures. The use of 'event trees' provides a graphical method of presenting the results of the analysis.

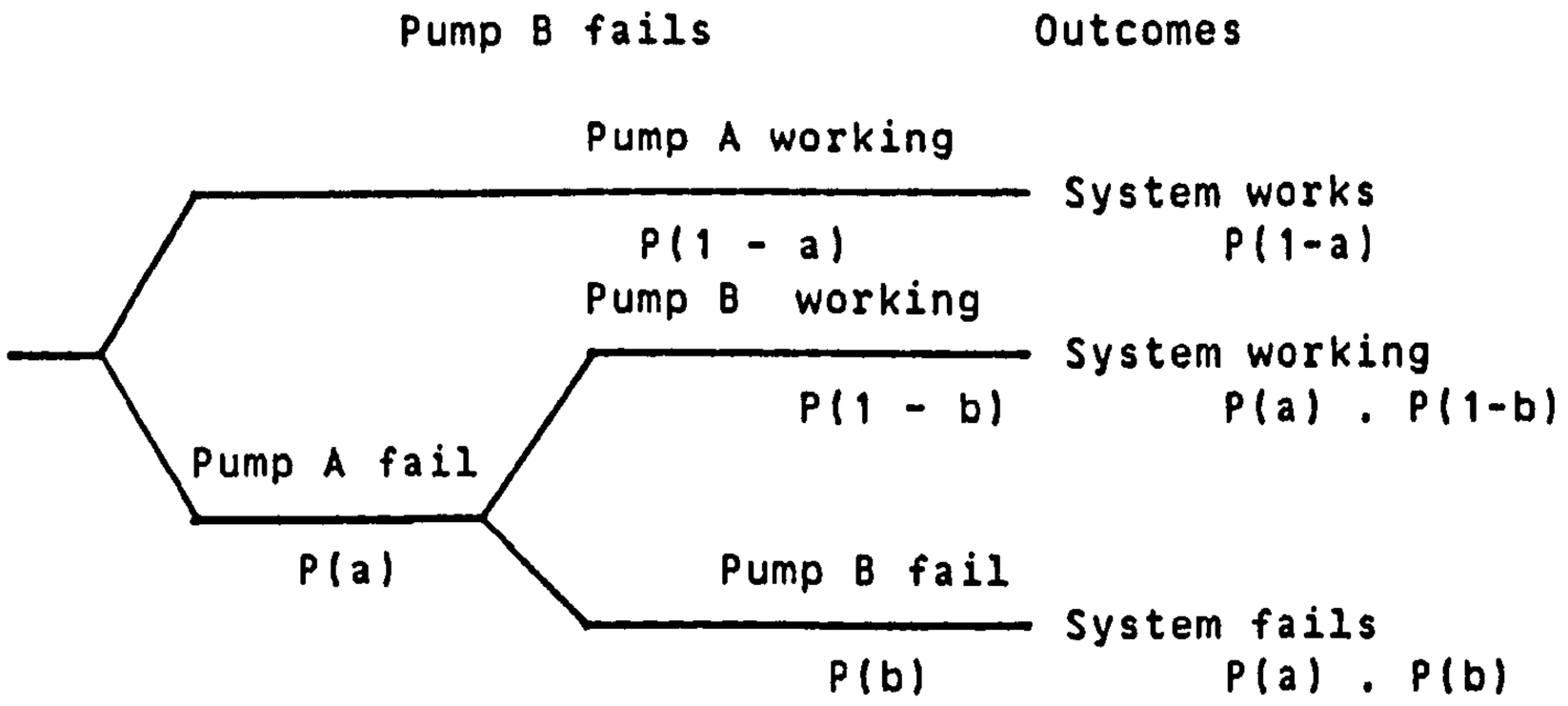
To construct an event tree of failures, each probable failure is considered from the start of the process being analysed to the finish. The first stage of the ETA construction is to consider the outcome of each component failure and to represent the outcome as a decision branch. For each outcome of the first stage consideration is given to the outcome of each subsequent component failing. The analysis of each subsequent stage is then added to the decision branch of the preceeding outcome. The analysis continues until each component in the process has been considered, its outcomes determined and added to the evolving tree structure. Probabilities of failure can then be attached to each outcome of the complete event tree. It is possible to determine the probability of success/failure at any given point in the process.

Figure 3.1.3.1 shows an event tree drawn for a parallel pump system employing two water pumps. The failure probabilities are included on the drawing as an example of the calculations.

The application of ETA to software is given the name Software Event Tree Analysis (SETA).

Figure 3.1.3.1 ETA Analysis of a Pumping System





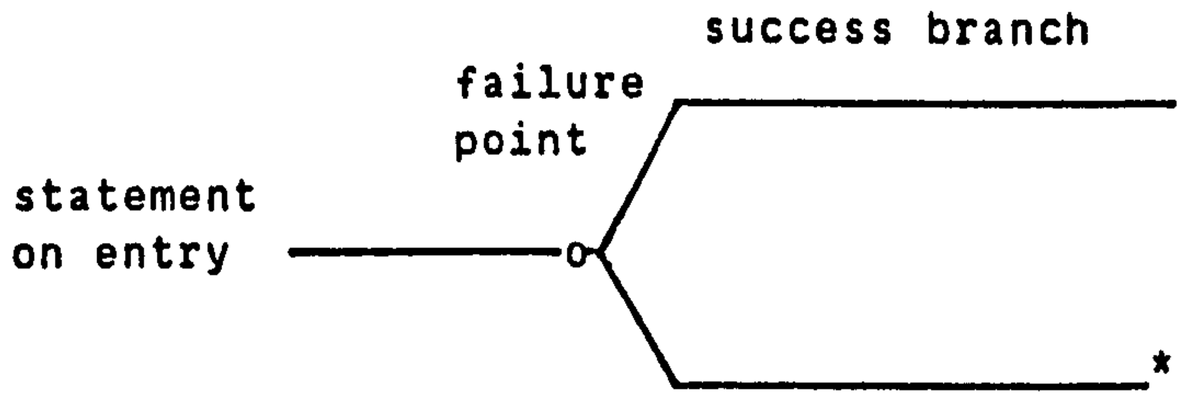
Each programming statement in a high-level language is executed according to a set of rules governing the logic of the statement, for instance the statement IF x THEN y ELSE z will execute y or z according to the logical condition of x. Further, the sequence in which the statements are executed is determined according to the logical relationship of one statement to another.

By convention the failure branch in an ETA diagram is drawn to the left and the success branch is drawn to the right. From a single entry to a complete program there are only two possible exits: success and failure. So for each statement within the program there are also two exits from a single entry. Within the statement the branching strategy continues to a lower level of detail but the respective exits are connected to maintain the higher strategy of the statement. The respective exits from the statements are connected in order to maintain the strategy of the complete program. Pascal-like programming statements represented in SETA format are shown in

Figure 3.1.3.2.

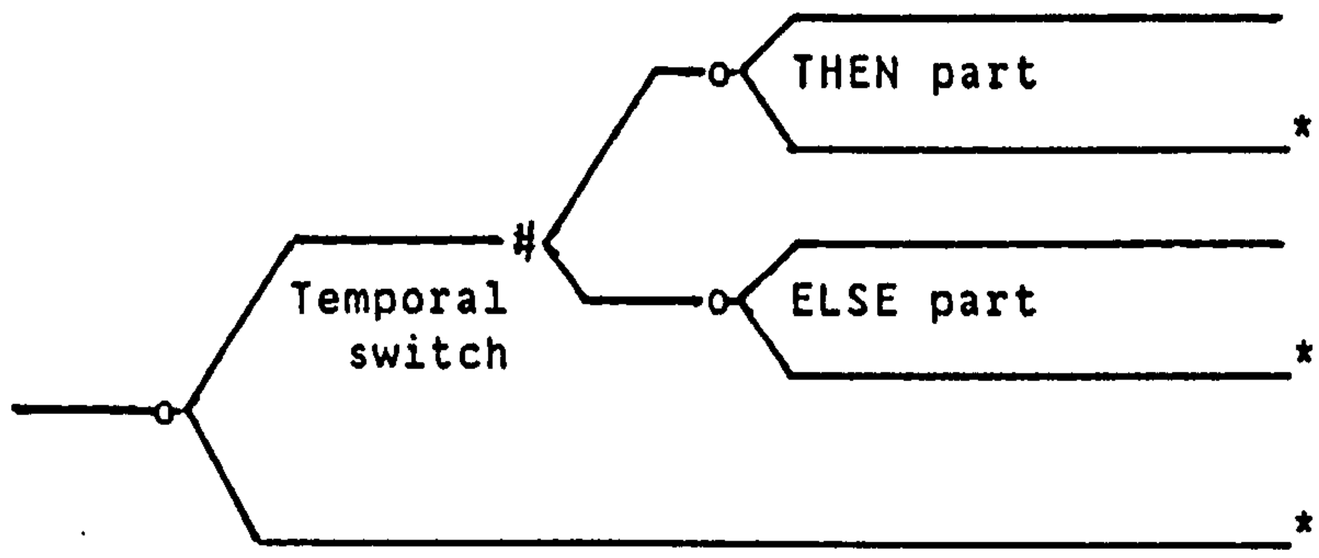
Figure 3.1.3.2 Program Statements using SETA

a) assignment

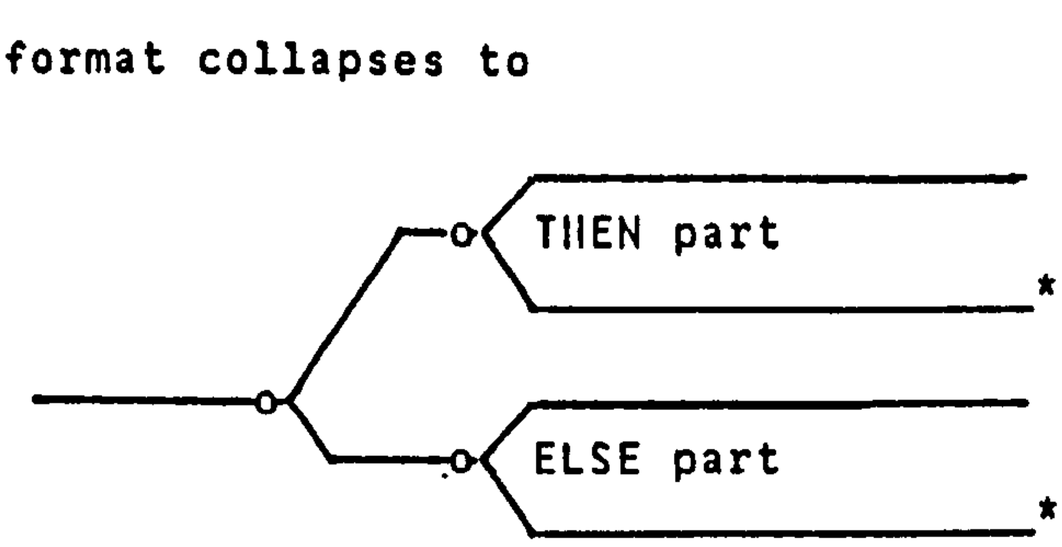


where the symbol '*' denotes a terminal failure which would cause an irrecoverable failure to exist. The failure branch can occur on other statements but has been labelled only on this one.

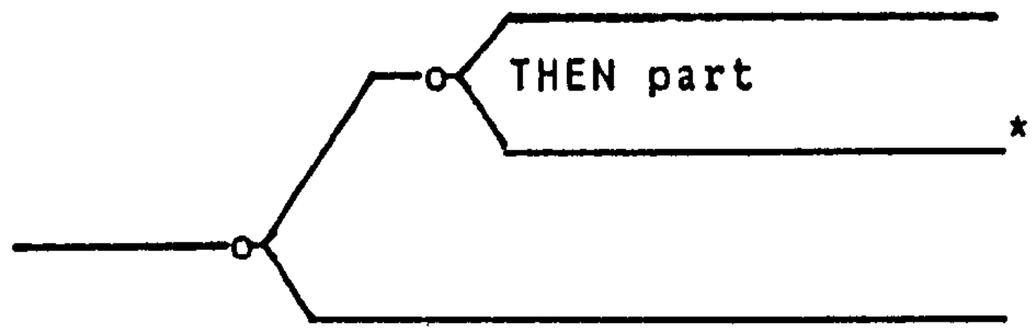
b) IF..THEN..ELSE..



N.B. The temporal switch, #, permits the flow to take whichever path is relevant according to the conditional expression assuming that it has not failed. Since it is the data flow that is the concern and not the control flow the format collapses to

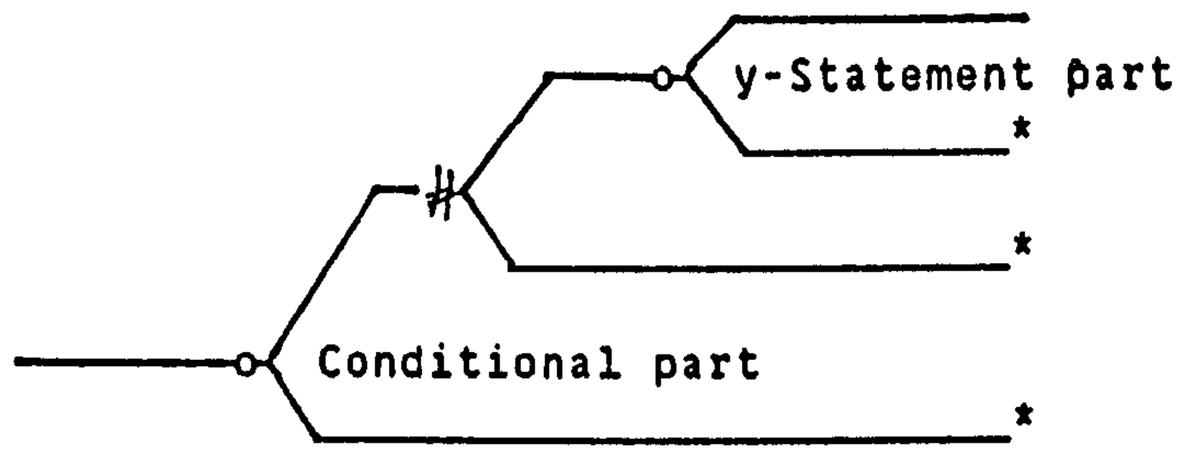


where the reduced statement, IF..THEN.., is used the diagram becomes

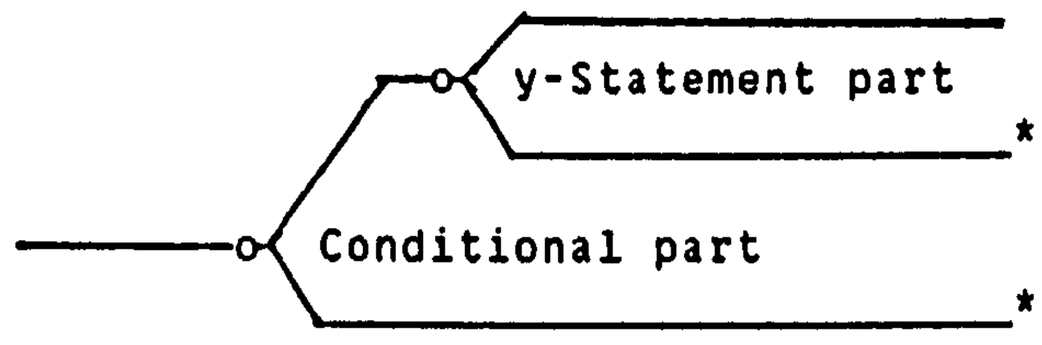


since the else-part is implied as being the following statement.

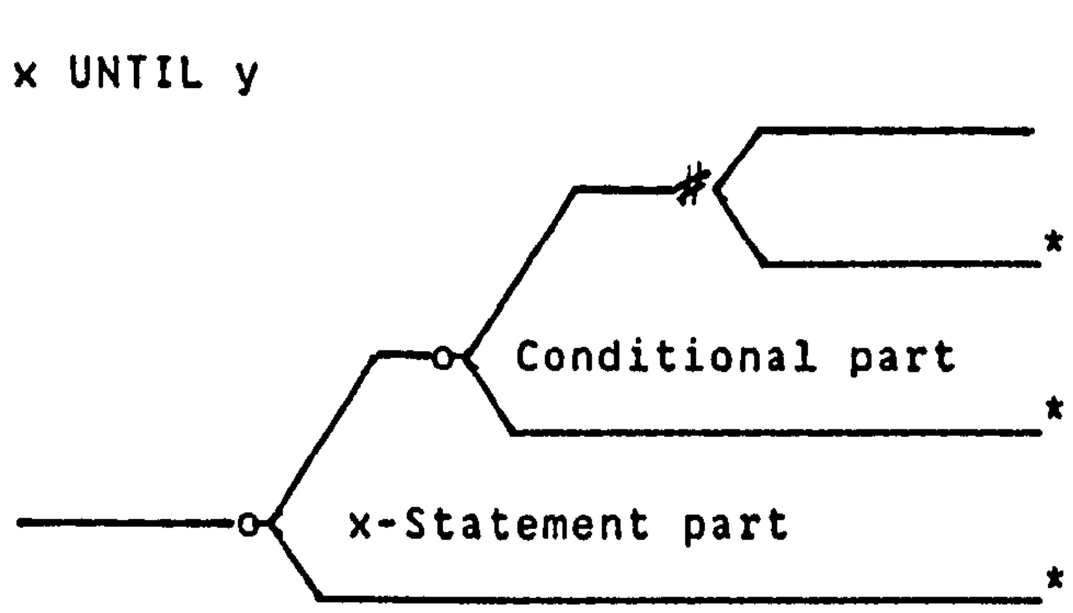
c) WHILE x DO y



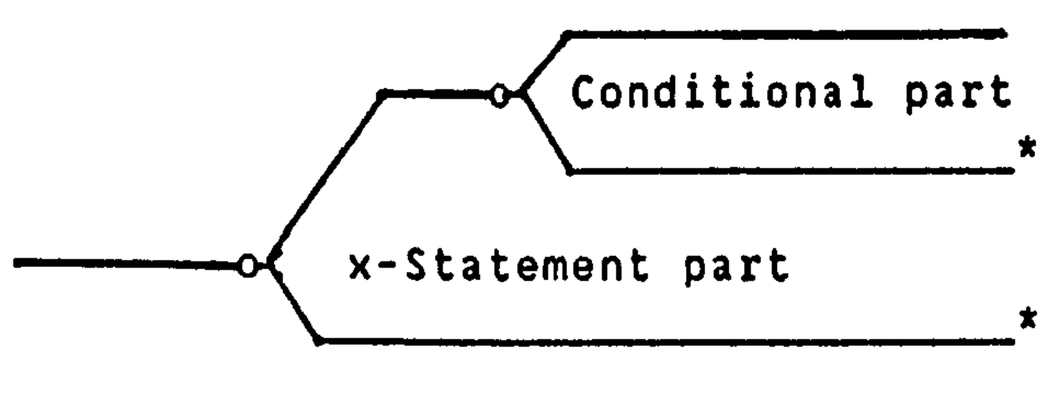
removal of the temporal switch causes the format to collapse to



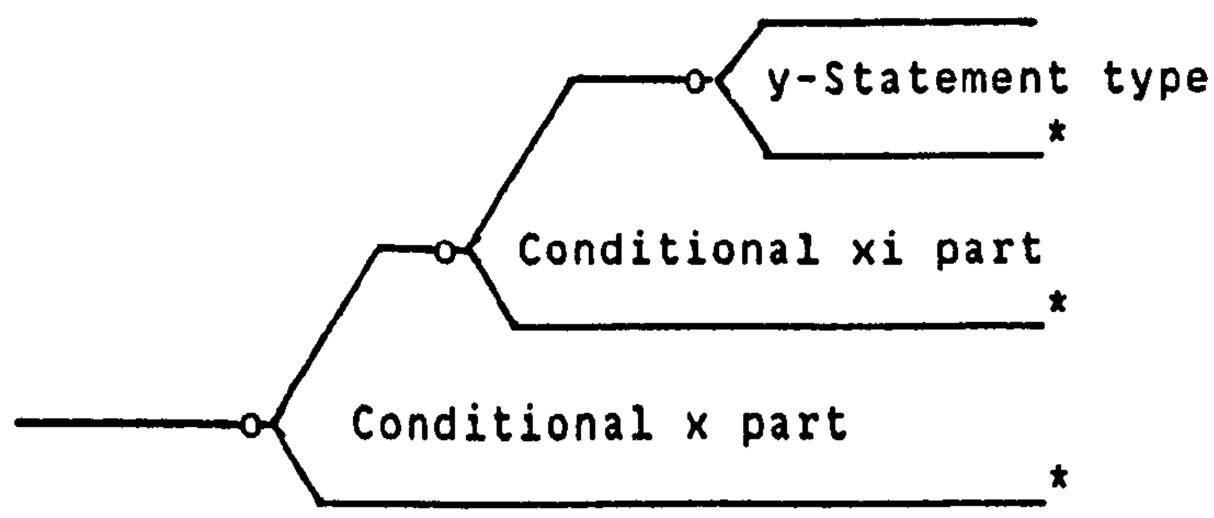
d) REPEAT x UNTIL y



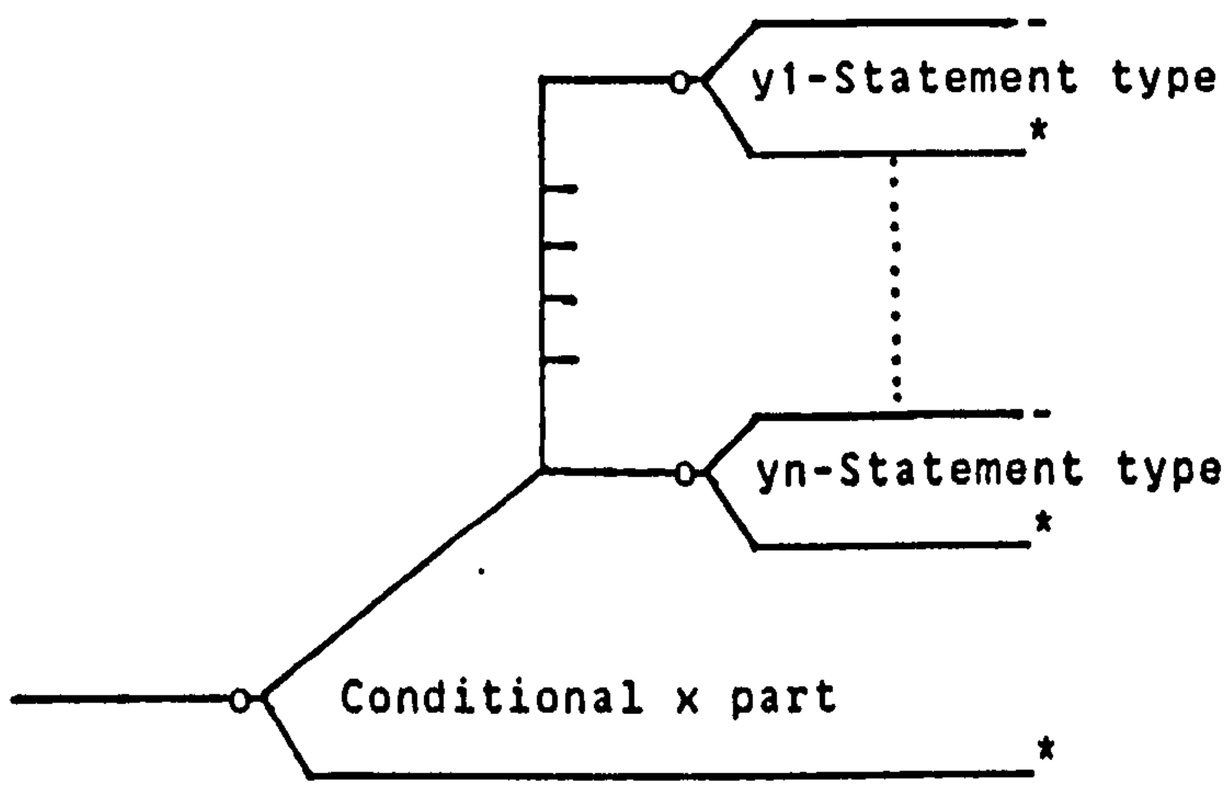
removal of the temporal switch causes the format to collapse to



e) FOR x TO x DO y
i



f) CASE x OF y1 .. yn



It is possible to apply SETA to program design languages in the same way that Leveson and Harvey have applied SFTA to design languages but the maximum benefit is to be gained by applying SETA to the source code, assuming that the compiler and other software development tools are dependable. This is the lowest level of abstraction needed for a meaningful representation of the program. SETA, like ETA, has the probabilities added to the diagrammatic representation and will be demonstrated by means of an example.

To show how SETA can be applied to a simple program consider the program below taken from Jensen and Wirth, [7].

```
PROGRAM fcount(input,output);
VAR ch:CHAR;
    count:ARRAY['a'..'z'] of INTEGER;
    letter:SET OF 'a'..'z';
BEGIN
     letter := ['a'..'z'];
     FOR ch := 'a' TO 'z' DO
          count[ch] := 0;
     WHILE NOT eof DO
     BEGIN
          WHILE NOT eoln DO
          BEGIN
               read(ch);
               write(ch);
               IF ch IN letter
                    THEN count[ch] := count[ch] + 1
          END;
     writeln;
     readln;
     END
END.
```

The declaration part of the program adds little to the

information flow of the program and is not included in the analysis. In the example that follows the omission of the declaration part presents the first line of the analysis as being an assignment statement.

The next statement is FOR ch := 'a' TO 'z' DO whose SETA statement format is added to the success branch of the preceeding assignment statement.

The analysis continues until the tree includes all the statements in the program. The structure is shown in Figure 3.1.3.3 with the tree orientated through 90 degrees. The success branch has been aligned vertically to prevent the tree tending towards the right.

Three significant items of information can now be deduced from this tree; those statements whose failure will cause a terminal failure, the probability of successful execution and the probability of particular terminal failures. To be able to extract information from the event tree the probability of successful and unsuccessful execution of each statement needs to be added to the tree as in Figure 3.1.3.4. To avoid presenting too much information at the expense of clarity the programming statement has been substituted by a probability of successful execution.

Figure 3.1.3.3 SETA of the Example Program

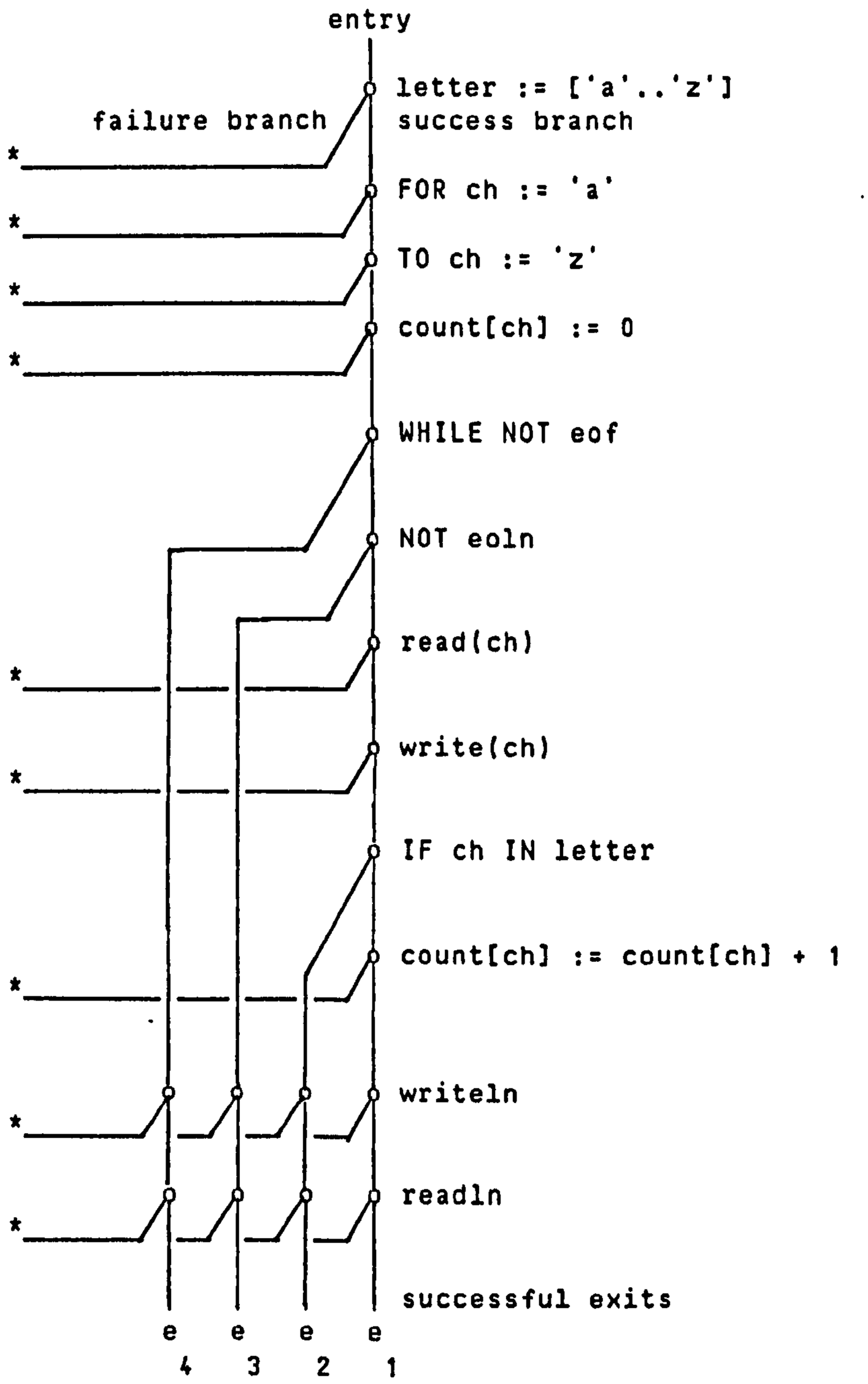
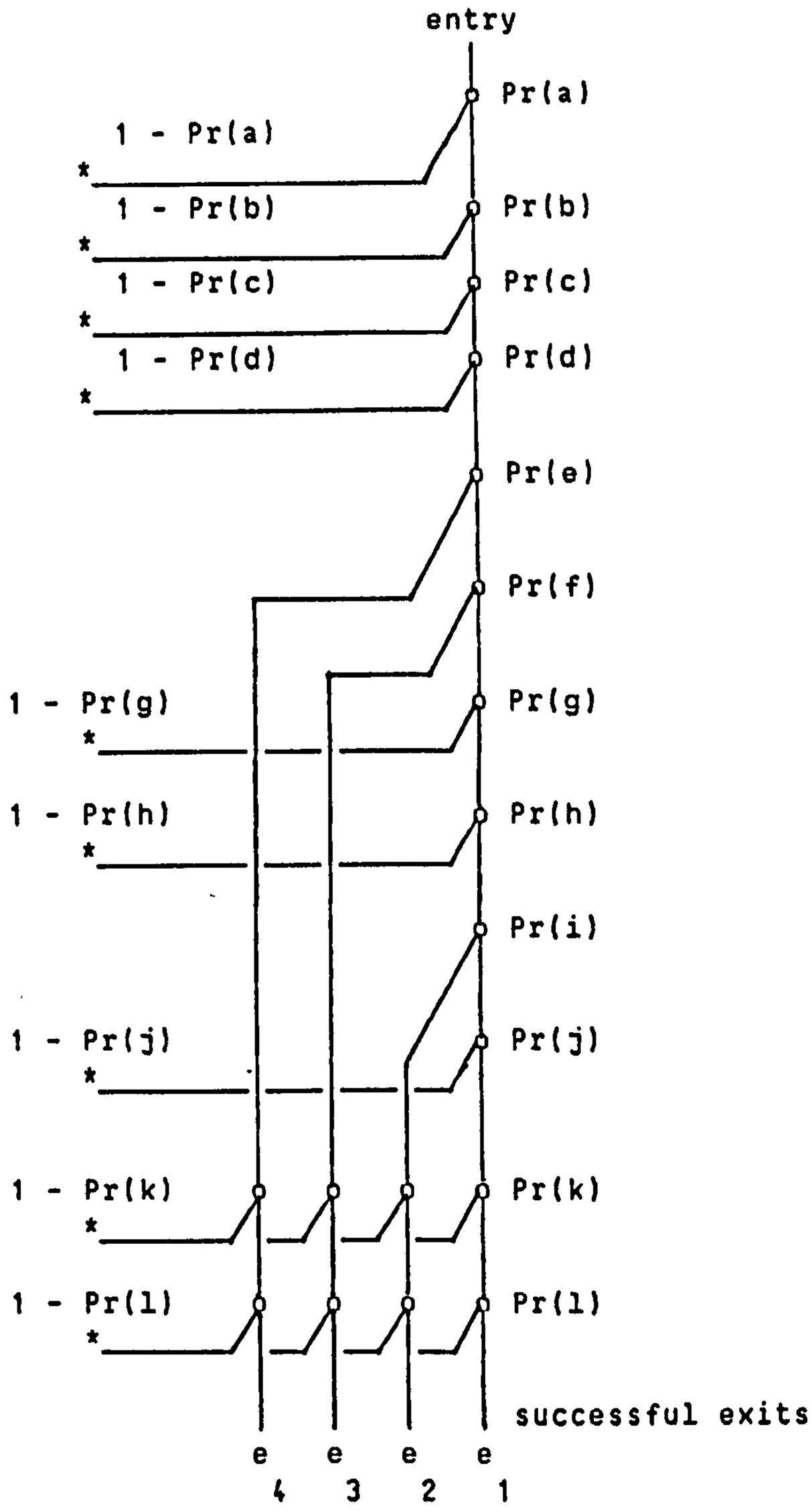


Figure 3.1.3.4 SETA of the Example Program with Probabilities Assigned



Each possible successful exit, ei, has an individual probability. Assuming s-independent events, the exit probabilities are

$$Pr(e1) = Pr(a).Pr(b).Pr(c).Pr(d).Pr(e).Pr(f).Pr(g).Pr(h).Pr(i).Pr(j).Pr(k).Pr(l)$$

$$Pr(e2) = Pr(a).Pr(b).Pr(c).Pr(d).Pr(e).Pr(f).Pr(g).Pr(h).(1 - Pr(i)).Pr(k).Pr(l)$$

$$Pr(e3) = Pr(a).Pr(b).Pr(c).Pr(d).Pr(e).(1 - Pr(f)).Pr(k).Pr(l)$$

$$Pr(e4) = Pr(a).Pr(b).Pr(c).Pr(d).(1 - Pr(e)).Pr(k).Pr(l)$$

There are four possible successful exits. The probability of a

successful exit from the program is the sum of individual probabilities and given as

$$Pr(S) = Pr(e1) + Pr(e2) + Pr(e3) + Pr(e4)$$

The probability of an unsuccessful exit is given as

$$Pr(F) = 1 - Pr(S)$$

$$= 1 - [\ Pr(e1) + Pr(e2) + Pr(e3) + Pr(e4)\ ]$$

As the number of statements increases so the probability of a successful exit is reduced. There are two issues to be considered;

1) the probability of failure of a statement is related to the syntactic and semantic complexity of that statement. The resulting probability of failure of the function being performed by that program statement is influenced by the programmers choice of statement. Therefore consideration has to be given to the trade-off between the number of statements and the probability of failure for particular statement types.

2) in Chapter 3.4 it was postulated that the probability of failure of a module is related to its length and that from a safety point of view a larger number of small modules is preferable to a small number of large modules. So a reduction in the length of a module will also influence the probability outcome.

3.1.4 Discussion

Leveson and Harvey, [8], observed that SFTA can be combined with FTA to provide a comprehensive analysis of a total system including hardware and software. The application of ETA to the hardware associated with a computer system can continue to a point where the software element needs to be considered. To consider the software, SETA can be used to provide a comprehensive analysis.

As an example of how ETA proceeds to the point where SETA can be used consider the case where plant sensors are used to pass data to a computer on the functioning of a critical plant area so that optimal control of the plant can be maintained. Using ETA the sensors, the instrumentation, the Analogue-Digital Converter and the computer input-output mechanisms are considered. However, once the analysis has reached the point where data is requested by the software making a request to the operating system, device driver or control software then SETA can be used. SETA can be used to assess the software in the context of programs or programming statements.

A complete ETA/SETA analysis is then possible to identify particular items of concern and to seek to reduce the probability of a failure. Assuming an item of concern can be described in terms suitable for analysis using SETA and that the risk is assessed to be such that further detailed analysis is necessary, then additional SFTA can be undertaken.

Summarising, the approach is to identify potential failures using ETA/SETA and then to further examine the concerns using SFTA.

The application of existing Event Tree Analysis (ETA) to software (SETA) is possible and provides useful information to the analyst on failure probabilities. By careful identification of the issues raised with SETA further analysis can be undertaken using what Leveson and Harvey have called Software Fault Tree Analysis (SFTA) in order to isolate the concerns. Once these concerns have been isolated then suitable remedial action can be taken to eradicate them.

3.2 The Use of State Transition Diagrams

The internal state of a process can be modelled in the abstract at any moment using graph theoretic methods such as the State Transition Diagram which is a special case of the Finite State Machine. The State Transition Diagram is commonly used by engineers to assess the behaviour of a system, whether that system is an industrial process or the internal function of a computer.

A Finite State Machine consists of a finite set of input symbols A, a finite set of internal states S, a finite set of output symbols Z, a next-state function f and an output function g. The machine M is denoted by M = {A,S,Z,f,g}. Additionally an initial state q0 may be included, when the machine M will be denoted by M = {A,S,Z,q0,f,g}.

An example Finite State Machine could be one with three input symbols, three internal states and three output symbols as

$$A = \{a,b,c\}$$

$$S = \{q0,q1,q2\}$$

$$Z = \{x,y,z\}$$

the next-state function f could be defined as

$f(q0,a) = q1$ $f(q1,a) = q2$ $f(q2,a) = q0$

$f(q0,b) = q2$ $f(q1,b) = q1$ $f(q2,b) = q1$

$f(q0,c) = q0$ $f(q1,c) = q0$ $f(q2,c) = q2$

the output function g could be defined as

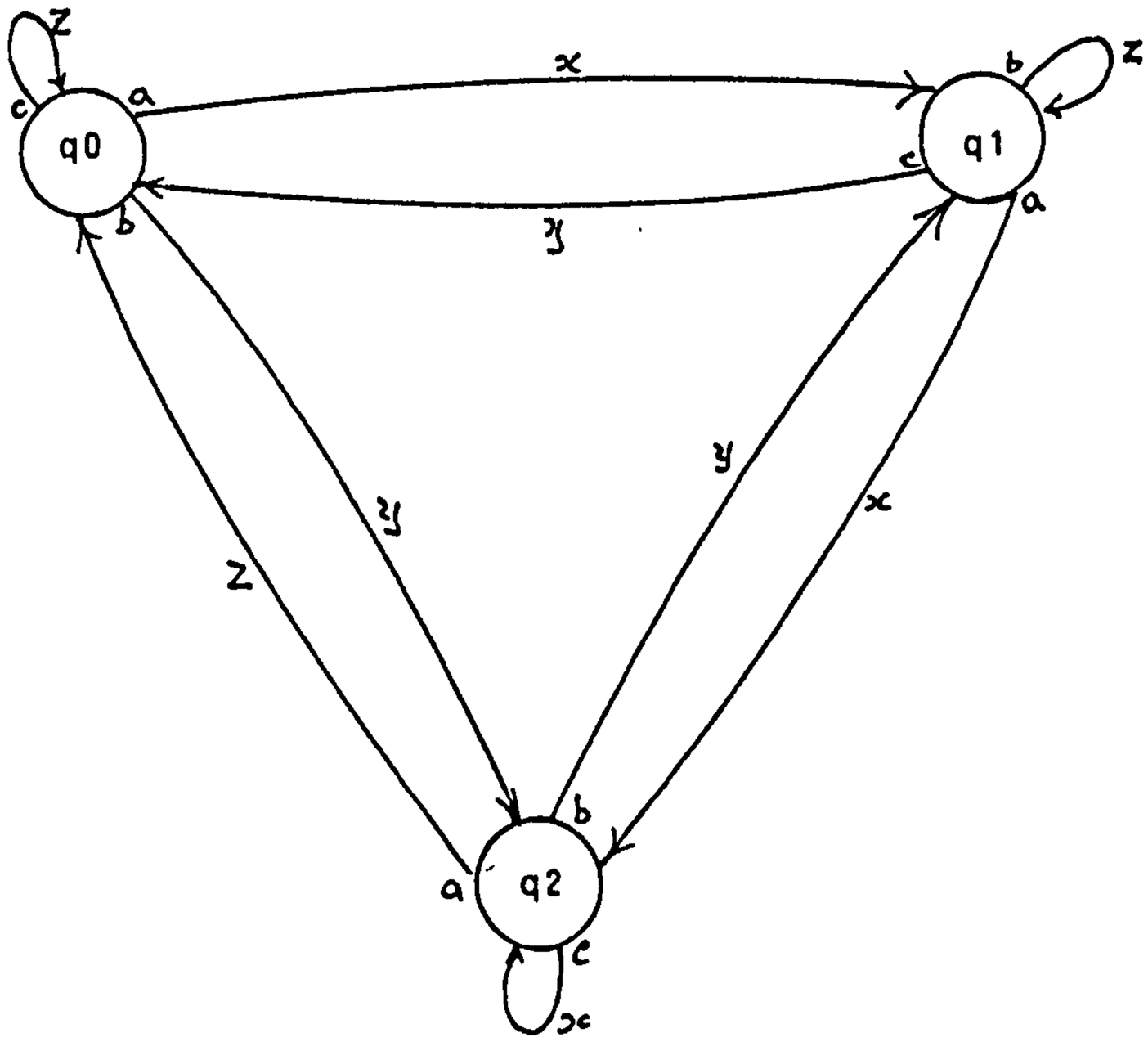
$g(q0,a) = x$ $g(q1,a) = x$ $g(q2,a) = z$

$g(q0,b) = y$ $g(q1,b) = z$ $g(q2,b) = y$

$g(q0,c) = z$ $g(q1,c) = y$ $g(q2,c) = x$

A state diagram is one way of representing the machine M. A state diagram is a labelled directed graph with the vertices being the states S of M such that an arc can be drawn between state q0 and q1 and labelled with the pair a,x representing the next-state function $f(q0,a) = q1$ and the output function $g(q0,a) = x$.

Figure 3.2.0.1 The State Diagram for the Example Machine M.



Another way of representing machine M is to use a state table which tabulates the next-state and output for each combination of current state and input. A state table for machine M would be

Current State	Input a	Input b	Input c
q0	q1,x	q2,y	q0,z
q1	q2,x	q1,z	q0,y
q2	q0,z	q1,y	q1,x

A State Transition Diagram consists of a set of states S, a set of events E and a transition function, t.

The state transitions for Finite State Machine M can be represented as

S = {q0,q1,q2}

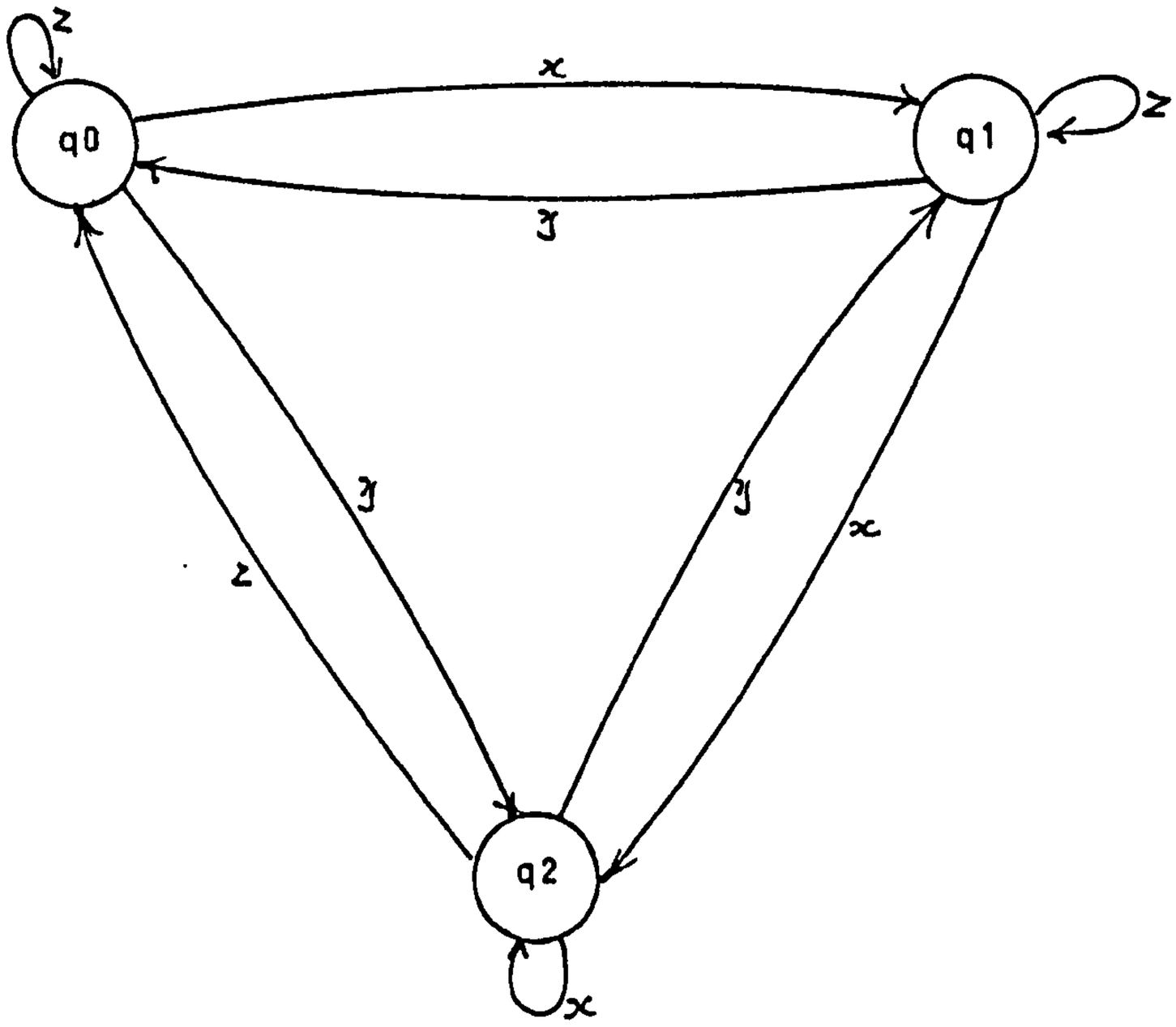
E = {x,y,z}

and the transition functions are

t(q0,x) = q1	t(q1,x) = q2	t(q2,x) = q2
t(q0,y) = q2	t(q1,y) = q0	t(q2,y) = q1
t(q0,z) = q0	t(q1,z) = q1	t(q2,z) = q0

the State Transition Diagram would be that shown in Figure 3.2.0.2.

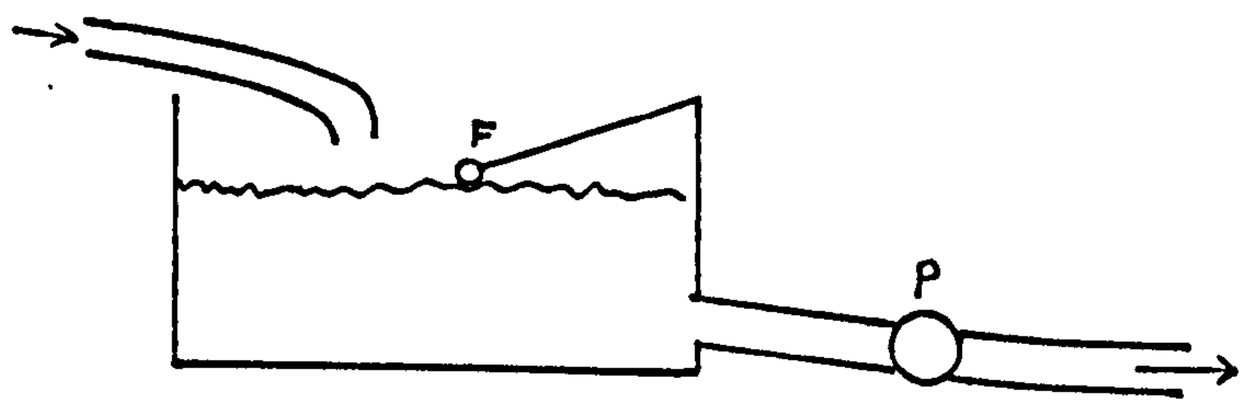
Figure 3.2.0.2 State Transition Diagram for the Example Machine



Industrial control systems can be modelled using both Finite State Machines (FSM) and State Transition Diagrams (STD) but the use of STD is more common.

As an example of the use of STD, take a simple control system consisting of a fluid pump, P, under the control of a fluid level float, F, whose aim it is to maintain the level of a liquid within a certain vessel by turning the pump 'on' to lower the level of the liquid when the level is indicated as 'high' by the float. Assuming that the liquid flow into the vessel is constant and not under the control of the system being modelled, the process scheme is shown in Figure 3.2.0.3.

Figure 3.2.0.3. Example Process



The objective of the control system is to ensure that none of

the liquid flows over the top of the vessel. If any liquid flows over the top of the vessel the condition is considered to be a catastrophic event.

To keep the model simple it is assumed that both the float F and the pump P work correctly even though there is a probability that the control signals may not. It is also assumed that no such failure of control signals exist.

The pump P is switched on when the float F is indicating 'high' and the pump is switched off when the float indicates 'low'.

The set of states S are

$$S = \{q0,q1,q2,q3\}$$

where q0 = level low, pump off

q1 = level low, pump on

q2 = level high, pump off

q3 = level high, pump on

the set of events E are

$$E= \{a,b,c,d\}$$

where a = Float high

b = Float low

c = Pump on

d = Pump off

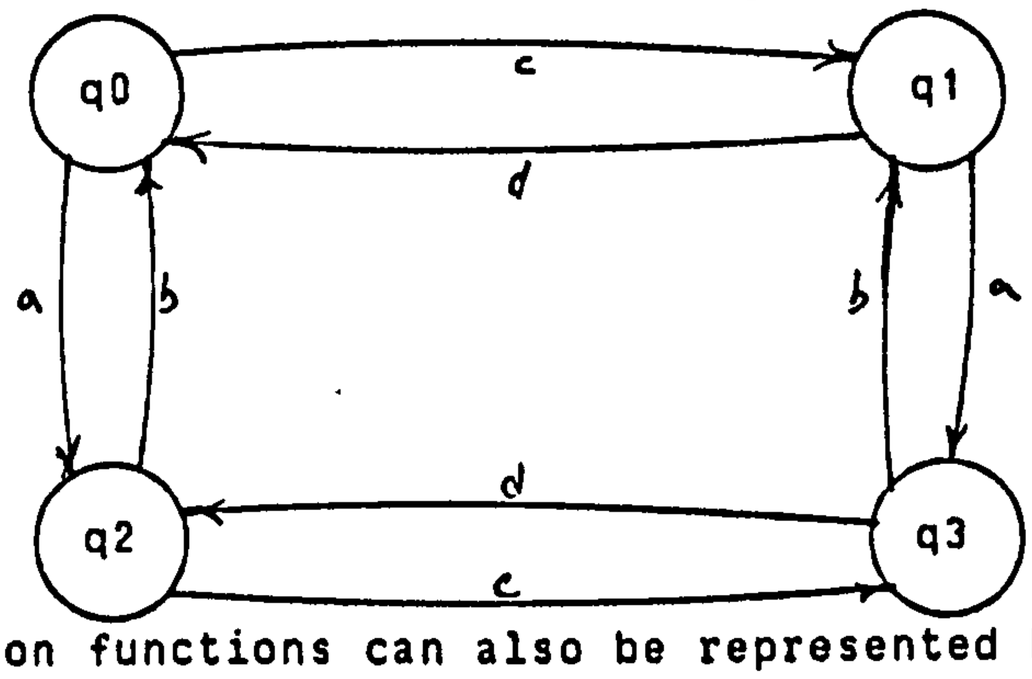
and the transitions functions t are

$$t(q0,a) = q2 \quad t(q1,a) = q3 \quad t(q2,b) = q0 \quad t(q3,b) = q1$$

$$t(q0,c) = q1 \quad t(q1,d) = q0 \quad t(q2,c) = q3 \quad t(q3,d) = q2$$

The state transition diagram for the control system is shown in Figure 3.2.0.4. With the transition T being T = {S,E,t} and the initial state q0 being included to give T = {S,E,q0,t}.

Figure 3.2.0.4 State Transition Diagram for the Example Control System



The transition functions can also be represented by a table called a transition table. Such a transition table for the control system being considered is

State	Float	Pump
q0	low (0)	off (0)
q1	low (0)	on (1)
q2	high (1)	off (0)
q3	high (1)	on (1)

State Transition Diagrams are deterministic and exhaustive. To demonstrate the exhaustive nature of STDs consider the control system to have been extended to ignore transient inputs from the float by requiring the float to indicate high for two successive observations before switching the pump on. The control algorithm is expressed as

$$Po = (Fs \wedge Fi \wedge \text{NOT } Poi) \vee (Fs \vee Fi) \vee Poi$$

where Po is the pump output value according to the logic

Poi is the initial or currently stores value for the pump output.

Fs is the stored value for the float

Fi is the input value for the float

assuming no errors experienced the transition table becomes

Poi	fs	fi	Po
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

In this thesis the concern is the identification of catastrophic failures/conditions and such a condition could arise in the control system if the level was high and the pump failed to operate causing the liquid to overflow. From the transition diagram and the transition table a catastrophic failure condition can be seen not to occur when all the equipment functions correctly.

So far the concern has been with representing the state transitions when all the equipment is working correctly and with no errors. When the control system has the same control algorithm but uses an industrial controller incorporating software to implement that algorithm then a catastrophic failure/condition can arise due to the failure of components of the controller, even though the electro-mechanical equipment may work correctly.

The transition tables for the control system using software considers three error types: stuck at 0, stuck at 1 and inversion. The conditions underlined are those which are considered to satisfy the criteria of a catastrophic failure/condition; fluid flowing into the vessel, float fluid level high and pump not on.

1) Error caused by Po being inverted when stored in Ps

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	0	1	0	0	0	1	0	0	0	1
0	0	0	0	0	1	1	1	1	0	1	1	1	0
0	0	1	1	0	1	0	0	1	0	0	0	0	1
0	0	1	1	0	1	1	1	1	0	1	1	1	0
0	1	0	0	0	1	0	0	0	1	0	0	0	1
0	1	0	0	0	1	1	1	1	0	1	1	1	0
0	1	1	1	0	1	0	0	1	0	0	0	0	1
0	1	1	1	0	1	1	1	1	0	1	1	1	0
1	0	0	0	0	1	0	0	0	1	0	0	0	1
1	0	0	0	0	1	1	1	1	0	1	1	1	0
1	0	1	1	1	0	0	0	0	1	0	0	0	1
1	0	1	1	1	0	1	1	1	0	1	1	1	0
1	1	0	0	0	1	0	0	0	1	0	0	0	1
1	1	0	0	0	1	1	1	1	0	1	1	1	0
1	1	1	1	1	0	0	0	0	1	0	0	0	1
1	1	1	1	1	0	1	1	1	0	1	1	1	0

2) Error caused by Ps being stuck at 1 (on)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	0	1	0	0	0	1	0	0	0	1
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	1	1	0	1	0	0	1	1	0	0	0	1
0	0	1	1	0	1	1	1	1	1	1	1	1	1
0	1	0	0	0	1	0	0	0	1	0	0	0	1
0	1	0	0	0	1	1	1	1	1	1	1	1	1
0	1	1	1	0	1	0	0	1	1	0	0	0	1
0	1	1	1	0	1	1	1	1	1	1	1	1	1
1	0	0	0	0	1	0	0	0	1	0	0	0	1
1	0	0	0	0	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	0	0	1	1	0	0	0	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	0	1	0	0	0	1	0	0	0	1
1	1	0	0	0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	1	1	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

3) Error caused by Ps being stuck at 0 (off)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	1	1	1	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	0	0	1	1	1	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	1	1	1	0	1	1	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	0	0	1	1	1	0
1	0	1	1	1	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1	1	0	1	1	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1	0	0	1	1	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	1	1	1	0	1	1	1	0

4) Error caused by Po being inverted on output, Ps is true value of Po

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	1	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	1	1	1	0	1	1	0	1
0	0	1	1	1	0	0	0	1	0	0	0	1	0
0	0	1	1	1	0	1	1	0	1	1	1	0	1
0	1	0	0	1	0	0	0	1	0	0	0	1	0
0	1	0	0	1	0	1	1	1	0	1	1	0	1
0	1	1	1	0	1	0	0	0	1	0	0	1	0
0	1	1	1	0	1	1	1	0	1	1	1	0	1
1	0	0	0	1	0	0	0	1	0	0	0	1	0
1	0	0	0	1	0	1	1	1	0	1	1	0	1
1	0	1	1	0	1	0	0	0	1	0	0	1	0
1	0	1	1	0	1	1	1	0	1	1	1	0	1
1	1	0	0	0	1	0	0	1	0	0	0	1	0
1	1	0	0	0	1	1	1	0	1	1	1	0	1
1	1	1	1	0	1	0	0	0	1	0	0	1	0
1	1	1	1	0	1	1	1	0	1	1	1	0	1

5) Error caused by Po being stuck at 1 (on)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	1	1	0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	0	0	1	1	0	0	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	0	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0	1	1	0	0	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	0	0	1	1	0	0	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	1	1	0	0	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

6) Error caused by Po being stuck at 0 (off)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	1	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	0	0	1	1	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	0	0	1	1	0	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	1	1	0	0	1	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	0	0	1	1	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	1	1	0	0	1	1	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1	0	0	1	1	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1	0	0	1	1	0	0

7) Error caused by Fi being inverted when stored in Fs

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	1	0	0	0	1	0	0	0	1	0	0
0	0	0	1	0	0	1	0	1	1	1	0	1	1
0	0	1	0	0	0	0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0	0	0	1	0	0	0
0	1	0	1	0	0	0	1	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1	1	1	0	1	1
0	1	1	0	1	1	0	1	0	0	0	1	0	0
0	1	1	0	1	1	1	0	1	1	1	0	1	1
1	0	0	1	0	0	0	1	0	0	0	1	0	0
1	0	0	1	0	0	1	0	1	1	1	0	1	1
1	0	1	0	1	1	0	1	0	0	0	1	0	0
1	0	1	0	1	1	1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1	1	1	0	1	1	1
1	1	0	1	1	1	1	0	1	1	1	0	1	1
1	1	1	0	1	1	0	1	0	0	0	1	0	0
1	1	1	0	1	1	1	0	1	1	1	0	1	1

8) Error caused by Fs being stuck at 1 (high)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	1	0	0	0	1	0	0	0	1	0	0
0	0	0	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	1	1	1	1	1	1
0	1	0	1	0	0	0	1	0	0	0	1	0	0
0	1	0	1	0	0	1	1	1	1	1	1	1	1
0	1	1	1	0	0	0	1	0	0	0	1	0	0
0	1	1	1	0	0	1	1	1	1	1	1	1	1
1	0	0	1	0	0	0	1	0	0	0	1	0	0
1	0	0	1	0	0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	0	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	0	0	0	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

9) Error caused by Fs being stuck at 0 (low)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	0	0	0	1	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	0	0-	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	1	0	0	0
1	0	1	0	1	1	0	0	0	0	0	0	0	0
1	0	1	0	1	1	1	0	1	1	1	0	1	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	0	0	0	1	0	0	0
1	1	1	0	1	1	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	0	1	1	1	0	1	1

10) Error caused by Fi being inverted on input

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0	1	1	0	0	0	0
0	1	0	0	0	0	1	1	0	0	1	1	1	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	0	0	1	1	0	0	0	0
1	0	0	0	0	0	1	1	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	1	1	0	0	0	0
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	0	0	0	0	0	0	0	0

11) Error caused by Fi being stuck at 1 (high)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

12) Error caused by Fi being stuck at 0 (low)

Poi	fsi	fi1	fs1	Po1	Ps1	fi2	fs2	Po2	Ps2	fi3	fs3	Po3	Ps3
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0

N.B. All occasions are potential catastrophes.

The example process modelled for the discussion has been a trivial control system and yet many error conditions have been identified. If the control system was more complex with many more parameters to consider then the dimensions of the state diagram would become unmanageable. The number of instances where an error can exist and create a catastrophic failure/condition becomes proportionately greater as the number of parameters increases making the use of state transition diagrams difficult to use for isolating potential hazards.

In many industrial control systems the number of states would be so great that exhaustive checking of all conditions would not be practicable.

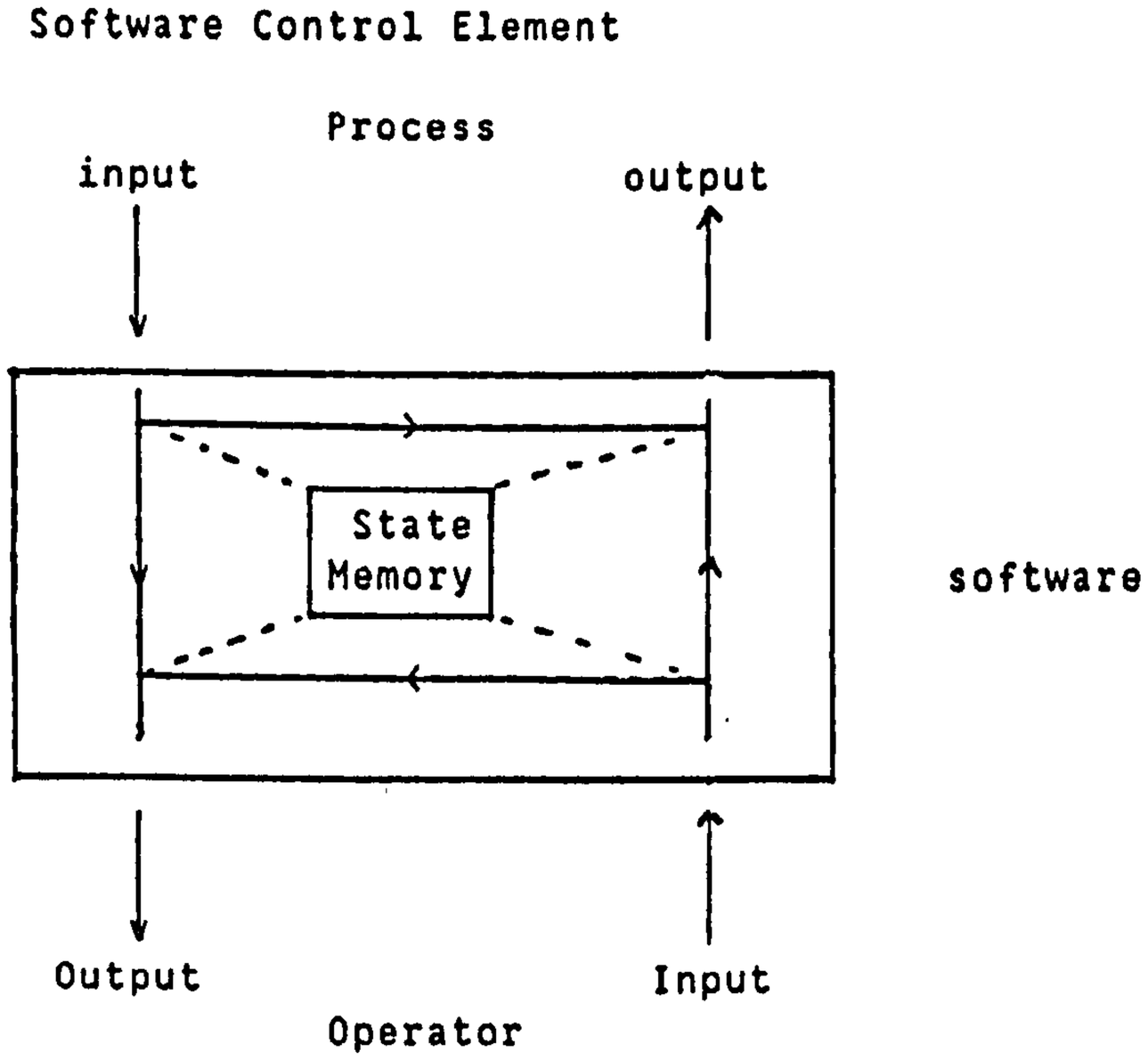
3.3 Categorisation of Dangers

Examination of the risk arising from the use of software in an industrial process control system requires the dangers to be categorised. An argument for three categories of danger called minor, Major and Serious is presented.

3.3.1 The Software Control Element

The flow of information through a control system is dependent on the control strategy adopted for that industrial process. A general structure for various routes that the information can take through the software element of the control system, depending on whether it is a fully automatic control system, a system with manual intervention or a simple data logger is shown in Figure 3.3.1.1. Each route through the software has its own unique function and potential for error. Each of the points at which an error can occur are called 'error points' and assigned a number.

Figure 3.3.1.1 Software Control Element



The Health & Safety Executive, [6] p.3, has suggested that there are three typical modes of operation:

"Mode 1

The computer receives signals from the plant or machine to

which it is linked and then processes this information and transmits or displays it. The computer does not send control signals to the plant or machine. The operator controls the plant without recourse to the computer except for information, and thus retains the power of both decision and control.

Mode 2

The computer acts as a link between the person, who is monitoring the process, and the control elements (e.g. valves or contactors). This role may involve the feedback of signals from the plant or machine to the computer but the computer's scope for plant alteration is limited essentially to carrying out the instructions of the person in control of the process. In this mode, therefore, the decision is made by the person but control is exercised by the computer.

Mode 3

The computer, without human intervention, makes significant changes to, or puts significant restrictions on, the plant or machine operating conditions in accordance with its program. The computer therefore retains the power of both decision and control".

Because of the reduction in the cost of automation and the economic pressure for more industrial efficiency there is a tendency to make greater use of industrial controllers operating in Mode 3.

Principally, there are two ways that an Industrial Controller can reduce the safety of the process it controls or create a dangerous condition; by abnormal operation of the program or by aberrant behaviour of the controller. In all modes there will be occasions when the Industrial Controller can exhibit aberrant

behaviour and produce dangerous situations.

3.3.2 Potential for Errors

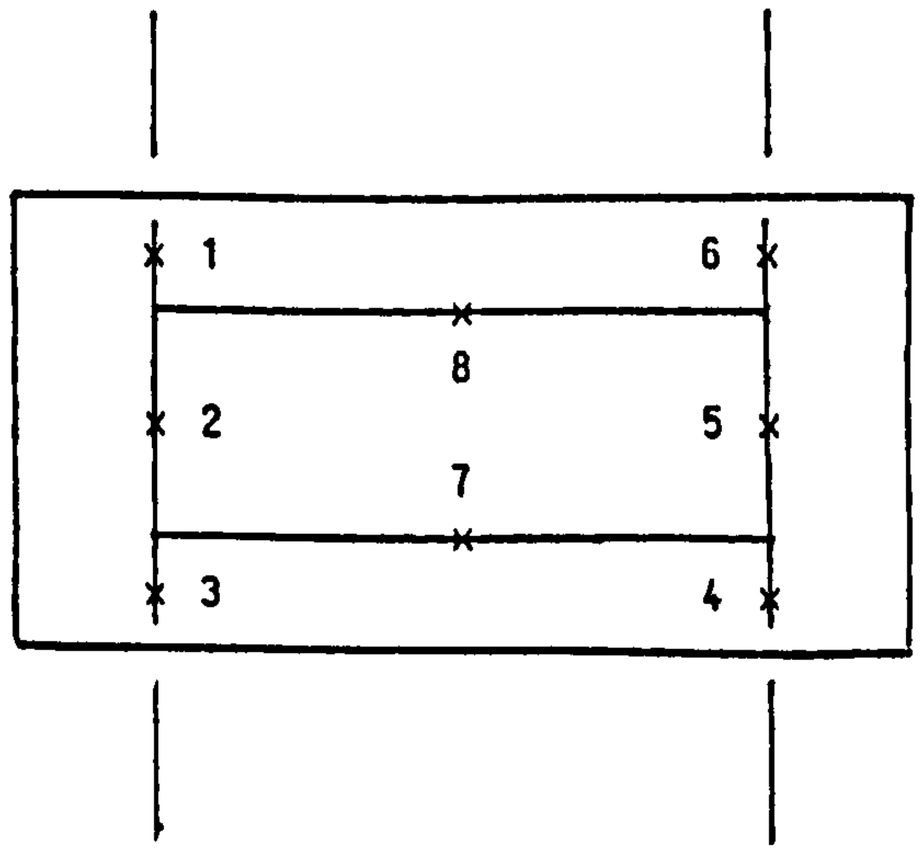
Mode 1 operation, data logging, would cause process inputs to enter the software and pass via the state memory to the Operator output port which may have a computer monitor attached for use by the Operator. There will be no response from the Operator entered to the software, in response to the output.

Mode 2 operation would be as Mode 1 but in addition the Operator responses would be input to the software and pass to the process outputs via the state memory. The Operator responses would be reflected back to the Operator via the operator output port and the computer monitor.

Mode 3 operation would be as Mode 1 but instead of the response coming from the Operator, as in Mode 2, the input will be routed to some decision making procedure which will effect the response through the process outputs. Knowledge of the response may only be available to the Operator by observing the process status displayed on a computer monitor.

From Figure 3.3.1.1 eight points of potential error can be identified, called 'Error Points'. These error points are shown in Figure 3.3.2.1.

Figure 3.3.2.1 Error Points



Where single extreme errors can occur and their effect will now be examined. It is assumed that extreme errors are those where the information contained in the data is completely wrong in a permanent way and not transient, which have special characteristics.

Error point 1 would cause the information on the process state to be corrupted. The corruption of data on input would cause erroneous information to be presented to the Operator and to the decision module. Information available to the Operator and any decision module would not represent the true process state.

At error point 2, an error in the input data which was correct on entering the software, would be corrupted. The effect of the error would be to cause the Operator to be misinformed on the process state. Any subsequent action by the Operator would be correctly conveyed to the process. Since the process state is incorrectly displayed the Operator would have some indication that an error had occurred from the observable plant status.

At error point 3, the introduction of an error would cause the process state information and the Operator input commands to be incorrectly displayed to the Operator. The Operator would be alerted to the error by noting the error shown in his input commands and also by monitoring the process state displayed compared with that observed.

If an error occurred at error point 4 the Operator input commands would be corrupted causing the wrong actions to be conveyed to the process and the commands displayed to the Operator would also be corrupted. The Operator would only become aware of an error by monitoring the response of the process state and monitoring his reflected commands. In a slow industrial process the risk would be limited by the Operators actions. In a fast industrial process the risk would be greater.

At error point 5, the existence of an error would cause commands from the Operator to be incorrectly interpreted by the control software and as a consequence convey the wrong actions to the process. Error point 5 has a greater risk than that of error points 2 & 7. In a process which is not time-critical the error would not cause an increase in the risk since the process state would still be displayed correctly. But the error is more dangerous in a time-critical process. Also the Operator is able to compare his commands with the resulting process reaction, which corresponds to an unexpected plant state being corrected by the feedback control mechanism.

A potentially great risk exists when an error occurs at error point 6; the Operator input commands or the decision module commands are corrupted on output. In either case the wrong actions are conveyed to the process. The result may be a situation with a high risk, without the Operator being aware of the danger.

Error point 7 has the potential to incorrectly display the commands of the Operator but the error will be identified as an error by the Operator noting the disturbance.

Error point 8 has a potentially great risk when the control strategy permits control actions to be taken directly by the software. The actions may also be monitored by the Operator, if there is one. An error at error point 8 in the software would cause the decision module to issue wrong commands which, though founded on correct process inputs, would then pass to the process undetected.

3.3.3 Categories of Danger

From the discussion above the effects of errors existing at various points in the software have been proposed. Whilst all errors have some effect there are some which present a much greater risk than others. Therefore, some weighting needs to be applied to

isolate the error and to place it in the appropriate category. The weighting used here is a subjective assessment of the degree of danger resulting from the occurrence of that error.

Three categories of danger resulting from errors in software used in control applications have been distinguished as;

minor - errors which are undesirable and inconsistent with the specification but do not cause a hazardous condition to exist. For instance, mis-spelled warning messages and file corruption.

Major - errors which cause a hazardous condition to exist but which allow correction by an Operator. For example, failure to check correct outputs by re-input, output action differing from that commanded and reported, corruption of command with resulting incorrect action (input or output). The effects of errors in this category are observable by the Operators.

Serious - errors which cause a high level of risk to exist; erroneous output on a fast or time-critical process, overriding of protection mechanisms like watch-dogs, uninformed bridging of safety checks, corrupted limit checks, wrong logical deduction from inputs resulting in a wrong output.

The category of minor is placed on a set of errors which, though undesirable and inconsistent with the specification, do not cause a hazard to materialise. As an example, consider an error in a module whose function is to log data. An error in the module might cause the correct output message "alarm 99" to be displayed incorrectly as "alm 99", where the number indicates an alarm number and not a sequence number. Such an error might cause the Operator to suspect a fault in the software but would not prevent him from understanding

the message. The ability to understand the corrupted word in a message is due to the message having in-built redundancy allowing the message 'alm' to be recognizable as 'alarm'. Message Redundancy is also known as the 'richness' of the language. The context of the message is contained in a descriptor which contains information. However, if the corruption had been that the alarm point "99" was corrupted to "9" then it is possible that the Operator would not recognise the correct message from this limited information.

Alternatively, had the Operator input the command "Open valve 6" which was corrupted to "Open valve 4" by the command input module then the error would no longer be in the minor category since the intended message cannot be determined. If the corruption was such that the erroneous command was displayed as "Open valve 4" and also effected the action on valve 6, then the Operator would be aware of the error and react accordingly. There is little redundancy in the message since the valve has been identified by a single character and not a descriptor containing more information. Therefore the message is considered to be unsafe. Errors of this type have been put in the set of errors called the Major category of errors and refers to a set of errors that cause a hazard to exist but which are not too great for the Operator to correct.

Taking the above example of the Operator inputting the command "Open valve 6", if an error occurs in the process output module and corrupts the command to "Open valve 4" then the error is in the set of errors called the Serious category of errors. The error is in the Serious category of error because the command will have been correctly displayed to the Operator, who now expects an action, but the output to the process is not as commanded; the Operator may be unaware of the potentially dangerous situation for some time, by which time a disaster may have occurred. The Serious category of

error refers to a group of errors which present a high level of risk.

Only single permanent failures have been considered so far but it is possible for there to be combinations of permanent errors and transient errors. Both types of error have severe implications to the safe working of the system.

Combinations of errors are many and varied. The consequence of combinations of errors is that individual single permanent errors may be masked by the accompanying permanent error and create a confused view of the problem. The category of danger for a combination of errors is the category of the higher single permanent error included in the combination. For example, a combination of a permanent minor error and a permanent Major error is considered to be a Major error.

Transient errors, however, present an error condition which may be short lived and infrequent. The consequence of which may be that an unsafe condition applies for the duration of the error and it is improbable that the error will be isolated immediately. The full effect of a transient error cannot be appreciated until the transient error is identified and safety requires that maximum caution should be exercised where uncertainty exists. A transient error is placed in the category of Serious until such time as the error is isolated. Due to transient errors being in the Serious category a combination made of permanent and transient errors is considered to be in the Serious category.

3.4 The Structuring of Software Modules for Safety

Software modules are discrete units of computer programming collectively providing a sphere of influence within a system. The software modules can be structured, or configured, in many different ways to achieve the same sphere of influence. The term "sphere of influence" refers to the extent to which the actions of a specific module are influential within a system and is not limited to first-order effects. The structuring of the system affects the amount of confidence the designer is justified in vesting in the system. Using robust programming techniques, such as N-Version Programming and Recovery Blocks, influences the safe execution of a program.

Some of the structural options available to software designers are considered in this Chapter and it is postulated that the use of a structuring technique called 'Safety Modules' improves the safe operation of control modules without an increase in either the run-time resources or the complication of the system.

3.4.1 N-Version Programming

Hardware fault tolerant systems commonly use a strategy called N-Modular Redundancy (NMR), involving an odd number, say three or five, redundant versions of the same hardware with a voting system. N-Version Programming is a software implementation of the NMR strategy for hardware and was first proposed by Chen and Avizienis, [5].

In N-Version Programming a number of similar programs, N, are written to perform identical functions using different programming techniques to perform the same function or using different source languages. To add diversity the programs may be written by different teams of programmers, even in different locations.

In N-Version programming structuring of the system is such that the N-versions of the program are usually placed under the control of

a driver program within the run-time environment. The driver program invokes each version of the program, awaits completion of the respective execution, compares the results and takes action accordingly.

The driver program synchronises the execution of the versions and maintains a record of those versions which take longer time to execute. Once the versions have all reached completion and have been synchronised then a voting mechanism compares the respective results. If it is not possible for all the versions to return the same result then 'inexact voting' is used when small discrepancies in the results are tolerated. In industrial systems the accumulation of such discrepancies, accumulated over a period of operation, cannot be disregarded as the error may become too severe to permit safe operation. Therefore N-Version Programming cannot be recommended in safety-related systems.

3.4.2 Software Fault Tolerance

Errors in the program itself can demonstrate the characteristic of having 'failed' in many ways; suspect inputs, inadequate inter-program communication, hardware malfunctions or loss of synchronisation with other programs with which it corresponds. For a system to continue operation whilst overcoming these 'faults' a technique known as fault tolerance is required.

One fault-tolerant technique is Recovery Blocks, [1] and [11]. Recovery block design makes use of one or more redundant programs in addition to the original program. The original program is called a 'primary block' and is tested for failure by an 'acceptance block'. On detecting a failure the acceptance block will cause one or more of the redundant blocks, called 'alternate blocks', to be executed until either all 'alternate blocks' have failed or one has functioned correctly.

The recovery block strategy is

1) the primary block is executed

2) the acceptance block tests for a satisfactory result

3) if the result is acceptable then the next primary block in the sequence is executed. If the result is not acceptable then the system is said to 'recover' to a point where the system state is restored to that existing before the failed primary block was executed and one of the alternate blocks is executed,

4) the execution of alternate blocks is repeated until an acceptable result is achieved,

5) if an acceptable result cannot be achieved then the system is said to have failed.

To implement the recovery block strategy requires two special procedures;

RECOVER - which keeps account of whether it is the primary block or one of the alternate blocks being executed and maintains a copy of the state of the system prior to the block being entered,

ACCEPTANCE - performs the acceptance test and causes a system recovery if a failure is detected. The procedure also has a record of whether it is the primary block or an alternate block being executed.

In a multi-processing environment where shared data is used to pass data between programs it is possible for there to be an overlap when two or more competing blocks are recovering. Randell, [11], called this situation the 'domino effect' and observed that whilst only one block may have failed the failure of more blocks may be indicated thus causing a system shut-down. To limit the domino effect additional facilities need to be provided, [1], which further

increases the amount of resources committed to the strategy.

Whilst the Recovery Block strategy is simple in its concept the implementation is more involved. It does allow alternate control strategies to be attempted on detection of a failure. The major disadvantages in using the recovery block strategy for industrial control systems are;

- the difficulty of restoring the system to a known state without causing a 'domino effect' where alternate blocks force other blocks to restore
- the time taken to restore the system to a known state may mean that the restored system state no longer reflects the current plant state
- considerable resources are required to implement multiple copies of the primary block
- if the system is safety-related then the personnel maintaining the system operationally need to be made especially aware of the nuances of such a strategy.

These difficulties could create a situation where the system actively seeks to restore itself without maintaining a safe plant status.

3.4.3 Safety Modules

If, due to the increased probability of an undetected error being present, it is assumed that the probability of failure of a program module is related to the number of characters forming the program then the probability of failure of a system is similarly related to the structure existing between the modules. In the case where the module effects control over some critical item of plant it is desirable to maintain a low failure probability which suggests that the module lengths need to be correspondingly short.

There are two ways of reducing the length of a module; dividing the module still further into a number of sub-modules or reducing the

length, usually by using advanced programming techniques.

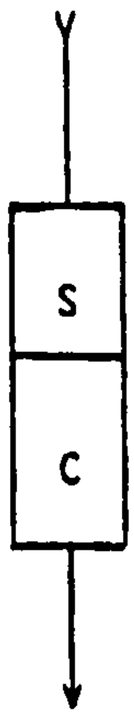
A satisfactory division of the roles of the original module is normally possible without a consequent increase in the complexity of the software other than in the interconnection coupling between modules. The outcome of the division of the module is that low probabilities of failure can be achieved for individual sub-modules and the software retains a simple internal structure which allows the sub-modules to be understood. The internal simplicity is important to allow changes to the function of the sub-module to be effected without disturbance to any safety checks in the module.

Length reduction using advanced programming techniques has an immediate disadvantage in the resulting program becoming so esoteric that only the originating programmer is able to fully understand its function which in turn means that it is only the originating programmer who can safely make changes arising out of testing. Such practices are undesirable from many points of view. Most significantly, from a safety view, is that the safety checks within the program may have been installed by the programmer and these can be unintentionally by-passed when changes are made by another programmer who is unfamiliar with the program.

Since modularisation of the software does not substitute convenience for safety, the principle of module sub-division is to be preferred to length reduction.

A control module will probably have safety checks built into the software. In which case the structure could conceptually be as in Figure 3.4.3.1 with the control part of the module intimately co-operating with the safety part of the module.

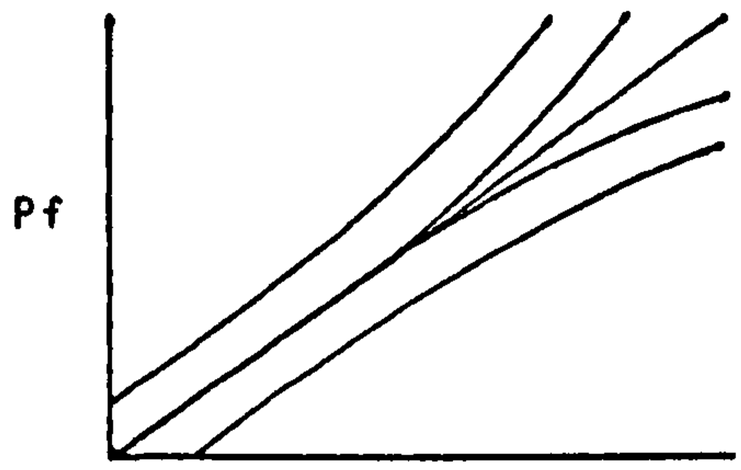
Figure 3.4.3.1 Safety and Control Software Integrated



where S = Safety Part

C = Control Part

It is assumed that the probability of failure of a sphere of influence, P(F), is related to the length of the modules. The relationship between the length of a module and its probability of failure may be exponential, linear or differential or any of the relationships below;



It is assumed in this thesis that as the length of the module increases it is more probable that errors will be introduced and that the relationship exhibits an exponential characteristic.

If the modules are structured as in Figure 3.4.3.1 then the failure probability for such a structure is

$$P(F) = P(S) + P(C)$$

where P(S) and P(C) are the failure probabilities for the Safety Part and Control Part, respectively and P(F) is the probability of failure.

If the safety part is seperated from the control part into a seperate Safety Module whose primary role is to ensure that the Control Module continues to function safely there will be distinct

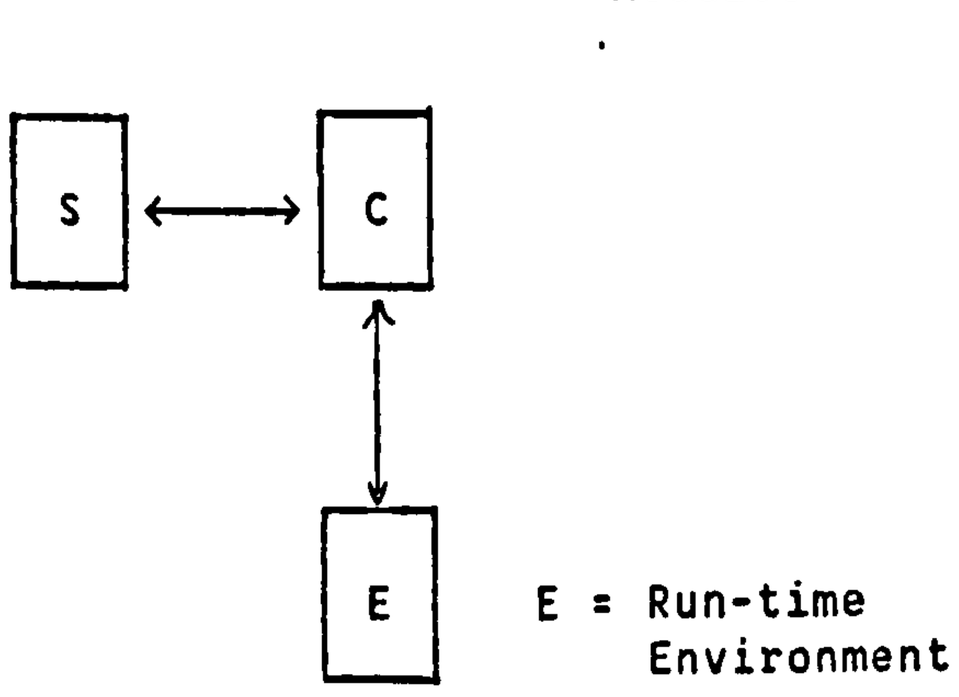
flows of data between them. The flow of data is considered to be between each module and the run-time environment of the computer, with channels to each.

Communication between the safety module and the control module may be such that before control of the plant is effected by the control module the safety module will check that the action is reasonable given the plant status. The safety module may have exclusive access to data concerning the operation of the item of plant it is concerned with, for example equipment design limits and rates of change of plant parameters. Plant data could be stored in a read only file. The control module may have access to a limited sub-set of plant data in order for it to be able to perform all the logical and mathematical functions necessary to maintain control.

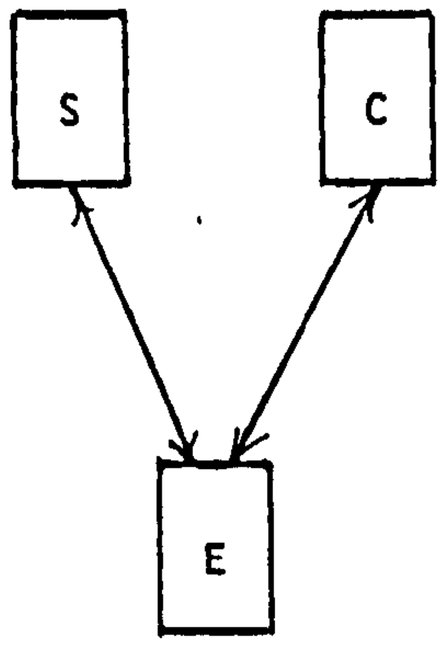
The control can be effected either by the control module or the safety module. If the action is taken by the control module as in Figure 3.4.3.2 a), b) and c) then there exists a probability that the action approved by the safety module will be corrupted in some way before being effected. Also a probability exists that corruption of the control action may occur if it is effected by the safety module, Figure 3.4.3.2 d) and e), though it is probable that the safety module will detect the corruption and take the necessary corrective action. Therefore the risk of an unsafe control action being effected is lower when the action is undertaken by the safety module.

Figure 3.4.3.2 Configuration of Safety and Control Modules

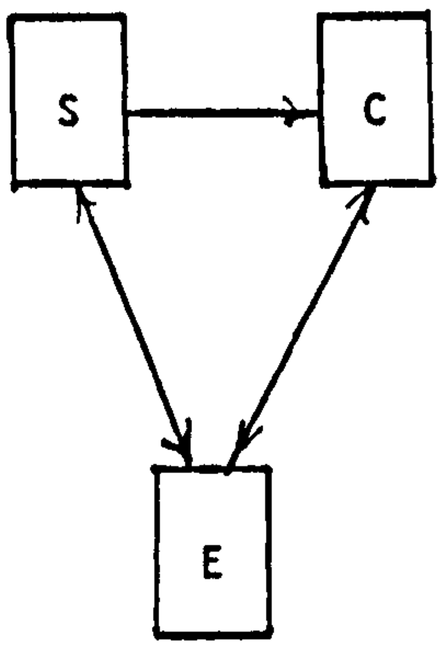
a)



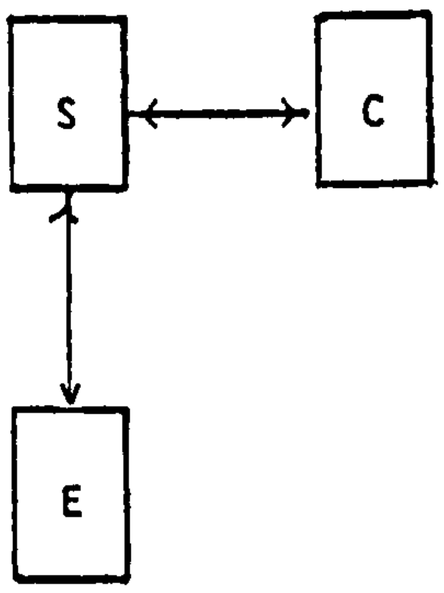
b)



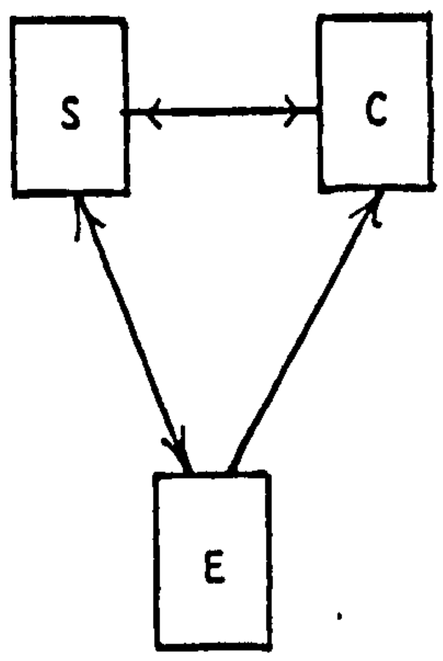
c)



d)



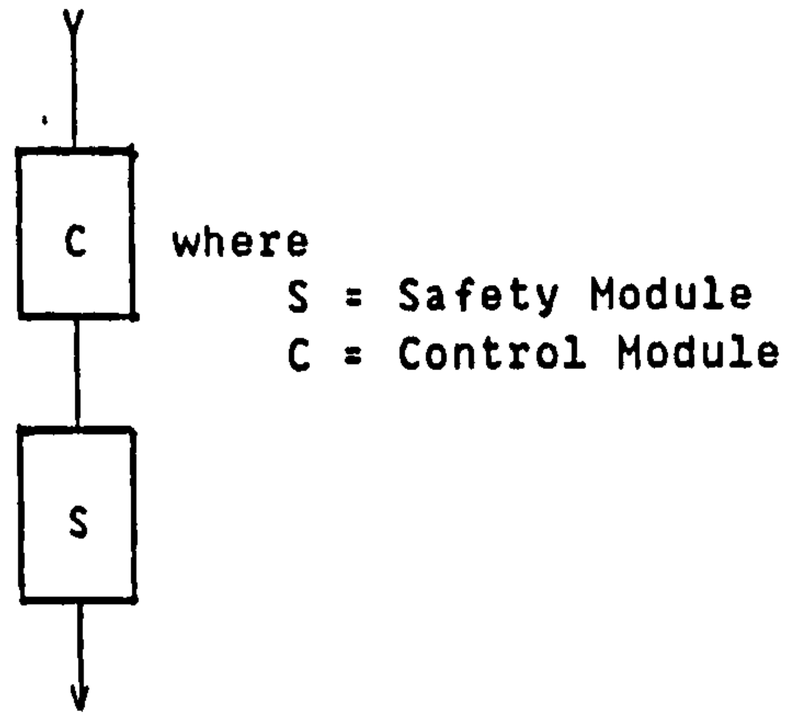
e)



When the two modules are seperated in the way discussed they can be structured to operate either sequentially or concurrently.

The conceptual structure for the Safety and Control modules operating sequentially is shown in Figure 3.4.3.3.

Figure 3.4.3.3 Safety and Control Modules Operating Sequentially

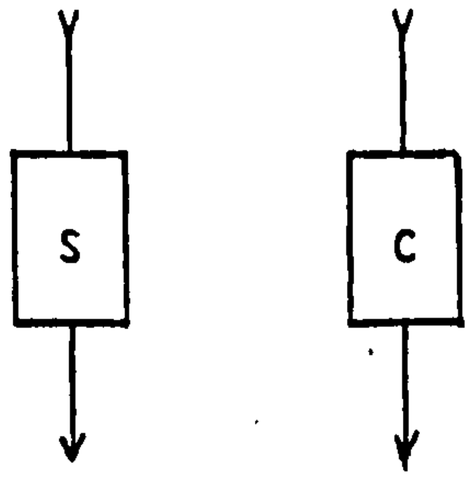


The operation of the modules involves the two modules functioning in a serial manner. It is assumed that the probability of failure is related to length such that L(C) and L(S) combined gives L(C) + L(S) then the probability of failure is given as

$$P(F) = P(C) + P(S)$$

Providing the run-time operating system orders the synchronisation of tasks, the safety module can also be configured to execute concurrently with the control module, Figure 3.4.3.4, and maintain a safe operation with respect to the control module through the linking mechanism. Since the run-time environment is required to schedule both modules the probability of failure of each module also needs to include the effect of the run-time environment on the outcome.

Figure 3.4.3.4 Safety and Control Modules Operating in Parallel



Each module can fail to execute its role in distinct ways; abnormal execution as a result of the other module, non-execution as a result of the other module, corrupt data as a result of the other

module and the run-time environment, corrupt instruction as a result of the other module, failure to communicate as a result of the other module and the run-time environment.

The module can fail as a result of any of these independent reasons. The probability of a failure in this configuration is

$$P_{FS} = [\, P_{Sp}(Lc) + P_{Sn}(Lc) + P_{Scd}(Lc,E) + P_{Sci}(Lc) + P_{Sfc}(Lc,E) \,]$$

$$P_{FC} = [\, P_{Cp}(Ls) + P_{Cn}(Ls) + P_{Ccd}(Ls,E) + P_{Cci}(Ls) + P_{Cfc}(Ls,E) \,]$$

where

P_{FS}, P_{FC} = prob. of failure of the Safety/Control Module

P_{Sp}, P_{Cp} = failure of safety/control module

P_{Sn}, P_{Cn} = non-execution of safety/control module

P_{Scd}, P_{Ccd} = corrupt data of safety/control module

P_{Sci}, P_{Cci} = corrupt instruction of safety/control module

P_{Sfc}, P_{Sfc} = communication failure of safety/control module

Ls = length of safety module in characters

Lc = length of control module in characters

E = run-time environment

The failure of the safety module can be caused by a failure of the control module and prejudice safety by allowing unjustified freedom of action to the control module. Therefore a mechanism is required to maintain the safe operation of the safety module. The paradox is not new and was noted almost 2000 years ago in the phrase

"Sed quis custodiet ipsos custodes?"
Juvenal, 'Satires' c60-130 A.D.

The probability of failure of the safety module can be reduced by placing a restriction on the ability of the control modules to

corrupt either data or instructions when communicating with the safety module. For the safety module to effect the actions requested by the control module it is necessary for the control information to be made available to the safety module through some secure communication mechanism such as parameter passing or sharing of data space. If parameter passing is used then there is a probability that errors will be induced by the run-time environment, which can itself cause data corruption. The option of using shared data space is subject to a lower probability of error because of the linking procedures used within the compiling system for declaring global data references.

A module whose sole function is to maintain an ultimate safe working condition by monitoring the safety modules within a system needs to be inviolate and must be allowed to make some judgement on the safety modules operational capability. The module would have a connection to the run-time environment but not with any other item of software. Connection with the run-time environment is exclusively for the purpose of checking that the version of the safety module to be executed by the run-time environment has not changed in any way from that considered to be safe when the module was first made operational, or that the execution of the safety module is not overdue in time with respect to the previous instance. If changes have been made to the safety module which is now considered to be 'suspect' or it is considered to be overdue the ultimate safety module will inform the responsible plant authority of the suspicion and effect a predefined safe control operation on that plant area.

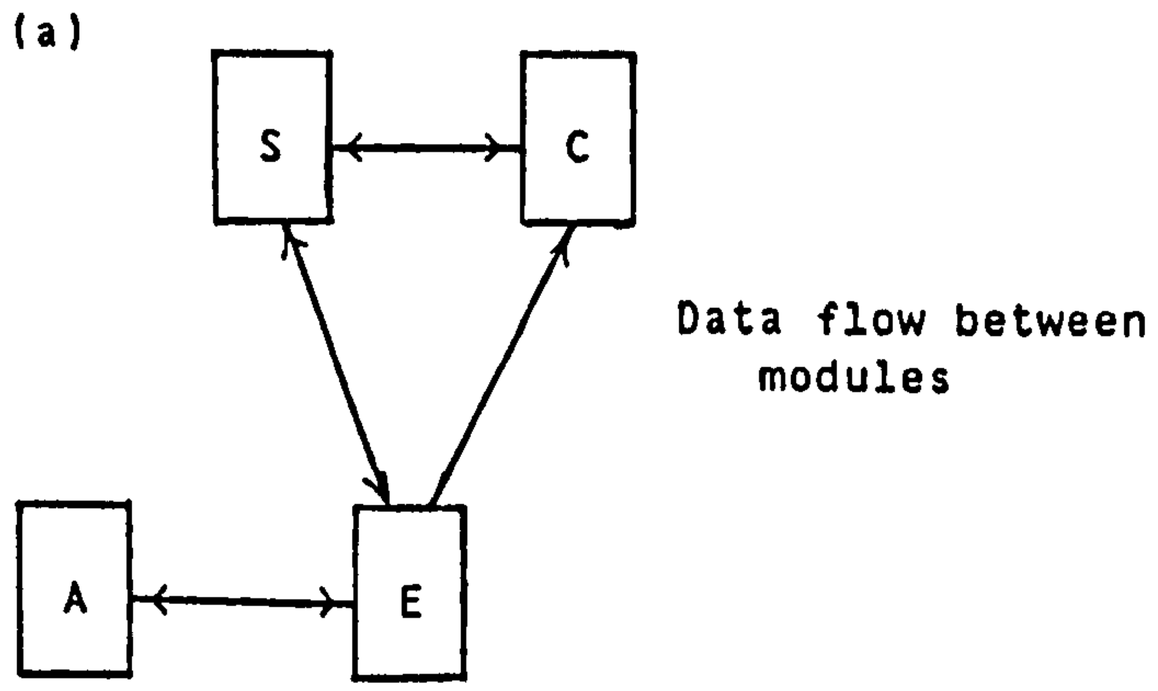
Changes to the safety module can come about by another module causing corruption to the safety module or by functional changes to the safety module requested by the plant authority. Functional changes to safety modules have a probability that the implications to

safe working of such changes may not be appreciated by those making the changes.

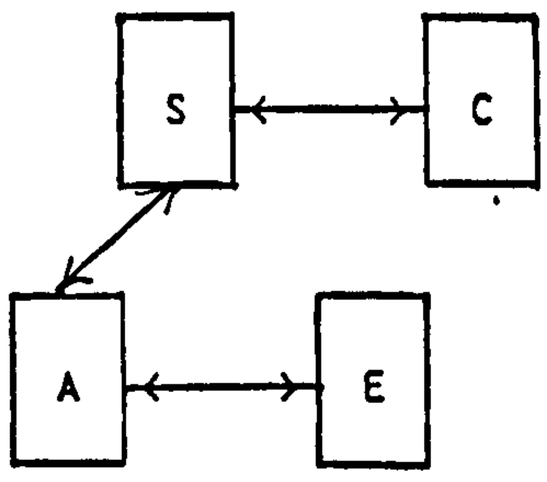
In most industrial plants it is a proscribed activity for an Engineer to override an ultimate safety limit without permission being granted on the authority of the Plant Engineer. Such authority may take the form of the possession, by Authorised Engineers, of the necessary key to physically unlock the safety protection system surrounding the limit. The safety limits proposed for the safety modules should be regarded in the same way. Access permission to the ultimate safety module should be restricted by managerial action of, say, the Plant Engineer. Such an ultimate safety module is called the Arbitrator Module.

To maintain the inviolate nature of the Arbitrator Module it could be located in a Read Only part of the main memory of the computer. The Arbitrator Module could use a strategy of checking the unique identity of a safety module in order to monitor the safe working of the safety modules. Conceptually the Arbitrator Module can be viewed as in Figure 3.4.3.5.

Figure 3.4.3.5 The Arbitrator Module



(c)



Functional relationship between modules

where A = Arbitrator Module
S = Safety Module
C = Control Module
E = Run-Time Environment

3.4.4 A Mechanism for Ensuring the Integrity of Software

The role of the Arbitrator Module and its relationship with the Safety Modules have been discussed on the assumption that the modules have not been corrupted as a result of software errors, incorrectly installed modifications to the system or deliberate sabotage. If the system has been corrupted in some way it cannot be said to be complete. The Oxford English Dictionary defines completeness as a synonym of integrity. It is in the context of completeness that the word integrity is used in this thesis. A mechanism to restrict the probability of corruption not being detected is called an Integrity Lock.

When the system is put into operational use it is reasonable for the Functional Authority to assume that all the modules are considered to be safe. If at this point a unique identity is given to each module such that safe operation is only possible when the identity is shown to be valid, then a strict regime of managerial control can be exercised on the installation of any changes to the system.

To create a unique identity some form of encryption based on the run-time code of the module can be used. A similar requirement is found in data communication systems where a unique code, such as a cyclic redundancy check or Hamming code, is generated to assist the receiver in determining whether an erroneous message has been received. The unique identity may be corrupted by a single error or

by multiple errors. Hamming codes have been developed to cater for at least one error and so could be the immediate choice for creating the unique identity. The creation of the unique identity ought to be done under the strictest controls, for instance under the authority of a Senior Engineer, to maintain security. The unique identity could be generated by a module called the Security Module.

The Security Module needs to be capable of reading the particular control or safety module as an ordered set of characters forming a message and generating the identity according to a specified algorithm. Having generated the identity the Security Module could then place it in an area of storage, called the Key Area, which could then be declared to the system as "Read Only".

The Key Area may contain many identities each mapped to a particular module by the module name. The correct functioning of the Integrity Lock would require that strict administrative controls existed and the location of the Key Area would not be commonly known, possibly only to the Plant Engineer since he is ultimately responsible for the safe working of equipment. It is a managerial decision on who would have the necessary information on how to run the Security Module in mode 1, generation mode.

When a control or safety module is called by the Operating System to be executed mode 2 of the Security Module, check mode, would read the control module as a message and generate the identity for that control module. As a function of the Safety or Arbitrator module the current identity would be compared with the stored identity. If the identities did not match then alarm conditions would be raised. However, when the identities match the Operating System would be allowed to execute the module. The procedure described is shown in Figure 3.4.4.1.

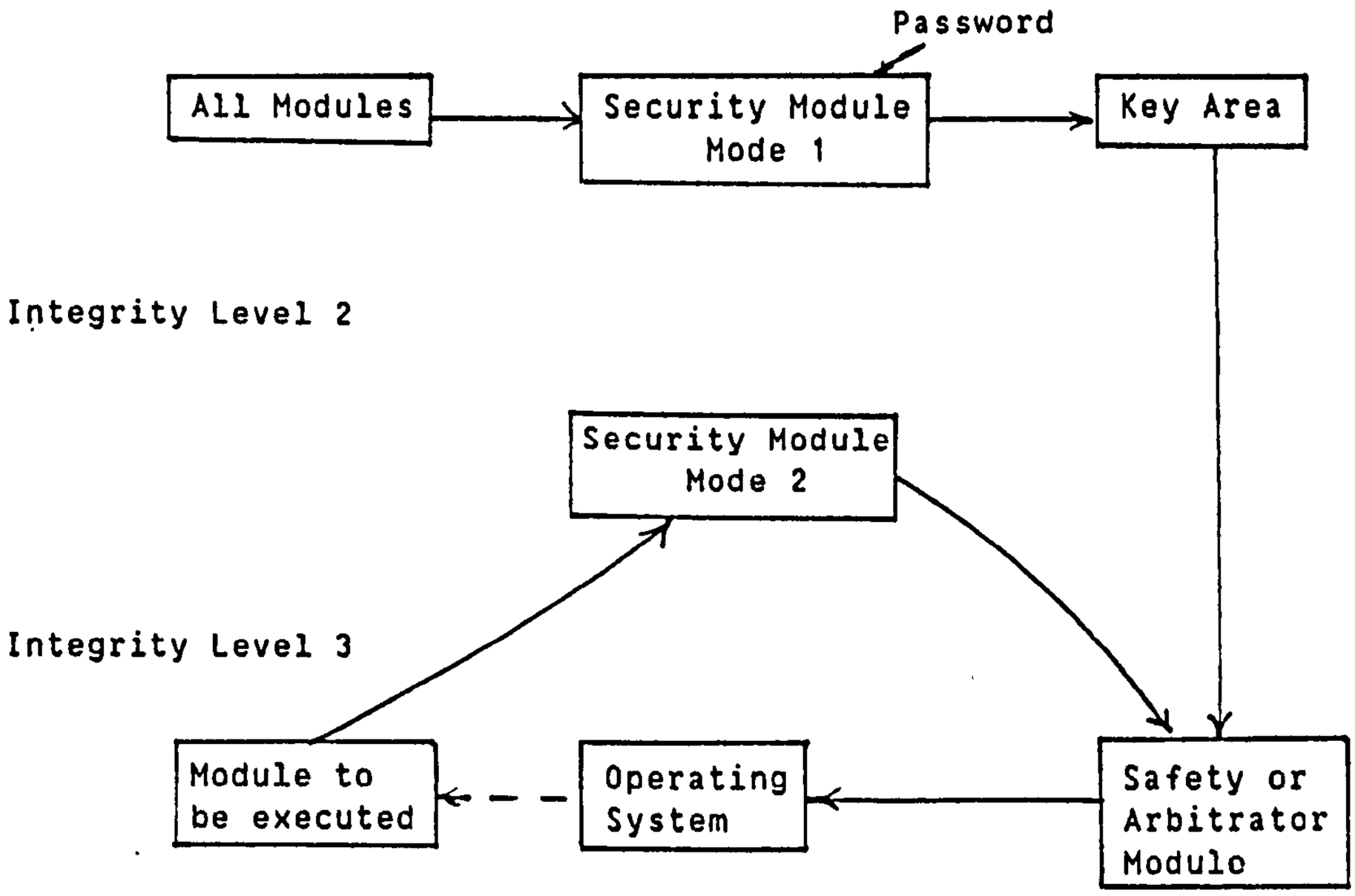
The function of the Security Module in mode 1, generation mode,

is the highest level of integrity, Integrity Level 1. Integrity Level 1 is only executed when a satisfactory password has been entered.

When the Security Module is executing in mode 2, checking mode, the level of integrity is less than mode 1 but higher than the level occupied by the control module, safety module, arbitrator module and the operating system which are all at Integrity Level 3. The level associated with Security Module mode 2 is the Integrity Level 2.

By using a technique such as the Integrity Lock there is a probability of executing a control module, or a Safety/Arbitrator Module, which has previously been categorised as safe.

Figure 3.4.4.1 Integrity Levels



3.4.5 Discussion

There exists methods for tolerating faults arising in the software.

The technique known as Recovery Blocks allows the system to retrace, or back track, to the last known point where safe computation took place and to re-establish a safe working attitude.

But back tracking can cause a 'domino effect' where the system retraces back so far that meaningful control actions are difficult to achieve given that the plant status may have changed significantly since the recovery began. Time can be important in maintaining effective control of an industrial process and if a recovery system cannot roll-back to a satisfactory point in a given time then decisive action will have to be taken, possibly by the Operator.

Recovery Blocks have been used in systems not having a plant status responsibility, such as Command and Control Systems, but in industrial control systems the speed at which the plant status changes may mean that some method is required which will maintain plant safety whilst the fault is investigated. Though Recovery Blocks may serve to protect the safety of the plant in some part they are not sufficient in themselves and require additional features, such as the strategy of using Safety Modules.

The use of Safety Modules is a strategy for seperating the software into control modules, which would determine the necessary control, and into safety modules which would be dedicated to ensuring safe control actions on industrial plant. There would also be an ultimate safety module, the Arbitrator Module, monitoring the safety modules. Such a strategy permits the plant designer/manager to specify or change the safe working limits for the particular plant areas without modifying the control module. The strategy also prohibits the main body of the system from effecting control outside the limits. The strategy is not a fault tolerant technique but it does ensure that safe control can be maintained.

3.5 References

[1] Anderson, T., & Lee, P.A., Fault Tolerance: Principles and Practice, Prentice-Hall, 1981

[2] Andow, P.K., The Numerical Analysis of Hazards and Failures, Proceedings of the Institution of Chemical Engineers Symposium Number 63

[3] Andow, P.K., Real Time Analysis for Process Plant Alarms using a Mini-Computer, Computers and Chemical Engineering, Volume 4, 1980

[4] Chelson, P.O., Reliability Computation using Fault Tree Analysis, NASA-CR-124740, Jet Propulsion Laboratory, 1971

[5] Chen, L., & Avizienis, A., N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation, FTCS-8 Eighth International Conference on Fault Tolerant Computing, 1978

[6] Health and Safety Executive, Microprocessors in Industry, HSE Occasional Paper Series: OP2, H.M.S.O., 1981

[7] Jensen, K., & Wirth, N., Pascal: User Manual and Report, Springer-Verlag, 1974

[8] Leveson, N.G., & Harvey, P.R., Analyzing Software Safety, IEEE Transactions on Software Engineering, Volume SE-9, Number 5, September 1983

[9] Leveson, N.G., & Stolzy, J., Private Communication

[10] Peterson, J.L., Petri Nets, ACM Computing Surveys, Volume 9, Number 3, September, 1977

[11] Randell, B., System Structure for Software Fault Tolerance, ACM SIGPLAN Notices, Volume 10, Number 6, June 1975

[12] Rasmussen, J., WASH 1400 Reactor Safety Study, USAEC, Washington, USA, 1974

CHAPTER 4

The Influence of the Development Process on the Safety of Software

Chapter 3 examined the safe operation of the system through the interactions of the software and the hardware with the emphasis being on the control flow.

In this Chapter the emphasis will be on the control flow of the software.

It is held in this Chapter that errors in the software affect safety and so the Chapter examines the occasions where errors can be introduced into the software, why it is not practicable to remove all errors from the software and introduces a basis for measuring certain features of the software. It is suggested that these measures, though not rigorously proved, do give some indication of the scope for error in an individual item of software.

The development and production of 'safe' software systems has five distinct stages, each having a quality assessment part; requirements specification, system specification, program specification, program production and system test and integration.

Before the software development can begin the originator of the development, the Requesting Authority, needs to obtain a concise understanding of the requirements. Once the software has been implemented and is in operation the Requesting Authority may identify what are considered to be short-comings in the produced system which may necessitate the requirements specification to be recompiled.

The requirements specification may have been prepared by a collection of people from differing disciplines and functions within the organisation, including the end-users. It is, therefore, necessary for the requirements specification to be unambiguous to all those people involved in its preparation. The ambiguity of the

requirements specification is a research topic using established formal mathematical methods to formulate the requirements specification but the use of such formal methods presents a paradox; to make the statements unambiguous the axiomatic methods used require a considerable degree of understanding of mathematical logic which may not present a problem to computer scientists but may to the Requesting Authority, who may not then understand the requirements specification. If written natural language is used for the specification then the computer scientist may find the specification to be imprecise, whereas the Requesting Authority may claim to understand it. At the state of the art there is a risk that ambiguity will persist in requirements specifications for industrial-based control systems.

The system specification, which follows from the requirements specification, is concerned with the design of the total system against the requirements specification.

Program specifications are concerned with the design of specific programs and the interfaces between them to meet the system specification. How the software is structured into a system influences the extent to which the system will conform to the requirements specification, as conjectured in Chapter 3. If the structuring of the software does not conform to the requirements specification then the software may need to be redesigned. To ensure that the software is structured in conformity with the requirements specification, an iterative process is called for involving all those personnel involved in the requirements specification and system specification. The process described is sometimes called the 'design process'.

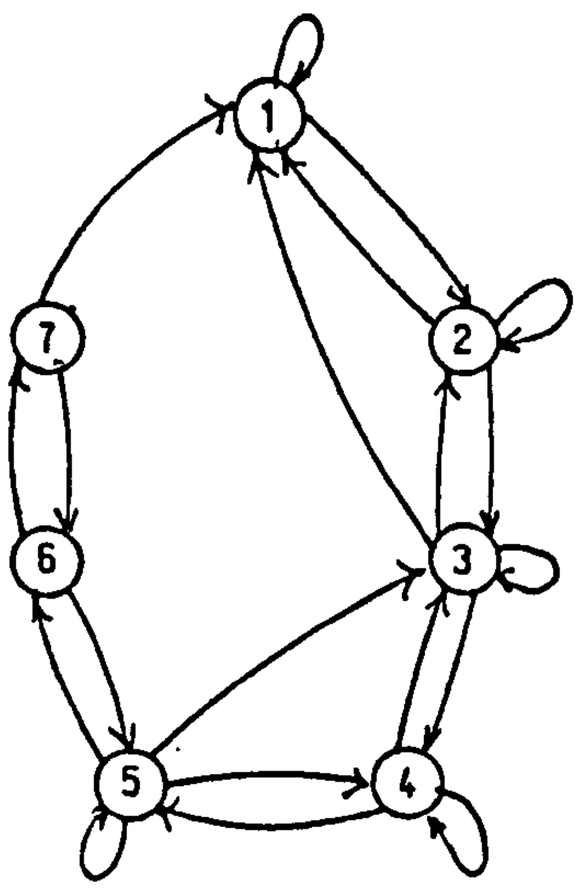
Once the design process has been satisfactorily achieved the 'program development' can begin. During program development the

program is written in accordance with the previously agreed program specification. At the end of program development, the program is tested in isolation from the other programs forming the system.

Following program development is a set of procedures called 'system test and integration' when the individually tested programs are tested as a complete system and integrated into a target implementation.

The multi-stage iterative process which describes the development process can be viewed as a directed graph, Figure 4.0.1, where the nodes represent stages of development, each having an associated activity;

Figure 4.0.1 Software Development Cycle



1 - Requirements Specification
2 - System Specification
3 - Program Specification
4 - Program Production
5 - System Test & Integration
6 - Implementation
7 - Operation

Nodes 1 to 5 in the directed graph of Figure 4.0.1 have an arc from that node and returning to that node to show that progress to the next stage (represented by a node) is not permitted until some form of quality assessment process has been satisfied for that stage. Each normal path between stages, except 7 to 1, has a forward and reverse arc indicating that when the quality assessment cannot be satisfied at a particular node it is necessary to return to the preceeding node and examine the transformation that took place. The arc between stage 7 and stage 1 is uni-directional since the logical

progression from node 1 is to node 2.

To account for the occasion when the quality assessment process has shown discrepancies from the specification such that a radical consideration of the design or structure is required some nodes have additional arcs to nodes other than the succeeding or preceeding node. As an example, if the Requirements Specification cannot be met in the Program Specification it may be necessary to follow the arc from node 3 to node 1.

There is a need in all the stages of the development process to analyse errors and to take the appropriate action. Error analysis takes three forms; error prevention, error detection and error correction.

Error prevention implies the use of good programming practice; the use of the best known methods of software production, for example, the selection of meaningful variable and constant names, structured programming and other methods of programming.

Finding and removing the cause of errors is an intensive and prolonged activity. Though it is important to correct an error it is equally as important to ensure that the knowledge of the error, its original cause and the correction is recorded in the guide to good programming practice being used by the programmer, possibly by some recording mechanism. Error detection and error correction will be carried out by the programmer in the most efficient way as part of his function.

The work in this Chapter on the determination of errors had the following concepts in mind

1 that the software compiles satisfactorily

2 that the Programmer has completed the test-set provided for the purpose and is satisfied that the tests were as exhaustive as one can make them given the restraints of time, effort and

possibly commercial urgency for the software

3 that the development stage when metrics will be used is that stage immediately preceeding the commissioning of the software into operational use, possibly during acceptance testing

4 that satisfactory test limits will have to be determined for the proposed metrics before the application of these metrics to Industry

5 that the test limits and the metrics presented will provide a pass/fail criteria for a Certification Authority seeking to approve the software.

The Chapter starts by examining the reasons for errors remaining in software, even after extensive testing. Having discussed the software development process in terms of a feedback model a method is developed to indicate the potential that exists within the software for perturbing the software through single character errors, followed by a discussion on the need to declare variable and constant names and a report on an experiment conducted to examine the probability of error through incorrect interpretation of mneomonics.

The final section of the Chapter is concerned with the development of a measure called Plexus which measures the syntactic complexity of software.

4.1 The Feedback Model of Software Production

The development of software involves a number of stages. The exact number of stages and their relationship is largely dependent on the organisation under which the development is done, the extent of the project and the development methodology used. Many authors, for example Sommerville, [12], Kopetz, [6], Peters, [8], have all tried to model the development process in varying degrees of detail. At the level of the gross model there is a consensus of opinion that five stages exist; requirements specification, system specification, program specification, coding and acceptance testing. Errors can be created, detected and corrected at each of these stages but there will still be residual errors which are not detectable until after the software has been commissioned. This section of the Chapter will demonstrate the enormity of the task required to eliminate all software errors (if that was possible) prior to commissioning.

4.1.1 Process Model

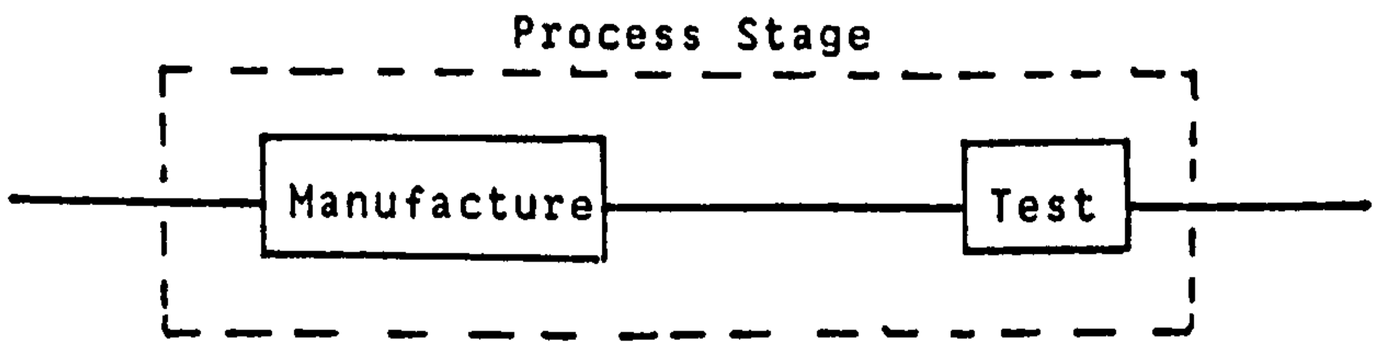
The process of developing software can be compared to the production processes of a manufacturing line and it is this analogy that has been used in this Chapter to develop a model.

Figure 4.1.1.1 Production Process

Each process in the model has an input and an output with rejects, from that process, being rejected at that process.

Inside each process stage there are two sub-processes, namely, manufacture and test, Figure 4.1.1.2.

Figure 4.1.1.2 Manufacture and Test within a Process Stage



Since testing will cause some corrective action to be taken on the rejects detected then these actions can be considered as feedback loops, Figure 4.1.1.3. Ideally testing will be such that no errors are passed to the next stage and all errors are fed-back to the preceeding manufacture sub-process or sub-processes for correction. However, the model must consider that errors can be created at both the manufacture and test stages and that the test coverage is limited, Figure 4.1.1.4.

Figure 4.1.1.3 Rejection Feedback in a Process Stage

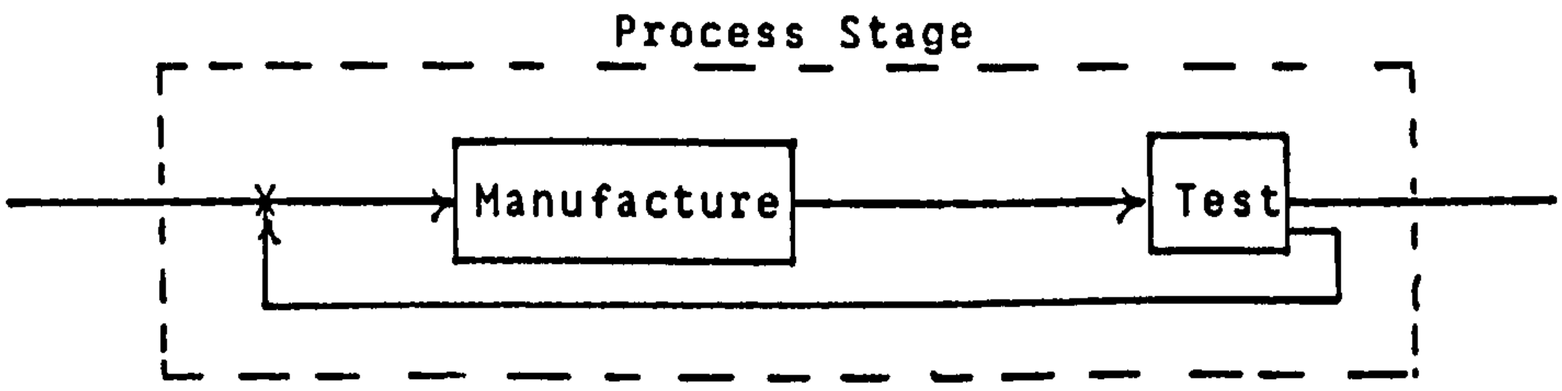
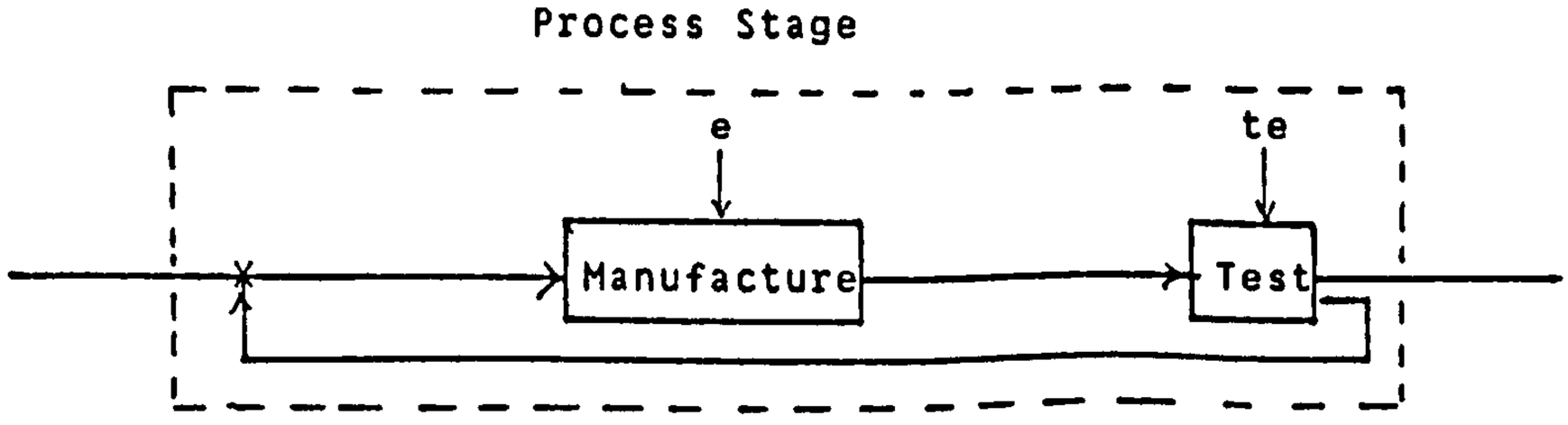


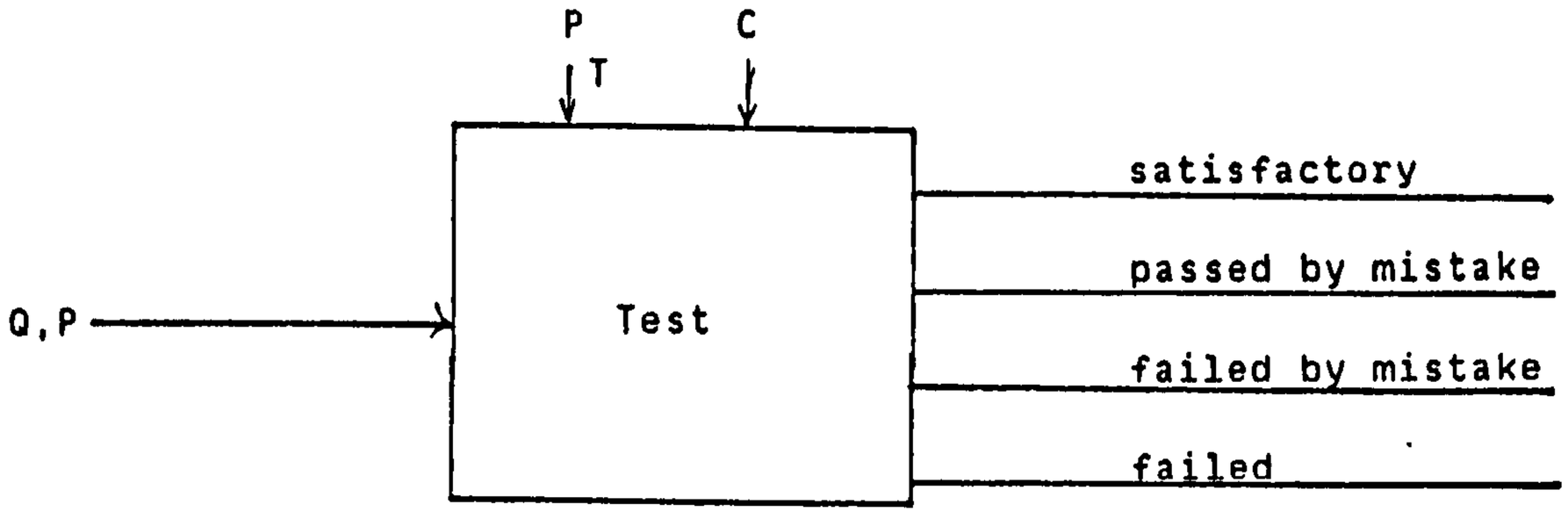
Figure 4.1.1.4 Test Coverage



where e is related to the errors introduced during manufacture and te is related to the errors undetected or allowed to pass as a consequence of the testing practice.

The input to the test stage will be a duple (quantity Q, probability of error P) and the output will be (failed, failed by mistake, passed by mistake, satisfactory). The model is now shown in Figure 4.1.1.5.

Figure 4.1.1.5 Test Stage

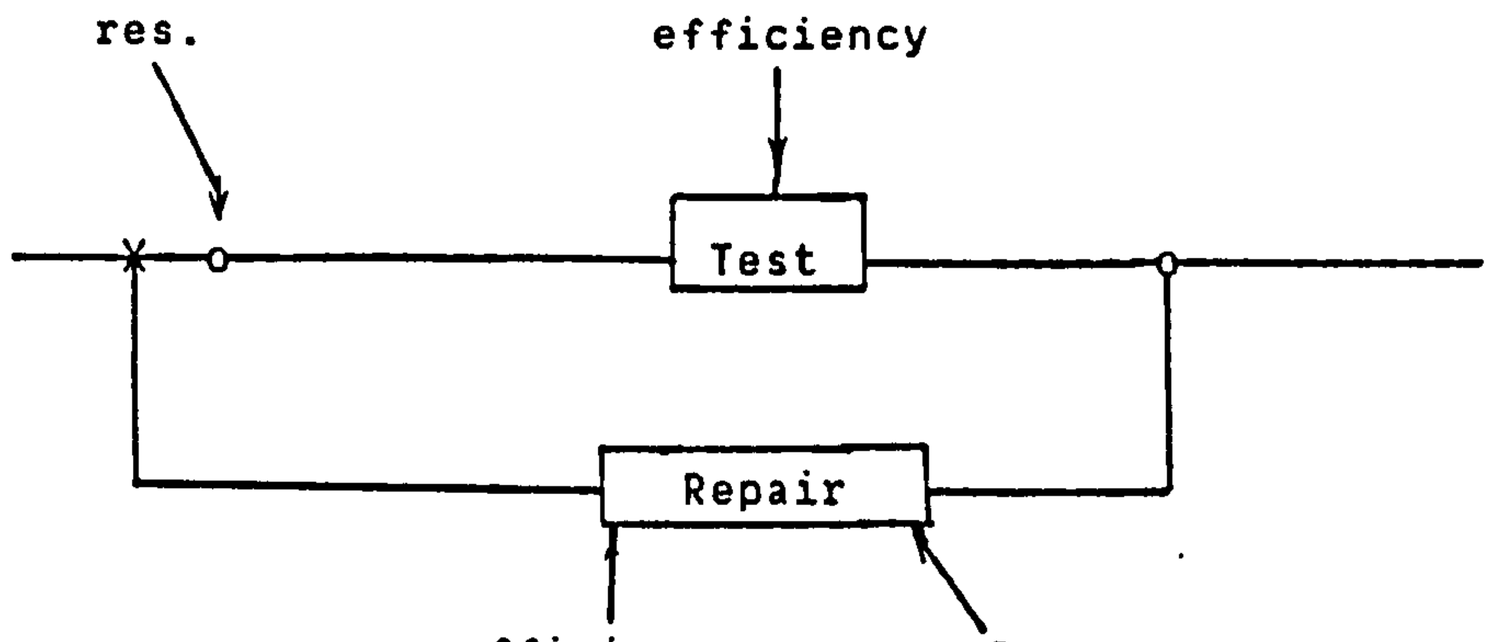


The test coverage (normally less than 100%) is shown by C and the probability of testing determining the error shown by PT. The quantity considered to be satisfactory, Qs, is given as

$$Qs = Q.Pr((1-PT).(1-C)) + Q.(1-P).(1-C)$$

The test stage will have an output leading to the input of the next manufacturing process and an output corresponding to the erroneous components detected, either correctly or by mistake. The detected erroneous components will be subjected to some form of repair mechanism before being re-submitted to the test stage. The repair mechanism has been added to the test stage model shown in Figure 4.1.1.6.

Figure 4.1.1.6 Repair Mechanism



where res. is the residual error

β is the effectiveness of repair

If it is assumed that the detection of errors by the model shows a reduction in errors according to some exponential function then the residual errors, res, is given as

$$res = f(t) = \exp(-at)$$

From control theory the testing stage can be given as a first order loop whose transfer function is given using Z-transformation

$$Qp = \frac{Z}{Z - e^{-at}}$$

where Qp = quantity produced

Since the repair mechanism forms a feedback loop from the test stage to the input of the manufacturing stage the residual error can be viewed using a first order feedback model

$$res = \frac{Qp}{Qr}$$

Qr = quantity repaired

$$\text{since } Qr = \frac{1 + \beta Z}{Z - e^{-at}}$$

$$res = \frac{\frac{Z}{Z - e^{-at}}}{\frac{1 + \beta Z}{Z - e^{-at}}} \cdot \frac{Z - e^{-at}}{Z}$$

$$= \frac{Z - e^{-at}}{Z - e^{-at} + \beta Z}$$

$$= \frac{Z - e^{-at}}{Z(1 + \beta) - e^{-at}}$$

$$= \frac{1}{1 + \beta} \cdot \frac{Z - e^{-at}}{Z - \frac{e^{-at}}{1 + \beta}}$$

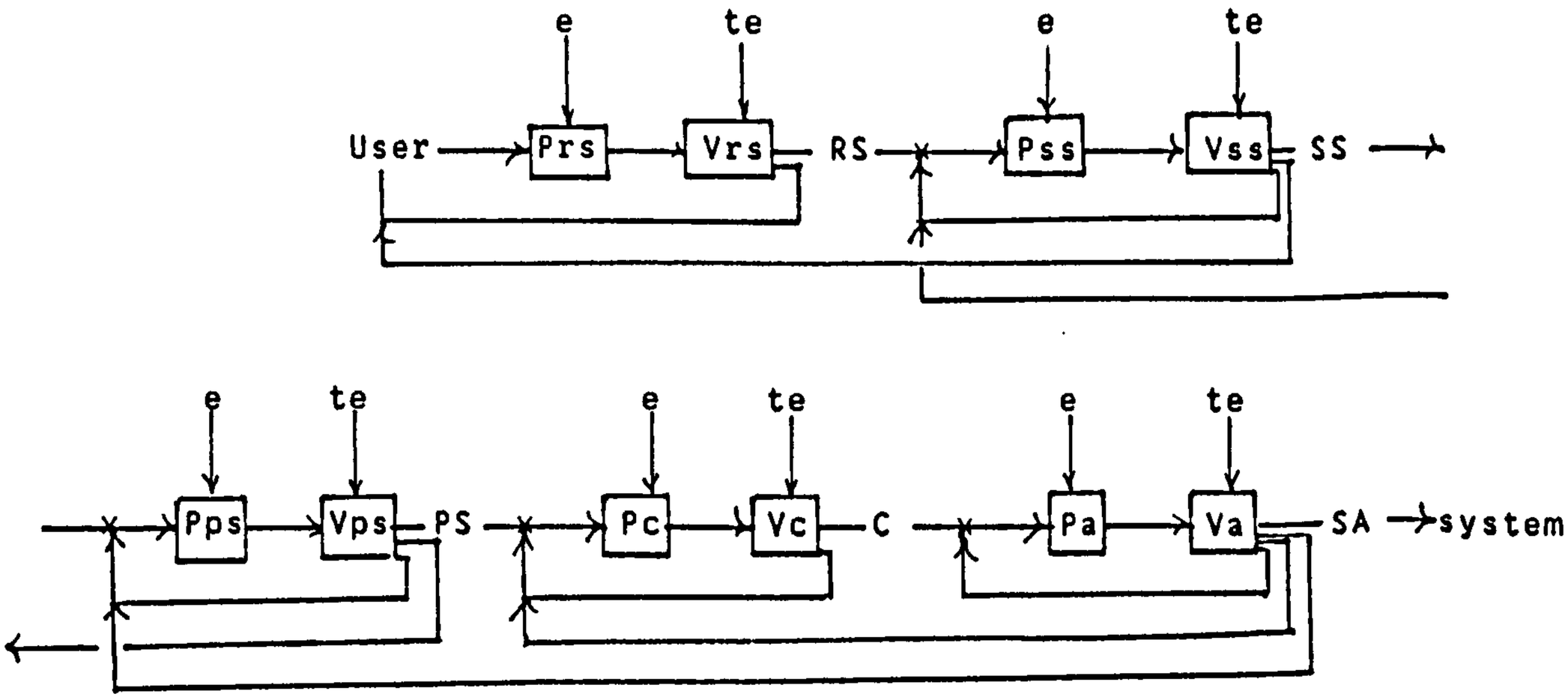
To reduce the residual errors requires a level of test efficiency and test coverage above that attainable. Therefore, it is concluded that there will always be a number of residual errors.

4.1.2. Software Model

Taking the model above and substituting the five stages of software development for the process stages, the model becomes that shown in Figure 4.1.2.1.

It has been suggested by some researchers, notably Boehm [1], that the majority of error-detection effort should be committed to the requirements specification stage and so reduce the number of errors needing to be detected at the following stages, especially at the acceptance tests. Whilst such a strategy may be intuitively sound it should not be assumed to be sufficient in itself since an extreme amount of effort at the requirements specification stage could cause a bottle-neck in the development process and be counter-productive to achieving the project time-scales.

Figure 4.1.2.1 Software Feedback Model



where Pi = Production of stage i

Vi = Validation of stage i

RS = Requirements Specification

SS = System Specification

PS = Program Specification

C = Code

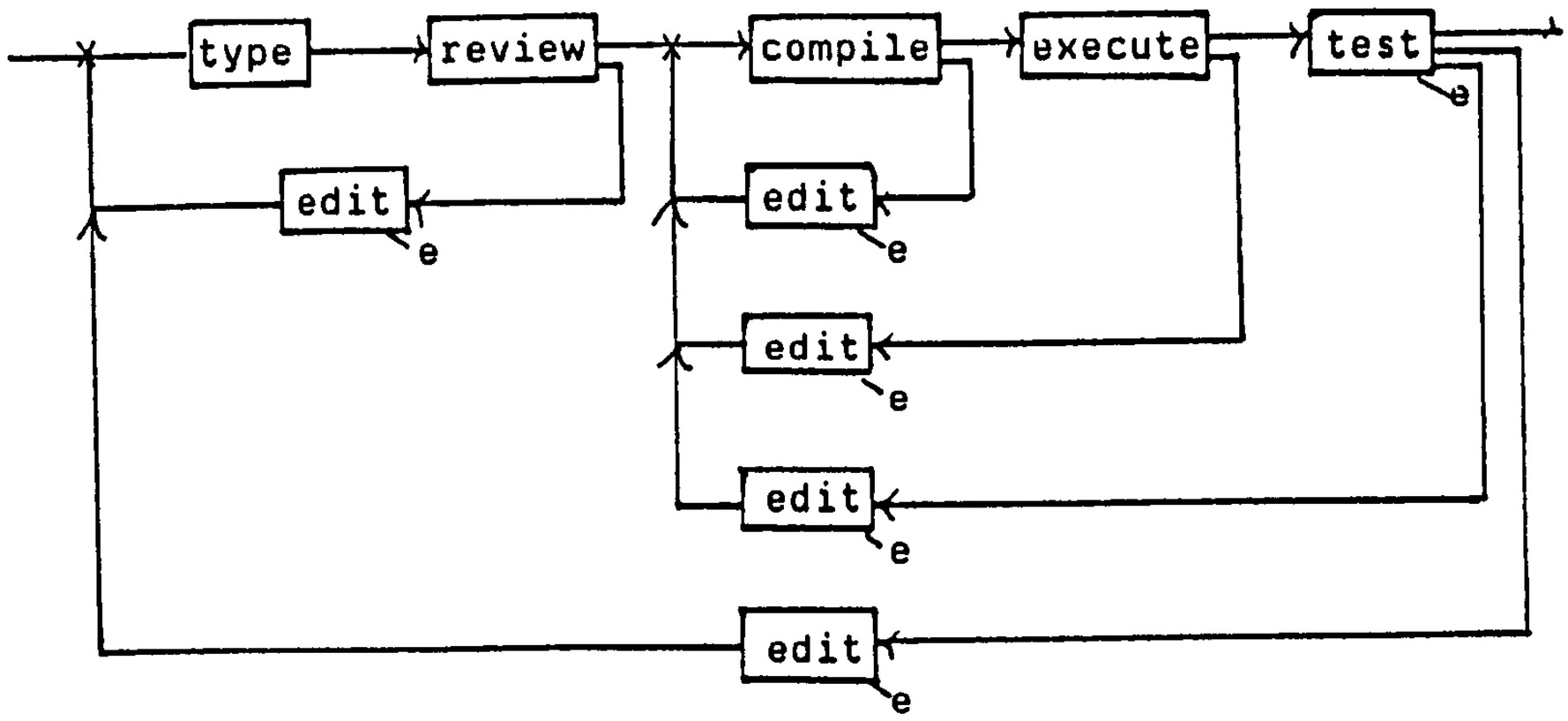
SA = Accepted system

4.1.3 Macro Model

The software development model can be applied to varying levels of detail. For instance, the process of producing the program code can be considered to consist of the following stages; typing the original program, review of the program, program compilation, execution of the code, testing of the code and editing stages, forming a model like that in Figure 4.1.3.1.

The edit process itself can be further modelled, Figure 4.1.3.2.

Figure 4.1.3.1 Software Edit Stages

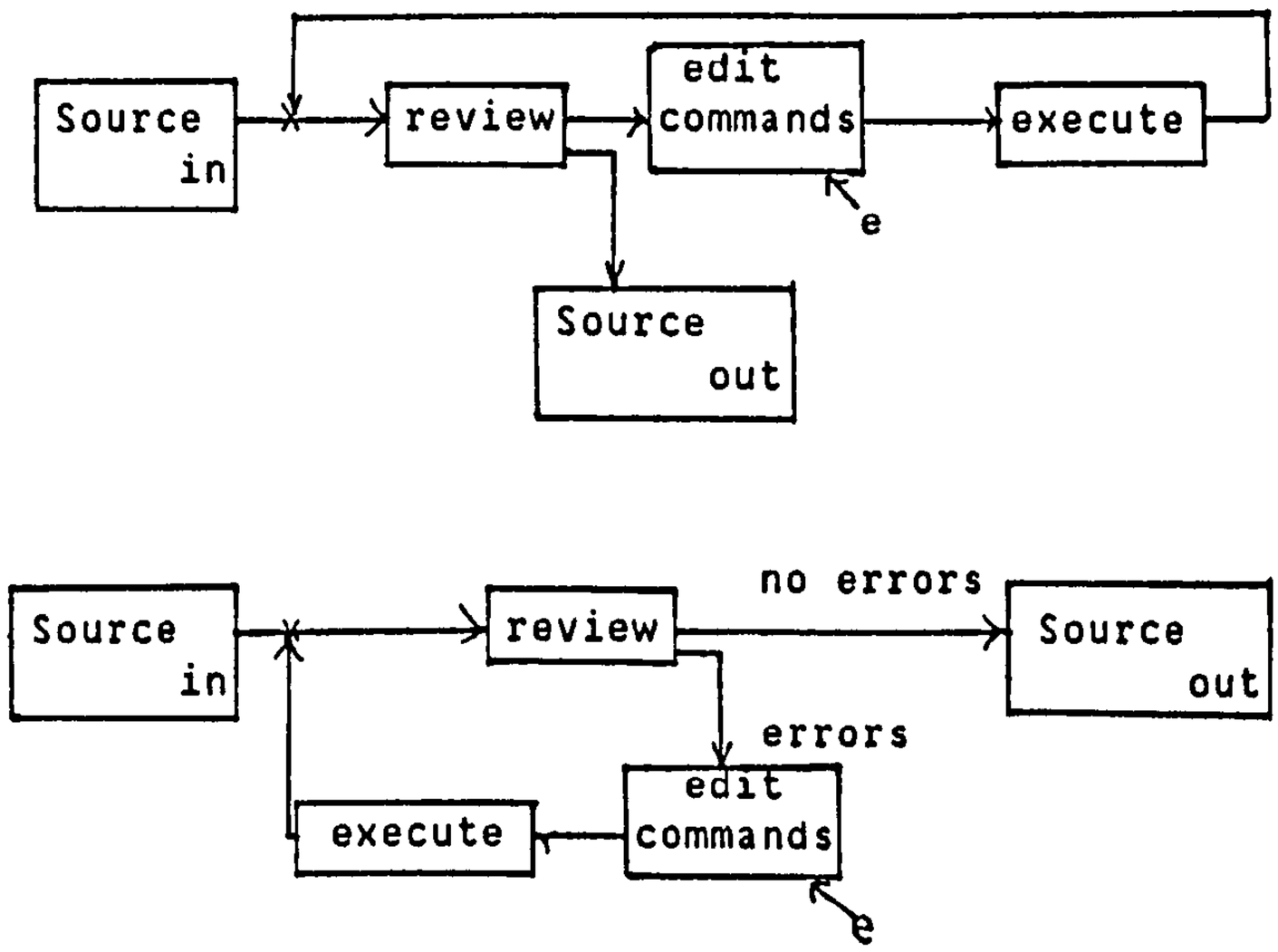


where e is the probability of introducing errors

It is conjectured that each stage of development is prone to errors and the process of correcting errors is itself prone to further errors. It cannot be assumed that the use of extreme amounts of effort at the requirements specification stage will produce significant improvements in error reduction. It is suggested that all items of non-trivial software contain some erroneous feature.

If all software has at least one erroneous feature then the criteria in testing hazard-related systems should be to identify those classes of errors which could cause a dangerous state to exist.

Figure 4.1.3.2 Edit Process



4.2. Single Character Errors in Programs

In writing even the simplest program there is a probability of a character being typed in error. When the program is submitted to the compiler a considerable number of these mistakes will be detected but it is not practically possible to remove all errors. There are occasions when the compiler is not able to detect the mistake.

The simple program below forms the basis of the discussion that follows;

```
PROGRAM name (FILE);
VAR XY,XZ:REAL;
BEGIN
  XY := 2;
  XZ := 3;
END.
```

If the initial letter of the keyword BEGIN, "B", was mistakenly typed as the letter, "N", the compiler would be able to detect that NEGIN is not included in the list of reserved words for that language, also that NEGIN is not a declared variable and so reject the line as being in error. The rejection of the line would cause the compilation to fail since BEGIN indicates the start of a procedure block.

Errors in program variables are not always so obvious. Take, as an example, a language which requires all variables to be declared. In a program, like that above, written in such a language the effect of typing X2:=2 instead of the correct XY:=2 would be easily detected since X2 has not been declared as a variable. Had the error been that XZ:=2 had been typed instead of XY:=2 then the error could not be detected by the compiler unless it checked for unused variables,XY being unused. If the language did not require all variables to be declared, as for example with the language BASIC, then neither of these example errors would have been detectable by the compiler or interpreter, but they should have been detected by the tests of the programmer, (concept 2).

When a single character is omitted or altered to another character, or when there is a single additional character there are similar opportunities for undetected errors.

4.2.1 Theory

There are three error classes of mistakenly typed programs;

- omitted single character (Po)
- additional single character (Pi)
- altered single character (Pa), which can also be considered as a combination of Po and Pi

To examine the effects of each of these error classes the following assumptions have been made;

- that individual errors are independent and no account is taken of complimentary errors since the probability of such is considered to be low
- that each character has an equal probability of error.

If it is assumed that errors can be introduced as a result of a mistakenly typed character and remain undetected the expectation of such a mistake is given as E{mistake} and the number of characters is given as Nc, then the expectation of the number of mistakes on initial input of the program is

$$E\{\text{No. of mistakes}\} = E\{\text{mistake}\} \cdot Nc$$

If the probability of making an undetected mistake is given as Pm then the expectation of the number of undetected errors is given as

$$E\{\text{No. undetected errors}\} = E\{\text{No. of mistakes}\} \cdot Pm$$

which can be expanded to

$$E\{\text{No. undetected errors}\} = E\{\text{mistake}\} \cdot Nc \cdot Pm$$

There are three classes of error influencing the number of undetected errors which are related to

- the expectation of the number of undetected alterations, E{a},

- the expectation of the number of undetected omissions, E{o},

- the expectation of the number of undetected additions, E{i}.

The number of undetected errors is given as

$$E\{\text{No. undetected errors}\} = E\{u\} = E\{a\} + E\{o\} + E\{i\}$$

The expectation of an undetected altered character, E{a}, is given by

$$E\{a\} = \frac{Pa}{Cs - 1} \cdot Nc \cdot Pm$$

where Pa is the probability of a character in a certain character position being altered and remaining undetected, Nc is the number of characters in the program, Pm is the probability of making an error in typing a character and Cs is the number of characters in the character set permitted in the language.

The Expectation of undetected altered characters in a program is given by

$$E\{a\} = \frac{(Pa1 + Pa2 + \ldots . Pan)}{Cs - 1} \cdot Nc \cdot Pm$$

$$= \frac{Na}{Cs-1} \cdot Nc \cdot Pm$$

where Pan is the probability of a character being altered in the n-th position of an ordered set of characters by each admissible character from the set of characters in the character set and being undetected. The number of altered and undetected characters is Na.

The expectation of the number of undetected omissions, E{o}, is given by

$$E\{o\} = No \cdot Nc \cdot Pm$$

where No is the number of occasions an omitted character will be undetected.

The expectation of the number of undetected additional characters, E{i}, is given by

$$E\{i\} = (Ni / Cs) \cdot Nc \cdot Pm$$

where Ni is the number of occasions an additional character.

The total number of possible errors is determined by;

$$\text{No. possible omissions} = Npo = Nc$$

$$\text{No. possible alterations} = Npa = Nc \,.\, Cs-1$$

$$\text{No. possible insertions} = Npi = (Nc + 1) \,.\, Cs$$

$$= Nc \,.\, Cs + Cs$$

giving

$$\text{Max. possible errors} = Npe = Npo + Npa + Npi$$

$$= 2\ (Nc \,.\, Cs) + Cs$$

since No, Na, Ni and Npe will vary with the size of the program the normalised ratio between these is given by the ratio

$$F = \frac{(Na + No + Ni)}{Npe} \,.\, Pm$$

which is called the Fallibility Index of a program and expresses some measure of the extent that undetected errors are possible. Since all errors contain a risk and the risk increases in relation to the number of errors, the Fallibility Index indicates the scope for undetected errors and gives some indication to the risk. As the scope for errors remaining undetected increases so does the probability that there will be at least one error which has the characteristics of creating a catastrophe.

4.2.2 Example Programs

The hypothesis above can be demonstrated by analysis of two simple programs. First, the Pascal-like program used earlier with line numbers added for clarity of discussion;

```
1  PROGRAM NAME (FILE);
2  VAR XY,XZ:REAL;
3  BEGIN
4    XY:=2;
5    XZ:=3;
6  END.
```

Although the program header "PROGRAM NAME(FILE);" is required in some languages it is not included in these calculations since it only

adds to the length of the program without constraining it in any way and FILE is only used by the compiler to signal the input/output requirements.

To determine the expectation of undetected altered errors, E{a}, an examination shows that alteration of the keywords PROGRAM, VAR, REAL, BEGIN and END will be detected by the compiler. Similarly, if an alteration caused either of the declared variables to have the same identity the similarity would be detected. If the altered character caused the variable to have an identity different from those in lines 4 & 5, then the error would be detected. There are only 4 instances where an alteration would remain undetected;

- P(Y=>Z) from line 4 [alternates = 1]
- P(2 => 0,1,3..9) [9]
- P(Z => Y) from line 5 [1]
- P(3 => 0..2,4..9) [9]

The number of undetected alternate characters, Na, is 20.

Counting the number of characters (including Newline as a terminating character but excluding leading Spaces and the program header as being unnecessary to the calculation) gives Nc = 41. The permissible character set Cs for the Pascal-like language is 96. So E{a} can be calculated as

$$E\{a\} = \frac{Na}{Cs-1} \cdot Nc \cdot Pm$$

which becomes

$$E\{a\} = 8.6 \cdot Pm$$

To calculate E{o}, the expectation of undetected omissions, the program is analysed for instances where an omitted character would be undetected by the compiler. In the example program the only instance where an omission would be undetected is in the program header which has been omitted. So

$$E\{o\} = 0 \text{ and } No = 0$$

The expectation of undetected additions, E{i}, requires the analysis of the program to determine the instances where an additional character could be inserted and undetected. The integer assignments could have additional characters before (+,-,.,0..9) or after (.,0..9) and be undetected. Since the program header is being ignored Ni = 48, E{i} can now be calculated as

$$E\{i\} = \frac{Ni \;.\; Nc \;.\; Pm}{Cs}$$

$$= 20.5 \;.\; Pm$$

The sum of E{a}, E{o} and E{i} is the expectation of the number of undetected errors and for the example program

$$E\{\text{No. undetected errors}\} = E\{u\} = E\{a\} + E\{o\} + E\{i\}$$

$$= 29.1 \;.\; Pm$$

$$\text{No. possible errors} = Npe = 2(Nc \;.\; Cs) + Cs$$

$$= 7968$$

giving a Fallibility Index of

$$F = \frac{(Na + No + Ni)}{Npe} \;.\; Pm$$

$$= 0.0085 \;.\; Pm$$

As a comparison an equivalent FORTRAN-like program is analysed in the same way. The program is

```
XY=2
XZ=3
END
```

In the Fortran-like example Nc = 14 and Cs = 48.

There are 6 possibilities for an undetected alteration;

- P(X => A..W,Y,Z) [25 alternates]
- P(Y => A..X,Z,0..9) [35]
- P(2 => 0,1,3..9,A..Z) [35]
- P(X => A..W,Y,Z) [25]
- P(Z => A..Y,0..9) [35]
- P(3 => 0..2,4..9,A..Z) [35]

thus Na = 190 and E{a} becomes

$$E\{a\} = \frac{Na}{Cs-1} \cdot Nc \cdot Pm$$

$$= 56.6 \cdot Pm$$

There are 4 possibilities of undetected omissions: X or Y in the case of XY and X or Z in the case of XZ being omitted individually, No = 4. So E{o} is given by

$$E\{o\} = No \cdot Nc \cdot Pm$$

$$= 4 \cdot 14 \cdot Pm = 56 \cdot Pm$$

Additional characters can be introduced without being detected as errors. The notation used to demonstrate these instances uses the symbol '=>' to indicate "may be perturbed to" and the symbol '|' to read "a character or set of characters preceeding a character or set of characters". In the example program the instances are:

- P(X => A..Z|X) [26 additions]
- P(X => X|A..Z,0..9) [36]
- P(Y => Y|A..Z,0..9) [36]
- P(2 => .,+,-,0..9,A..Z|2) [39]
- P(2 => 2|0..9,.) [11]
- P(X => A..Z|X) [26]
- P(X => X|A..Z,0..9) [36]
- P(Z => Z|A..Z,0..9) [36]
- P(3 => .,+,-,0..9,A..Z|3) [39]
- P(3 => 3|0..9,.) [11]

Thus Ni = 296 and E{i} becomes

$$E\{i\} = \frac{Ni}{Cs} \cdot Nc.Pm$$

$$= 86.3 \cdot Pm$$

The expectation of the number of undetected errors is

$$E\{u\} = E\{a\} + E\{o\} + E\{i\}$$

$$= 198.9 \cdot Pm$$

The number of possible errors, Npe, is given as

$$Npe = 2(14 , 48) + 48 = 1392$$

Thus the Fallibility Index is

$$F = \frac{490}{1392} . Pm = 0.352 . Pm$$

If Pm is assumed to be equi-probable for both the Pascal-like language and Fortran-like language then a comparison can be made between the two trivial programs analysed. In the Pascal-like program the Fallibility Index was 0.85% whereas in the Fortran-like program the Fallibility Index was 35.2% suggesting that in the simple examples the Pascal-like program can be considered to be less fallible and having a lower risk of error.

The mandatory use of the declaration of variables has at least one disadvantage; that declarations can be mistaken for other similar but unique identifiers because of a single character error within the body of the program. These errors can result from the omission, insertion or deletion of a single character. Perturbations to the program resulting from single character errors are an indication of the scope for undetected errors existing in software which, when in use, could be in control of potentially hazardous equipment.

A realistic program taken from Jensen and Wirth, [5] p.38, written in Pascal, FORTRAN and BASIC will now analyse. To illustrate the analysis each program has 4 columns; number of characters on the line, number of undetectable alternate characters Na, number of undetectable character omissions No and the number of undetectable character additions Ni.

4.2.3 Pascal Version

The program has been compiled using UCSD PASCAL.

Nc		Na	No	Ni
	PROGRAM graph2 (output);			
16	CONST d=0.0625;	45	6	72
6	s=32;	18	2	35
7	h1=34;	18	2	35
7	h2=68;	18	2	35
11	c=6.28318;	54	7	82
8	lim=32;	18	2	35
21	VAR i,j,k,n:INTEGER;	0	0	0
10	x,y:REAL;	0	0	0
24	a:ARRAY[1..h2] OF CHAR;	10	1	0
6	BEGIN	0	0	0
29	FOR j:=1 TO h2 DO a[j]:=' ';	94	1	153
19	FOR i:=0 TO lim DO	17	0	18
6	BEGIN	0	0	0
8	x:=d*i;	45	0	0
23	y:=EXP(-x) * SIN(c*x);	68	1	19
12	a[h1]:=':';	74	1	128
20	n:=ROUND(s*y) + h1;	68	1	0
11	a[n]:='*';	77	0	128
21	IF n < h1 THEN k:=h1	47	2	2
11	ELSE k:=n;	13	0	0
30	FOR j:=1 TO k DO WRITE(a[j]);	34	0	25
9	WRITELN;	0	0	0
10	a[n]:=' '	77	0	0
4	END	0	0	0
5	END.	0	0	0

Nc = 334 Cs = 64 Na = 795 No = 28 Ni = 767

Substituting, the Expectation of an undetected error for this version is:

$$E\{a\} = \frac{Na}{Cs-1} .Nc . Pm$$

$$= 4215 . Pm$$

$$E\{o\} = No . Nc . Pm$$

$$= 9352 . Pm$$

$$E\{i\} = \frac{Ni . Nc}{Cs} . Pm$$

$$= 4003 . Pm$$

$$E\{u\} = E\{a\} + E\{o\} + E\{i\}$$

$$= 17570 . Pm$$

$$Npe = 2(Nc . Cs) + Cs$$

$$= 42816$$

giving a Fallibility Index of

$$F = \frac{(Na + No + Ni)}{Npe} \cdot Pm$$

$$= \frac{1590}{42816} \cdot Pm = 0.037 = 3.7\%$$

4.2.4 BASIC Version

This program has been prepared using Microsoft BASIC-80.

Nc		Na	No	Ni
11	10 DIM A$[68]	43	0	30
14	20 FOR J=1 TO 68	79	2	113
15	30 A$[J]=CHR$(32)	81	2	94
7	40 NEXT J	0	1	0
14	50 FOR I=0 TO 32	79	2	113
10	60 X=I*.0625	124	6	229
25	70 Y=EXP(-X)*SIN(6.28318*X)	191	8	336
16	80 A$[34]=CHR$(58)	93	4	63
11	90 N%=32*Y+34	157	6	243
16	100 A$[N%]=CHR$(42)	76	3	94
29	110 IF N%<34 THEN K=34 ELSE K=N%	200	6	369
13	120 FOR J=1 TO K	70	0	111
13	130 PRINT A$[J];	38	1	98
7	140 NEXT J	0	1	0
6	150 PRINT	0	0	2
16	160 A$[N%]=CHR$(32)	76	3	94
7	170 NEXT I	0	1	0
4	180 END	0	0	0

Nc = 234 Cs = 68 Na = 1307 No = 46 Ni = 1989

$$E\{a\} = \frac{Na}{Cs-1} .Nc . Pm$$

$$= 4565 . Pm$$

$$E\{o\} = No . Nc . Pm$$

$$= 10764 . Pm$$

$$E\{i\} = \frac{Ni . Nc}{Cs} . Pm$$

$$= 6845 . Pm$$

Therefore

$$E\{u\} = E\{a\} + E\{o\} + E\{i\}$$

$$= 22174 . Pm$$

$$Npe = 2(Nc . Cs) + Cs$$

$$= 31892$$

giving a Fallibility Index of

$$F = \frac{(Na + No + Ni)}{Npe} \cdot Pm$$

$$= \frac{3342}{31892} \cdot Pm = 0.105 = 10.5\%$$

4.2.5 FORTRAN Version

This program has been tested using the Microsoft Fortran-80.

Nc		Na	No	Ni
	PROGRAM GRAPH2			
14	LOGICAL M(68)	18	2	0
10	DO1J=1,68	76	3	92
11	1 M(J)=' '	62	1	156
10	DO2I=0,32	75	3	89
11	X=I*0.0625	106	6	194
25	Y=EXP(-X)*SIN(6.28318*X)	165	8	302
10	M(34)=':'	71	3	120
10	N=32*Y+34	160	5	216
9	M(N)='*'	62	1	156
16	IF(N.LT.34)K=34	150	4	216
15	IF(N.GE.34)K=N	142	3	232
12	WRITE(1,3)M	36	1	0
19	3 FORMAT(1H ,68A1)	78	5	30
11	2 M(N)=' '	62	1	156
4	END	0	0	0

Nc = 187 Cs = 48 Na = 1263 No = 46 Ni = 1959

$$E(a) = \frac{Na}{Cs-1} .Nc \cdot Pm$$

$$= 5025 \cdot Pm$$

$$E(o) = No \cdot Nc \cdot Pm$$

$$= 8602 \cdot Pm$$

$$E(i) = \frac{Ni \cdot Nc}{Cs} \cdot Pm$$

$$= 7632 \cdot Pm$$

Therefore

$$E(u) = E(a) + E(o) + E(i)$$

$$= 21259 \cdot Pm$$

$$Npe = 2(Nc \cdot Cs) + Cs$$

$$= 18000$$

giving a Fallibility Index of

$$F = \frac{(Na + No + Ni)}{Npe} \cdot Pm$$

$$= 0.182 = 18.2\%$$

Although the language Fortran requires the first six characters of each line of a program to be spaces, continuation markers or labels, the analysis has only considered the effect of an erroneous label because any additional labels will not introduce an error and also any label which is not numeric will be rejected by the compiler.

4.2.6 Discussion

An increase in the number of characters to write a program causes a corresponding improvement in the Fallibility Index (Figure 4.2.6.1 and Graph 1). One possible reason for such an improvement is the increase in characters causing an increase in the useful redundancy.

Figure 4.2.6.1.

Language	Nc	F%
Fortran	187	18.2
Basic	234	10.5
Pascal	334	3.7

The analysis of programs is tedious and time-consuming and can be eased if automatic analysis techniques are used to analyse programming languages that have a publicly available syntax.

As part of the research a suite of programs was developed to analyse programs. Since the algorithm used perturbed each character position in the program with each character from the allowable character set for each class of error the computer time needed was considerable. The analysis was performed using a DEC VAX 11/750 computer with programs written in the language Pascal and C supported by the YACC and LEX tools of UNIX. An analysis typically took 30 minutes of computer time to execute. Refinement of the algorithm used would, no doubt, cause the analysis to be executed with less

computer time but the current algorithm has established the principle that the analysis can be automated.

4.3 The Need to Declare Variable and Constant Names

High-level programming languages, like FORTRAN and BASIC, do not require the declaration of variable and constant names at the beginning of a program. With the introduction of the so-called structured languages, like ALGOL and PASCAL, which require variable names to be declared there has been some discussion on the need for declaring variables and constants, how many characters should be used in a name and also how representative such names should be.

Chapter 4.2 considered the influence that variables have on the resulting Fallibility Index. It is hypothesised that the declaration of variables based on the uniqueness of a name gives the variable security from misinterpretation and the number of instances of names is large before unique names contribute to the misunderstanding of the objects.

4.3.1 Variables in Declarative Languages

If no range checks are in force and no account is taken of omissions or additions in the symbol the probability of an error in the variable Pwv is

$$Pwv = I \cdot m \cdot Pm \cdot \frac{Nd}{N}$$

where I is the number of instances

m is the number of characters forming the variable

Nd is the number of permissible positions

N is the size of the permissible character set

To calculate the probability of an error in the symbol consider the occasion where the symbol is wrong (Ps) and the occasion when the symbol is correct but the value is wrong (1-Ps), giving the probability of using the wrong variable, Pwrong, as

$$Pwrong = I.Ps + Ps + (1-Ps).m.Pm.\frac{Nd}{N} + Ps(1-Ps).m.Pm.\frac{Nd}{N}$$

putting the expression in terms of Ps, it becomes

$$2Ps.m.Pm.\frac{Nd}{N} = I+1 - m.Pm.\frac{Nd}{N} + m.Pm.\frac{Nd}{N}$$

$$Ps = \frac{N \,.\, I+1}{2Nd.Pm.m}$$

which suggests that declarations in the local context increase the redundancy, yet in practice highly secure programs are not infinitely large.

4.3.2 Constants in Declarative Languages

Constant declarations are generally of the form

$$C_1 \ldots\ldots C_n = D_1 \ldots\ldots D_m$$

for numbers only the probability of using the wrong value Pwrong is given as

$$Pwrong = I.Pd$$

for numbers and declarations, $Pwrong = I.Ps + Ps + Pd$

which optimises when

$$I.Pd > I.Ps + Ps + Pd$$

and

$$Ps < Pd\ \frac{(I-1)}{I+1} - Pd$$

where $Ps = S-1\ /\ (Nc-1)^{n} \,.\, Pm \,.\, \frac{Nc}{N}$

S = No. Symbols

Nc = character set

m = no. of digits

I = no. of instances

n = no. of character positions

and

$$Pd = m \,.\, Pm \,.\, \frac{Nd}{N}$$

In terms of the number of instances, I, the equation becomes

$$(S-1\ /\ (Nc-1)^{n}) \,.\, Pm \,.\, \frac{Nc}{N} = m.Pm.\frac{Nd}{N} \,.\, \frac{I-1}{I+1}$$

which reduces to $(S-1 \,.\, Nc) \,/\, (Nc-1)^{n} = m.Nd. \frac{I-1}{I+1}$

substituting A for $Nc \,/\, (Nc-1)^{n}$ and B for m.Nd the expression becomes

$$(S-1).A = B \,.\, \frac{I-1}{I+1}$$

$$I.(S-1).A + (S-1).A = B.I - B$$

$$I.((S-1).A-B) = -(S-1).A-B$$

$$I = -\frac{(S-1 + B/A)}{(S-1 - B/A)}$$

So if Nc = 26 , Nd = 10 and m,n = 8

$A = Nc \,/\, (Nc-1)^{n} = \frac{26}{258}$ $\qquad B = m.10$ $\qquad B/A = \frac{(m.10.25^{8})}{26}$

when is I > 2

$$2 = \frac{((S-1) + (B/A))}{(B/A - (S-1))}$$

giving B = C
A

$$2C - 2S - 2 = C + S - 1$$

$$C - 1 = 3S$$

So the number of symbols at which the declaration of constants reduces the useful redundancy is given by

$$S = \frac{(C-1)}{3}$$

Substituting for C

$$S = \frac{\frac{m \,.\, Nd}{Nc} \,.\, (Nc - 1)^{n}}{3} - 1$$

The expression above demonstrates that the number of symbols is a controlling influence in the use of declarations.

4.3.3 Errors in Variable and Constant Names

Declaring variable names protects the program from randomly distributed errors in the naming of variables and constants, with the resulting misinterpretation of the object being referenced.

The stream of characters forming the variable or constant is regarded as a message having

a) an equal probability of error in each character position, p, and

b) the errors in the character positions are independent.

Since typing errors can be introduced in the preparation of the program, conditions a) and b) above apply.

The probability of no errors in n-positions is given by

$$(1 - p)^n$$

and the probability of a single error in the n-positions is given by

$$np(1 - p)^{n-1}$$

The probability of k errors is given by the k-th term in the binomial expansion:

$$1 = [(1 - p) + p]^n = (1 - p)^n + np(1 - p)^{n-1} + \frac{n(n-1)}{2}p^2(1 - p)^{n-2} + \ldots + p^n$$

so the probability of exactly two errors, Pe2, is

$$Pe2 = \frac{n(n-1)}{2}p^2(1 - p)^{n-2}$$

4.3.4 Error-Protection of Variable and Constant Names

Hamming, [4], defined the concept of the 'Hamming distance' of a message as being 'the number of digit positions by which two states differ from each other'.

Hamming, [4], considers a message string of 0's and 1's as a point in a vector space of n-dimension where each digit is a value giving a co-ordinate in the space. Each vertex is a string of n 0's and n 1's. The space will therefore consist of 2^n vertices.

Since each vertex is a received message a single error moves the

message pointer along one edge of the space to an adjacent point. Hamming speculates that if every originating message was required to be a 'distance' of at least two edges away from any other message then any single error will move the message pointer along only one edge and thus indicate that the received message is illegal.

Assuming independent errors a minimum 'distance' of one character in the name makes the name unique. Whilst a name with a minimum 'distance' of two allows single errors to be detected.

The use of 'Hamming Distance' in variable and constant names suggests that each name varies from each other name by at least two characters then a single error would be detectable by the compiler when the name is referenced and would not transfer the context to another similar name.

The use of two-character names does not provide a sufficiently rich choice of names. Whereas, names of greater than two characters provide a rich choice of names and gives protection against single character errors.

From the argument presented the hypothesis is that variables and constants should be declared. However, when the number of symbols is extremely large the gross choice will add to the programmer's misunderstanding of the program. Since such a number of symbols is large it is concluded that the names of variables and constants should always be declared.

A controlling influence in the declaration of variables is the number of symbols used to declare the identifier. Recognisable identifiers can normally be constructed using the 26 letters of the English alphabet and the numerals 0..9. When each identifier consists of upto eight characters, then the number of non-repetitive permutations is P(36,8), which yields a large number of choices for the programmer so uniqueness is assured.

Since most languages allow the use of letters and numerals, each identifier in a program can vary from each other identifier by at least two characters and be protected against the effects of single typographical errors.

Shneiderman, [11], found that a typical typing error-rate amounted to 6.175%, or about 1 error in every 16 characters typed, suggesting that there is a 0.5 probability of an error in character streams of 16 characters or more, on initial input of the program. It follows that to limit the probability for error in a name, each name should be no longer than 15 characters and no less than two; the median value being eight characters which restricts the incidence of typing errors, provides a rich choice of uniqueness in the name whilst giving scope for a 'distance' of two.

4.3.5 An Experiment on the Use of Mnemonics

As part of the research for this thesis an experiment was conducted to examine the hypothesis that;

"In recognising the significance of a variable name the probability of an error is lower when the number of characters used to represent the name is at least two and less than 16".

The experiment was conducted at the 1983 Open University Summer Schools at the Universities of Warwick, Bath and York with 119 students taking part distributed as 23, 63 and 33 respectively. The participating students were volunteers mainly from the Technology Foundation Courses, though some volunteers came from a Second Level Technology Course. The experiment used a Superbrain QD microcomputer running a database package, dBase II.

The experiment required each volunteer to suggest eight mnemonics in response to eight descriptive texts presented on the computer screen seperately. After the eight mnemonics had been input by the volunteer, the volunteer was asked to repeat the

mnemonics for the same eight descriptions which were presented in a different order. Finally, the eight mnemonics suggested originally by the volunteer were displayed seperately to the volunteer who was asked to provide a description. A record was kept of each volunteer's name, venue, date, original mnemonic, second mnemonic and the description provided by the volunteer.

To maintain independence of results the volunteers were only permitted to ask questions regarding the purpose of the experiment and were not told their accuracy.

A correct answer was one where the description supplied by the volunteer conveyed the same information as the description supplied by the experimenter. Each volunteer took approximately 12 minutes to participate in the experiment.

The descriptive texts supplied to the volunteer were;

TIME OF DAY

LIQUID FLOW IN LITRES/MINUTE

WEIGHT OF PRODUCT IN TONNES

VALVE 8 POSITION

PERCENTAGE OF SULPHUR DIOXIDE IN THE ATMOSPHERE

AMBIENT TEMPERATURE

DISTANCE FROM THE VALVE CONTROLLER TO THE VALVE IN METRES

MOTOR SPEED IN R.P.M.

The database program is listed in Appendix 1 and the analysis is in Graph 2 and Table 2.

Table 2 contains the analysis of correct answers, incorrect answers and all answers by mnemonic length and mnemonic number with the frequency of the type of answer. The responses to the experiment were analysed and it was found that 70% of all answers were correct.

From the analysis the conclusion is that:

1. mnemonics of one character in length are prone to misinterpretation. All single character mnemonics were found to be incorrectly interpretted.
2. The mean character length for correct mnemonic usage was 7.88 characters with a standard deviation of 3.83.
3. The probability of a correct interpretation of a variable was found to be greatest when seven characters were used, with a probability of 0.125. The probability of a correct interpretation fell significantly when more than seven characters were used. The probability of a correct interpretation of less than 0.01 was found for variables of 15 characters or more.

It is concluded that the hypothesis was proved with one observation; that the ordering of the construction of the mnemonic may place a context on it. The context of a mnemonic may not be apparent to everyone using it and an experiment could be conducted to examine the optimum construction of mnemonics in order to reduce the risk of misinterpretation. So that the context could be recognisable to all users, a system of significant character positions could be used.

4.4 A Measure of Syntactic Structure and Error-Proneness for Application Programs

In an optimally encoded program there will be no information beyond that necessary to encode the program but in all computer programs there is additional information. The additional information is redundant to the main body of the program but serves to establish the programs context. The redundant information is called useful redundancy. Decision Content, Information Content and Redundancy are all properties of Information which have been explored in the context of error detection in computer programs. The method of calculating these properties is discussed and simple examples given. Example programs are analysed to illustrate the usefulness of Information Theory as an indicator of the amount of information required to declare a program.

Information Theory has its origin in the work of C.E.Shannon, [9], who published a paper in the Bell Systems Technical Journal concerning the communication of information through symbols. Information Theory can be used to indicate the amount of useful redundancy but from a safety aspect the concern is with the inverse of redundancy called Error-Proneness.

4.4.1 Halstead's Software Science Metrics and McCabe's Cyclomatic Number

Halstead's work on Software Science, [3], put the emphasis on quantitative measures of programs using a count of the number of operators and operands in the program.

Halstead's work presents a set of metrics which are derived from a basic set of measures. These measures are

n_1 = number of unique operators

n_2 = number of unique operands

N_1 = total occurrences of operators

N_2 = total occurrences of operands

Halstead's Vocabulary metric is given as

$$n = n_1 + n_2$$

and the Length metric by

$$N = N_1 + N_2$$

which is intuitively apparent.

The Volume metric uses the Vocabulary and the Length metrics to estimate the size of a program and uses bits as its dimension assuming a uniform binary encoding. The metric is given as

$$\text{Volume, } V = (N_1 + N_2)\text{Log}(n_1 + n_2) = N \text{ Log } n$$

The Volume will vary with the amount of coding required for the program but does not take any account of the frequency of occurrence of individual operators or operands.

The estimated length of the program is given as

$$N^{\wedge} = n_1 \text{ Log } n_1 + n_2 \text{ Log } n_2$$

The results of Halstead's work has found criticism, [10], principally because of the empirical foundation of the work. One problem with the work is that Halstead's measure of length remains constant regardless of the number of times individual operators or operands are used. Halstead's 'length' refers to the number of symbols being used and is not a measure of linear expansion.

McCabe, [7], suggested a measure of complexity based on Euler's formula for planar graphs given as

$$V(G) = \text{edges(e)} - \text{nodes(n)} + \text{no. of connected components(p)}$$

McCabe suggested that each node was a branch point in a program and the edges were the lines of flow between branches. McCabe stated that in a strongly connected graph of a program control network the value of p will be 2 and so reducing the equation to

$$V(G) = e - n + 2$$

The measure can be considered as a count of the number of branch points plus one. McCabe reduces the vocabulary of the program to branch points (nodes) and terminals with linear code sequences forming links (edges) between each node. The method corresponds to a measure of the number of choices presented at each branch point. Each branch point being represented by Log 2 bits and one terminator represented by Log 2 bits. McCabe's Complexity can be represented by nLog 2 + 1 = n+1 bits, where n is the number of branch points.

There is similarity between Halstead and McCabe in the way that a programs' complexity is regarded as a number of mental discriminations and represented by sums involving the expression nLog 2.

4.4.2 Information Theory

Information Theory measures information relating to the number of symbols in a message and the richness of choice of those symbols. In programs the amount of coding required depends on the problem, the programmer and the language used.

The symbols of a program are syntactic elements and the richness of choice depends on the syntax of the language being used, the constraints placed on the programmer through organisational standards, style or inexperience and the context in which the symbols are to be used within the program. Although McCabe, [7], and Halstead, [3], did not explicitly invoke Information Theory their results have a form similar to results involving Information Theory. However, McCabes work whilst well founded refers only to control structures. Halstead's work, based on empirical measures, neglects a number of features of programs and has been criticised for this by Shen and colleagues, [10].

Decision Content of a message, is defined in [2], also called the Maximum Information, H, is a logarithmic measure of the total

vocabulary from which a statement is chosen, assuming each event has an equal probability of being chosen.

Information Content of a message, defined in [2], given by I, is a logarithmic measure of the actual amount of coding required to represent the choice. Since the syntax of the language restricts the choice of symbols in certain character positions, there are two different measures.

Once the Decision Content and Information Content have been calculated it is possible to determine the Relative Redundancy. Relative Redundancy is the measure of the amount of information available but not required to represent the program. Relative Redundancy, r, is given by

$$r = (H - I) / H$$

and converted to a percentage.

4.4.3 Calculation of Plexus and Error-Proneness

When an object, whether it is a program, a calculation or whatever, is considered to be "complex" some assessment is made of what is commonly termed "complexity". In order to assess complexity account needs to be taken of two factors; the syntactic content and the semantic content, which combine in some way to give an individual view of complexity based on the individual's knowledge. If the object is a computer program then the syntactic content is a function of the language syntax whilst the semantic content is some function of the 'meaning' or function of the program. Though there are many expressions for software complexity none consider the semantics of the program and cannot, therefore, justifiably be called measures of complexity.

For any measure to be useful it must be finite and not subjective. Since the measures of complexity use only the syntactic element of an individual's view of complexity then these measures are

subjective judgements. Information Theory can be used to measure the syntactic features of a program but cannot assess the semantic content. In order to express a measurement of the syntactic element some term is needed, other than complexity.

When developing a program the syntactic choice available to the programmer can be considered as being a multi-nodal network with each node representing a choice. The network of nerves in the human body is greater than the network being discussed here for software, but the function of such a neural network is described by the term 'plexus'. A network such as the McCullock-Pitts neural network, where the primitive units are called neurons, was the network structure Von Neumann used to demonstrate that reliable machines can be built from unreliable components, [13].

The term 'Plexus' has been used in this thesis to refer to a measure of the syntactic choice being made from a network of choices to express a program.

A program written in a high-level language is constructed of two parts; the declarative part and the procedural part. The procedural part is influenced in its richness of choice by the syntax of the language. The declarative part of a program further restrains the vocabulary in addition to that already existing in the syntax. Both Halstead and McCabe excluded the declarative part from their calculations. Any measure of the syntactic structure of the program is influenced by the declarations. In the research for this thesis it was concluded that where declarations exist they should be included in the calculations concerned with choice.

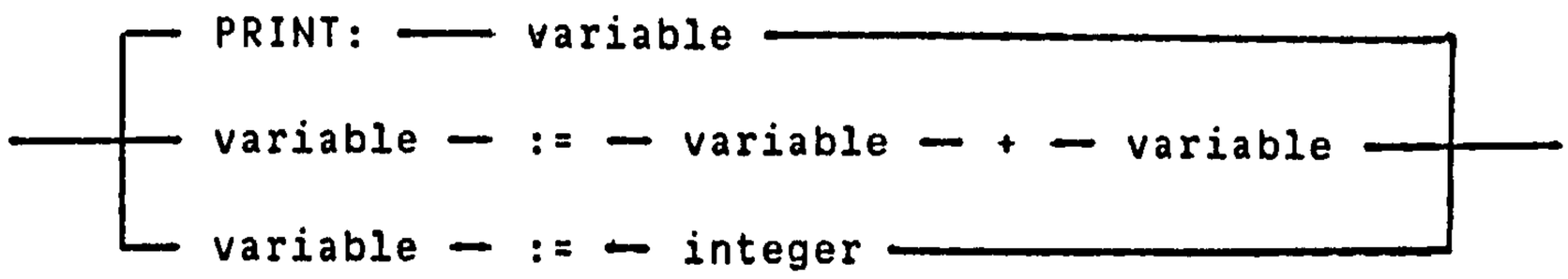
The syntax of a simple language in the form of a BNF formalism is shown in Figure 4.4.3.1.

Figure 4.4.3.1 BNF Syntax for the Example Language

```
<program> ::= PROGRAM <declare>
              BEGIN <body>
              END

<declare> ::= <declare statement> | <declare> <declare statement>

<declare statement> ::= VAR: <variable> ;

<variable> ::= <letter>

<letter> ::= A..Z

<body> ::= <procedure> | <body> <procedure>

<procedure> ::= <print> | <assign> | <add>

<print> ::= PRINT: <variable> ;

<add> ::= <variable> := <variable> + <variable> ;

<assign> ::= <variable> := <integer> ;

<integer> ::= <numeral> | <numeral> <numeral> |

<numeral> ::= 0..9
```

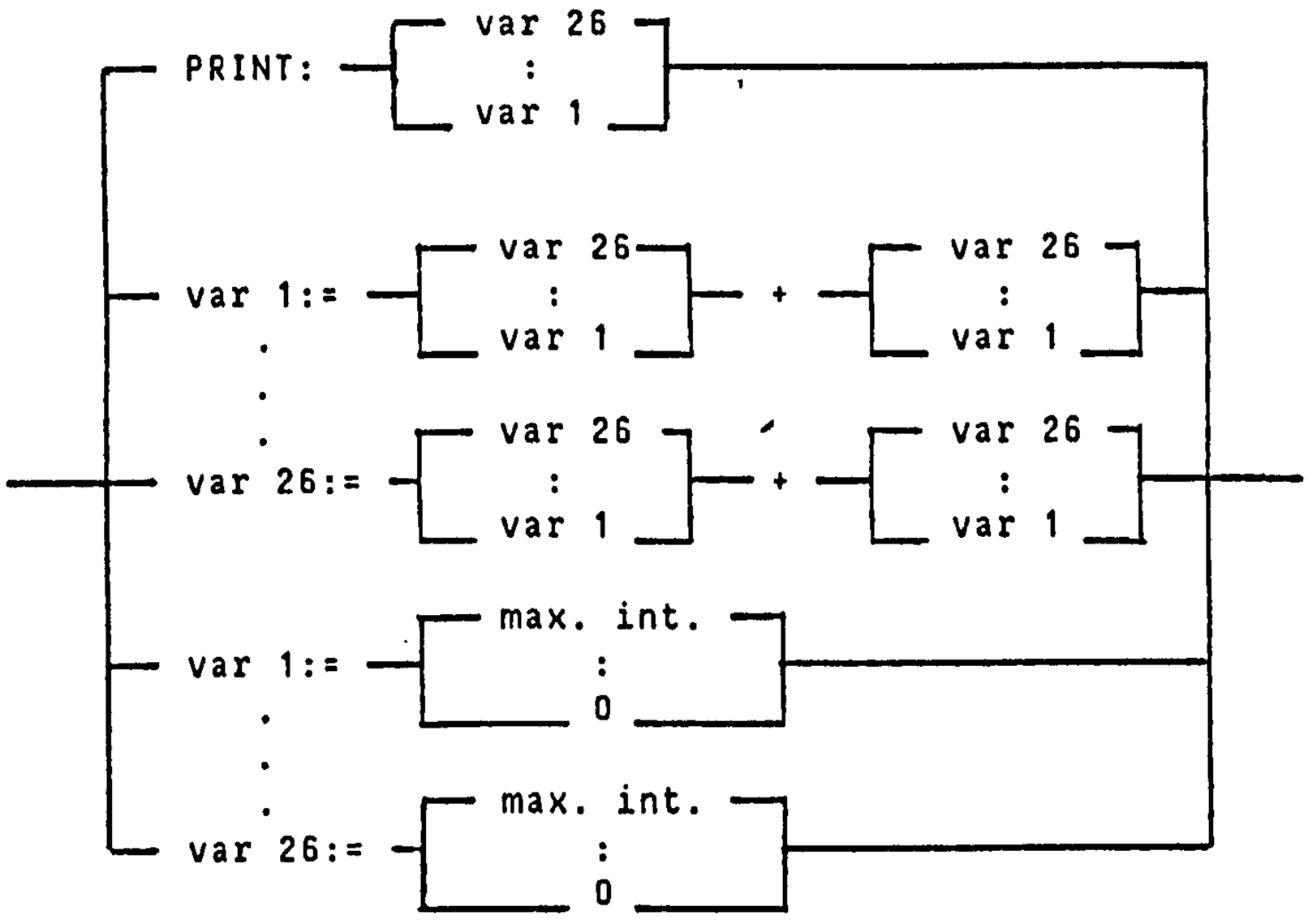
The diagrammatic style of presentation of a syntax was modified by Jensen & Wirth, [5], in the syntax description of the language Pascal. The procedure part of the BNF syntax can be drawn as a syntax diagram and expanded, after [5], as shown in Figure 4.4.3.2.

Figure 4.4.3.2 Syntax Diagram for the Example Language



The amount of choice available in the selection of a message from a restricted range of allowable variables is 26, (a..z), giving a set of symbols of 26. Integers are selected from a range of 0 to the maximum integer permissible for the implementation (MAXINT). Taking these restrictions into account the syntax diagram in Figure 4.4.3.2 can be mapped onto Figure 4.4.3.3 to reflect the range of choice available.

Figure 4.4.3.3 Revised Syntax Diagram for the Example Language



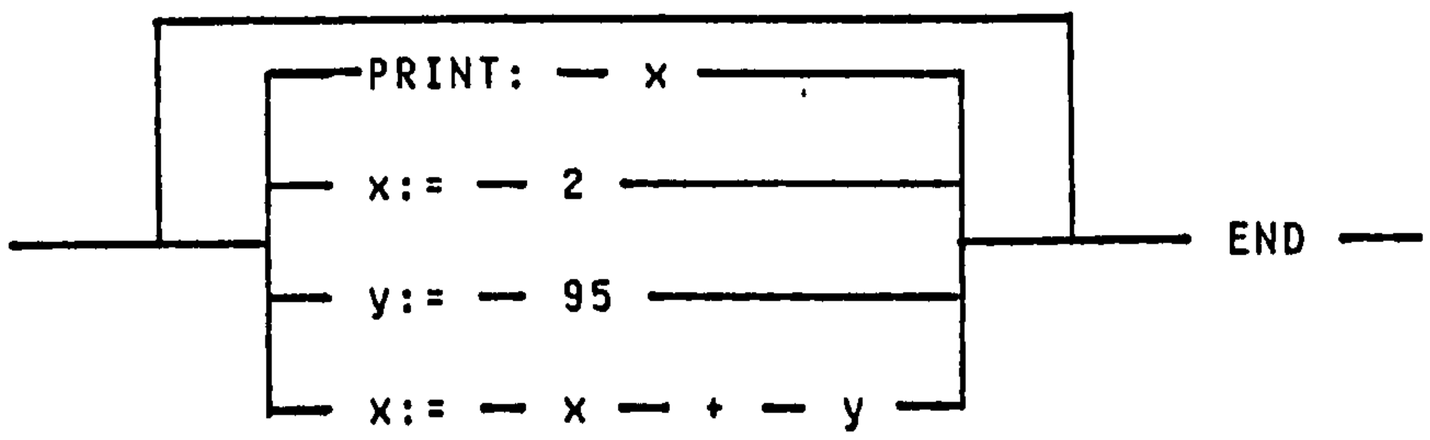
Measures based on the diagram, Figure 4.4.3.3, reflect properties of the language and not a particular program as there is nothing in the diagram to illustrate the structure of a particular program.

A particular instance of a program represents an ordered selection of items from the syntax diagram. An example of such a program (Program 1) written in the language is

```
PROGRAM
VAR: x;
VAR: y;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  PRINT:x;
END
```

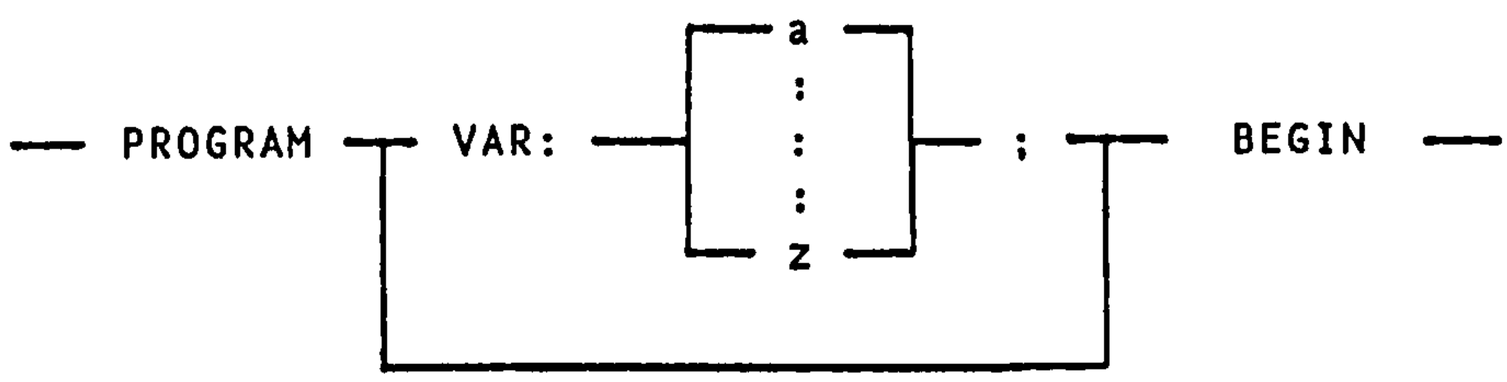
If the syntax diagram of Figure 4.4.3.3 is redrawn to reflect only the syntactic items used in the sample program, the syntax reduces to the diagram shown in Figure 4.4.3.4.

Figure 4.4.3.4 Syntactic Items of Sample Program



The declarations are considered in a similar way with a seperate syntax diagram being required. In the language being used the options available within the syntax for declarations can be represented diagrammatically, Figure 4.4.3.5.

Figure 4.4.3.5 Syntax Diagram of the Declarations



Keywords are syntactic necessities to the language and because of their certainty can be considered to have a probability of occurrence of one. Whilst keywords contribute to the Decision Content and Relative Redundancy they contribute nothing to the Plexus. Therefore when the diagram is redrawn to take account of the declarations used in the program it becomes that shown in Figure 4.4.3.6.

Figure 4.4.3.6 Revised Syntax Diagram of the Declarations

4.4.4 Method

Program 1 can be regarded as a message transmitted as a stream of symbols to the compiler (or to a person reading it), where a symbol is one or more characters in a defined syntactic group. There is a probability of occurrence associated with each individual

symbol and each symbol pair. Each symbol is assigned an information value according to the choice available and the constraints of the syntax. The method used concerns the syntactic structure and takes no account of the intended computation of the program, that is to say its semantic context.

Once a statement type is encountered in the message there are only three possibilities within the syntax of Figure 4.4.3.2; PRINT, assignment or addition. The positioning of the character P immediately identifies that a PRINT statement is to occur and the characters 'RINT' add no more information. Therefore the group of characters 'PRINT' can be treated as a single symbol. Similar to the keyword PRINT, the pair of characters ':=' and the group of characters forming the ordered set ':= ...+..' are considered to be single symbols forming a set of three statement types. From an information viewpoint Plexus represents the choice available.

The numbers 2 and 95 might be viewed as three numerals and a probability of 1/3 could be equally assigned to the numerals 2, 9 and 5 suggesting that 2 has a probability of 1/3 and that 95 has a probability of 1/3 * 1/3. The syntax diagram of Figure 4.4.4.1 is implied and allows for numbers other than 2 and 95. The choice is wider than the program allows.

Another view is that if 2 is chosen there are no further options and if 9 is chosen the only option is 5. Since there are only two options the probability of occurrence that can be assigned to the value 2 and 95 is 1/2., Figure 4.4.4.2.

Figure 4.4.4.1 A Method of Assigning a Probability of Occurrence to Three Numbers

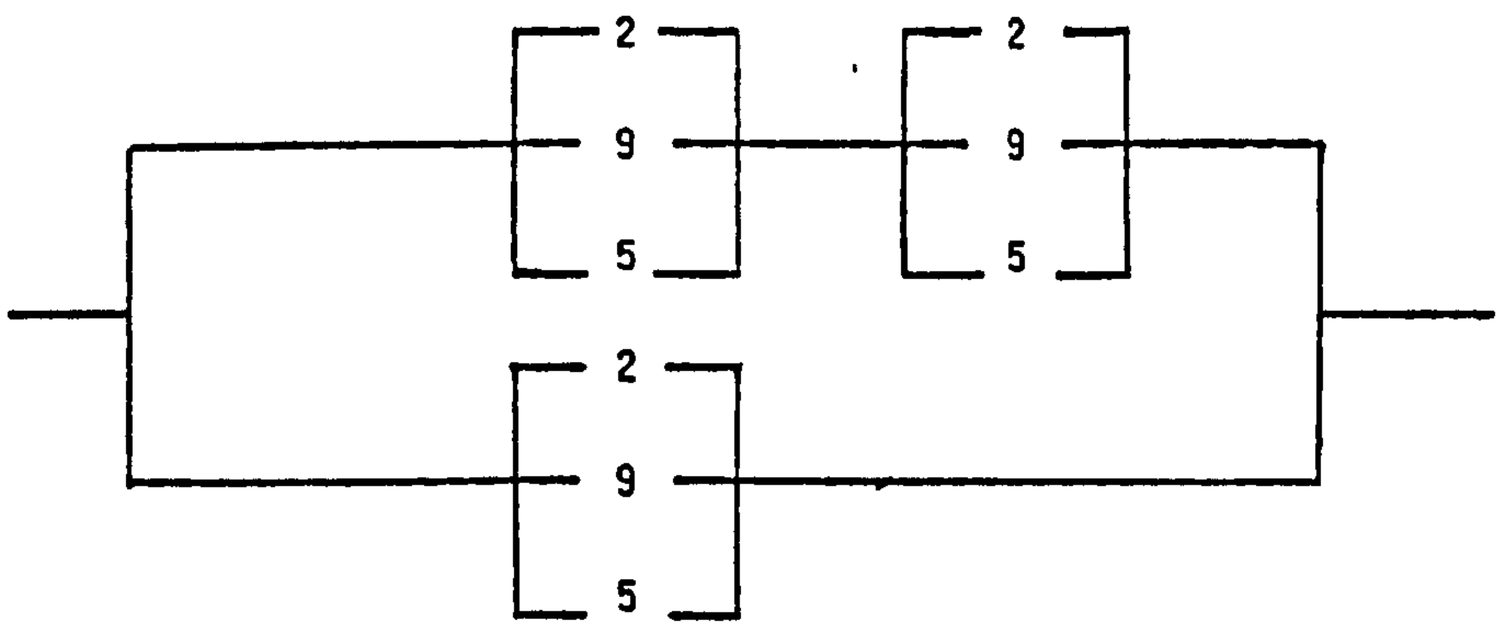
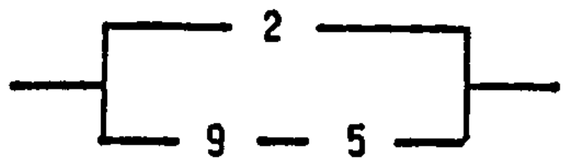


Figure 4.4.4.2 Another Method of Assigning a Probability of Occurrence to Three Numbers



There are many ways of viewing how the numbers can be represented and also any collection of syntactic elements. The selection is dependant on the view of what constitutes a syntactic entity. Ultimately the selection is a subjective judgement. However the syntax descriptions of languages are based on the designers views of syntactic entities and these form a basis for analysis.

In Program 1 the PROGRAM statement has a probability of one. PROGRAM is followed by the declarative part of the program. In the declarative part the choice exists between a VAR declaration of a variable and a BEGIN statement which punctuates the recursive declarations and signals the start of the procedural part. In the language only single letter variables, in the range a..z, are allowed. Therefore as each variable is declared the remaining choice is reduced by one.

From the program the following probability of each symbol being used on a line can be determined:

for declarations

$$P(\text{VAR:}) = 1/2$$

$$P(BEGIN) = 1/2$$

$$P(x) = 1/26$$

$$P(y) = 1/25$$

for statements, assuming all statements are equally probable

$$P(PRINT) = P(assignment) = P(addition) = 1/3$$

for variables

$$P(x) = 6/9 \qquad P(y) = 3/9$$

for values

$$P(2) = P(95) = 1/2$$

The probable occurrence of each line of the program is given by

$$P(PROGRAM) = 1$$

$$P(VAR:x) = 1/2 * 1/26$$

$$P(VAR:y) = 1/2 * 1/25$$

$$P(BEGIN) = 1/2$$

$$P(x:=2) = 1/4 * 4/6 * 1/2$$

$$P(y:=95) = 1/4 * 3/6 * 1/2$$

$$P(x:=x+y) = 1/4 * 4/6 * 4/6 * 2/6$$

$$P(PRINT:x) = 1/4 * 4/6$$

$$P(END) = 1/4$$

Since PROGRAM has a probability of unity it conveys no information and is omitted from the calculations. Each line of code has an Information Content expressed by the term

$$I = - \text{Log } P_i$$

assuming equal probabilities of occurrence of each statement type. Since the program is influenced by the number of changes that can be made a measure is required of the number of possibilities for change to be made to the program. If there was only one way to declare a program then the number of possible changes would be low but as the number of alternatives increases so does the number of possible changes. Therefore, the number of different ways that a program can

be declared must be reflected in some measure. Information Content measures a program's potential perturbations and in this context the principals of Information Content are regarded as the measure of Plexus. Plexus is given by P, so

for VAR:x; P = Log 2 + Log 26

for VAR:y; p = Log 2 + Log 25

for BEGIN P = Log 2

for x:=2; P = Log 4 + Log (6/4) + Log 2

for y:=95; P = Log 4 + Log (6/2) + Log 2

for x:=x+y; P = Log 4 + 2Log (6/4) + Log (6/2)

for PRINT:x; P = Log 4 + Log (6/4)

for END P = Log 4

The Plexus of Program 1 is given by the sum of the individual expressions as

P = 5Log 2 + 5Log 4 + Log 26 + Log 25

+ 4Log (6/4) + 2Log (6/2)

= 29.8 bits

Decision Content, from 4.4.2, is given by

H = NLog Cs

= 53Log 40

= 282.1 bits

where N = no. of characters required to express the procedure part

Cs = the size of the available character set

giving a Relative Redundancy for the whole program of

r = 89.4%.

The amount of information contained in the program and which is not required for optimal encoding is given as the Relative Redundancy. The scope that exists within the program for errors to be introduced is given as the Error-proneness, E, and is the inverse

of relative redundancy. The lower the error-proneness, the lower the probability that the program can be perturbed without detection.

$$E = 100\% - r$$

$$= 10.6\%$$

Applying Halstead's Estimated Length and Volume metrics to Program 1 the following measures are arrived at

$n_1 = 3$ $n_2 = 4$ $N_1 = 5$ $N_2 = 8$

$$V = 36.49 \text{ bits}$$

$$\hat{N} = 12.75$$

As another example, the program can be linearly expanded to give Program 2

```
PROGRAM
VAR:x;
VAR:y;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  x:=x+y;
  PRINT:x;
END
```

using the method discussed above the values for Program 2 become

$$H = 313.3 \text{ bits}$$

$$P = 34.54 \text{ bits}$$

$$r = 88.98\%$$

$$E = 11.02\%$$

Halstead's measures for Program 2 are

$n_1 = 3$ $n_2 = 4$ $N_1 = 7$ $N_2 = 11$

so the Volume is

$$V = 50.58$$

and the Estimated Length

$$\hat{N} = 12.75$$

Linear expansion of Program 1 and Program 2 has been included in

Appendix 2 along with comparison measures of Length, Volume, Decision Content, Plexus and Error-proneness. The measures have been plotted graphically and these are contained in Graph 3 to Graph 5.

In the method described the recursive declarations are punctuated by a BEGIN. The declarative part comprises two elements; the declaration statement and the variable used. The Plexus of the declarative part includes the number of permissible declaration statements from which a choice is made Nsd, the number of declarations used, D, plus the BEGIN statement and the amount of choice represented by each variable declared.

Assuming equiprobable choice, the Plexus of the declaration statements is the product of the number of statements used, D, and the Plexus of each statement, Log Nsd. The Plexus of the variables declared is the sum of the Plexus of a reducing set of available and undeclared variables, Vs - i.

The Plexus of the statements and the variables is summed to give the expression

$$D.\text{Log Nsd} + \sum_{i=0}^{(b-1)} b \text{ Log } (Vs-i)$$

where D = number of declarative statements used (including BEGIN)

Nsd = number of choices of declarative statements

Vs = number of possible variables

b = number of variables declared

The procedure part comprises control statements, variables and values. The Plexus of the procedural part takes account of the number of lines of control statements and the number of permissible control statements from which a choice is made. The frequency of occurrence of variables and values is summed giving the expression

$$Sc.\text{Log Nsc} + \sum_{i=1}^{b} Fvi \text{ Log } (Nv/Fvi) + \sum_{i=1}^{d} Fdi \text{ Log } (Nd/Fdi)$$

where Sc = number of control statements used (lines of code)

Nsc = number of types of control statements

Fvi = number of occurrences of variable (i) within the procedure part

Nv = number of occurrences of all variables in the procedure part

Fdi = number of occurrences of value i within the procedure part

Nd = number of occurrences of all values in the procedure part

d = number of values

The Plexus of the whole program can be considered as being the sum of the plexus of the declarative part and the procedural part, given as

$$P = D.\text{Log } Nsd + \sum_{i=0}^{(b-1)} b \text{ Log } (Vs-i) + Sc.\text{Log } Nsc + \sum_{i=1}^{b} Fvi \text{ Log } (Nv/Fvi)$$

$$+ \sum_{i=1}^{d} Fdi \text{ Log } (Nd/Fdi)$$

Conventional programming languages, unlike the example language used so far, include relational operators and allow the use of external routines called procedures and functions. Procedures and functions are one type of external call which are implicitly declared and handled in the same way as any other declaration, that is to say by considering their frequency of occurrence.

Considering the frequency of external calls, pc, and relational operators, op, in the same way as values adds the term

$$\sum_{i=1}^{p} Fpci \text{ Log } (Np/Fpci) + \sum_{i=1}^{q} Fopi \text{ Log } (Nop/Fopi)$$

where Fpc = number of occurrences of external calls (i) within the procedure part

Np = number of all external calls in the procedure part

Fop = number of occurrences of relational operator (i) in the procedure part

Nop = number of all relational operators used

p = number of individual external calls

q = number of individual relational operators

The expression for the Plexus' of the whole program is the sum of each of the terms combines to give the expression

$$P = D.\text{Log } Nsd + \sum_{i=0}^{(b-1)} b \text{ Log } (Vs-i) + Sc.\text{Log } Nsc + \sum_{i=1}^{b} Fvi \text{ Log } (Nv/Fvi)$$

$$+ \sum_{i=1}^{d} Fdi \text{ Log } (Nd/Fdi) + \sum_{i=1}^{p} Fpci \text{ Log } (Np/Fpci) + \sum_{i=1}^{q} Fopi \text{ Log } (Nop/Fopi)$$

The term for variables, values, external calls and relational operators is the same so the expression can be simplified to

$$P = D.\text{Log } Nsd + \sum_{i=0}^{(b-1)} b \text{ Log } (Vs-i) + Sc.\text{Log } Nsc$$

$$+ \sum_{i=1}^{m} \sum_{j=1}^{n} Fi,j \text{ Log } (Ni/Fvi,j)$$

where i = symbol type: variable, digit, procedure and relational operator

j = symbol

i,j = symbol (j) of type (i)

By using the above equation on sample programs written in trivial languages the method can be examined further.

Language 1.

The first simple program has been written in a Pascal-like language;

```
PROCEDURE product (x:REAL,y:REAL)
VAR i:INTEGER;
BEGIN
   FOR i = 1 TO 5 DO
   BEGIN
      x:=x*y;
   END
END.
```

Various measures can be calculated using the expressions developed. Figures for Decision Content, Plexus, Relative Redundancy and Error-proneness for the whole program are;

$$H_t = 88 \text{Log } 64 = 528$$

$$P_t = 41.8$$

$$r_t = 0.934 = 92.1\%$$

$$E_t = 7.9\%$$

The actual calculations are given in Table 3.

Language 2.

The second program is written in a FORTRAN-like language.

```
SUBROUTINE (X,Y)
DO 20 I = 1,5
20 X=X*Y
RETURN
```

and for the whole program the measures are

$$H_t = 43 \text{Log } 64 = 258$$

$$P_t = 38.0$$

$$r_t = 0.853 = 85.3\%$$

$$E_t = 14.7\%$$

These trivial examples show that whilst Language 1 has a Plexus of 41.8 and an Error-Proneness of 7.9%, Language 2 has a Plexus of 38.0 and an Error-Proneness of 14.7%. These measures are absolute for each language and comparisons are not easily made between languages.

Having seen the method applied to simple programs written in trivial languages the method can be applied to non-trivial languages, like Pascal, Basic and Fortran, using example programs.

4.4.6 Example Programs

Pascal Version

The program has been compiled using UCSD PASCAL. The program and syntax rules have been taken from Jensen & Wirth, [5].

```
PROGRAM graph2 (output);
CONST d=0.0625;
      s=32;
     h1=34;
     h2=68;
       c=6.28318;
    lim=32;
VAR i,j,k,n:INTEGER;
    x,y:REAL;
    a:ARRAY[1..h2] OF CHAR;
  BEGIN
  FOR j:=1 TO h2 DO a[j]:=' ';
  FOR i:=0 TO lim DO
  BEGIN
    x:=d*i;
    y:=EXP(-x) * SIN(c*x);
    a[h1]:=':';
    n:=ROUND(s*y) + h1;
    a[n]:='*';
    IF n < h1 THEN k:=h1
              ELSE k:=n;
    FOR j:=1 TO k DO WRITE(a[j]);
    WRITELN;
      a[n]:=' '
  END
END.
```

Using the expression developed and the data contained in Table 3 the Plexus for the whole program is calculated as

$$H_t = 312 \text{Log } 96 = 2054.5$$

$$P_t = 483.0$$

$$r_t = 0.769 = 76.5\%$$

$$E_t = 0.235 = 23.5\%$$

If the measures are to be used for comparison with similar programs written in languages not requiring declarations, then the calculations should omit the declaration part. The figures for the program become

$$H_t = 161 \text{Log } 96 = 1060.2$$

$$P_t = 248.2$$

$r_t = 0.766 = 76.6\%$

$E_t = 0.234 = 23.4\%$

BASIC Version

A program written as a BASIC equivalent of the Pascal program used, using Microsoft BASIC-80, could be

```
 10 DIM A$[68]
 20 FOR J=1 TO 68
 30 A$[J]=CHR$(32)
 40 NEXT J
 50 FOR I=0 TO 32
 60 X=I*.0625
 70 Y=EXP(-X)*SIN(6.28318*X)
 80 A$[34]=CHR$(58)
 90 N%=32*Y+34
100 A$[N%]=CHR$(42)
110 IF N%<34 THEN K=34 ELSE K=N%
120 FOR J=1 TO K
130 PRINT A$[J];
140 NEXT J
150 PRINT
160 A$[N%]=CHR$(32)
170 NEXT I
180 END
```

for the whole program

$H_t = 216\text{Log } 68 = 1314.9$

$P_t = 283.7$

$r_t = 0.784 = 78.4\%$

$E_t = 0.216 = 21.6\%$

If the declaration part is not included in the calculations the figures become

$H_t = 206\text{Log } 68 = 1254.0$

$P_t = 248.9$

$r_t = 0.802 = 80.2\%$

$E_t = 0.198 = 19.8\%$

FORTRAN Version

The following program is a FORTRAN version of the Pascal program and has been prepared using Microsoft FORTRAN-80.

```
      PROGRAM GRAPH2
      LOGICAL M(68)
      DO1J=1,68
    1 M(J)=' '
      DO2I=0,32
      X=I*0.0625
      Y=EXP(-X)*SIN(6.28318*X)
      M(34)=':'
      N=32*Y+34
      M(N)='*'
      IF(N.LT.34)K=34
      IF(N.GE.34)K=N
      WRITE(1,3)M
    3 FORMAT(1H ,68A1)
    2 M(N)=' '
      END
```

for the whole program

$$H_t = 181 \text{Log } 68 = 1101.8$$

$$P_t = 304.1$$

$$r_t = 0.724 = 72.4\%$$

$$E_t = 0.276 = 27.6\%$$

and omitting the declarations

$$H_t = 156 \text{Log } 68 = 949.6$$

$$P_t = 269.3$$

$$r_t = 0.716 = 71.6\%$$

$$E_t = 0.284 = 28.4\%$$

It was stated earlier that the metric, Plexus, cannot be used to compare programs written in different languages unless a measurement is available to take account of the richness of vocabulary within the languages. However, a meaningful comparison can be made if the declarative part is omitted from the Plexus equation.

4.4.7 System Plexus

Having addressed the problem of measuring the Plexus of single programs the expression for the Plexus of two or more programs coupled together in some way, whose individual Plexus is known, can be addressed. Assuming that each module will be activated through some kind of operating system, then any communication between the modules is equally probable. Therefore the effects of uni-directional communications between modules is ignored.

As an example of how the Plexus of two modules can be combined to give the System Plexus, take two modules, A and B, whose Plexus is known.

The combining rule for two measures of information relating to two equally probable entities m and n, is

$$- \text{Log } (m + n)$$

and the combined value for A and B is

$$- \text{Log } (A + B)$$

however, A and B are logarithmic values so the expression for System Plexus, Ps, becomes

$$Ps = - \text{Log } (2^{B} + 2^{A})$$

$$= - B - \text{Log } (1 + 2^{A-B})$$

As an approximation the Log series is expanded to

$$\ln (1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} \ldots\ldots\ldots$$

for logarithms of base 2 the expression is

$$\text{Log } (1 + x) = \frac{\ln (1 + x)}{\ln 2}$$

$$= \frac{2^{A-B}}{\ln 2}$$

sustituting, an aproximation is

$$\text{System Plexus, } Ps = - B - \frac{2^{A-B}}{\ln 2}$$

Assuming module A has a Plexus of 1 and module B has a Plexus of 3, then the actual value of the System Plexus, Ps, is

$$Ps = - \text{Log } (2^{B} + 2^{A}) = 3.32$$

and as an approximation, ^Ps, is

$$\hat{}Ps = - 3 - \frac{2^{-2}}{\ln 2} = 3.36$$

Combining three modules A, B and C whose Plexus figures are individually given as 1, 3 and 5 respectively, then the System Plexus, Ps, is

$$Ps = \text{Log } (2^{C} + 2^{B} + 2^{A})$$

$$= \text{Log } (35)$$

$$= 5.13$$

The general rule for combining two or more modules into a system when the individual Plexus is known can be expressed as;

$$Ps = \text{Log} \sum_{p=1}^{n} 2^{p}$$

where Ps is the System Plexus

p is the individual module Plexus

n is the number of modules being combined.

4.4.8 Discussion

The concern in this section of Chapter 4 has been the amount of information required to prepare a program and developing a method of measuring syntactic complexity, called Plexus. If the Plexus for each of the example programs is compared, omitting declarations to allow comparison, with the Fallibility Index from Chapter 4.2 and the number of characters in the program, it can be shown that as the number of characters increases so the Plexus and Fallibility Index

decreases, Figure 4.4.6.1. A comparison of the values derived from the proposed metric, Plexus, and those of Halstead for the same programs is contained in the Graphs 6 to 9.

Figure 4.4.6.1 Comparison of Fallibility Index and Plexus

Language	Nc	Fallibility Index (%)	Plexus
FORTRAN	187	18.5	304.1
(omitting the declarations)			
	174	19.4	269.3
BASIC	234	10.5	283.7
(omitting the declarations)			
	197	12.2	248.9
PASCAL	334	3.7	483.0
(omitting the declarations)			
	233	3.7	248.2

4.5 References

[1] Boehm, B.W., Software and Its Impact: A Quantitative Assessment, Datamation, May, 1973

[2] B.S.I., Glossary of Terms in Data Processing, BS3527, Part 16

[3] Halstead, M., Elements of Software Science, North-Holland, 1977

[4] Hamming, R., Coding and Information Theory, Prentice Hall,1980

[5] Jensen, K., & Wirth, N., Pascal: User Manual and Report, Springer-Verlag, 1979

[6] Kopetz, H., Software Reliability, MacMillan Computer Science Series,1979

[7] McCabe, T.J., A Complexity Measure, IEEE Transactions on Software Engineering, Volume SE-2 Number 4, December, 1976, pp. 308 - 320

[8] Peters, L.J., Software Design: Methods & Techniques, Yourdon Press, 1981

[9] Shannon, C.E., A Mathematical Theory of Communication, Bell System Technical Journal, 1948

[10] Shen, V.Y., Conte, S.D. & Dunsmore, H.E., Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support,IEEE Transactions on Software Engineering, Volume SE-9, Number 2, March 1983, pp.155-165

[11] Shneiderman, Software Psychology, Winthrop, 1980

[12] Sommerville, I., Software Engineering, Wiley, 1982

[13] Von Neumann, J., Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components, in Automata Studies, Princeton University Press, 1956, pp. 43 - 98

CHAPTER 5

A Method of Conducting a Safety Audit on Software

The assessment of the level of risk resulting from the use of software as a control element has been advanced in previous Chapters. The possibility of introducing errors at each stage of the development of the software has been discussed along with a basis for measuring the possibility for such errors.

As part of the research for this thesis a commercially available product was examined and the methods and measures discussed earlier in this thesis were applied in an attempt to determine the applicability of such techniques.

Based on the research for this thesis, this Chapter presents an argument for using a set of procedures called Software Safety Audits to assess the software used in industrial-based control systems. Such a Software Safety Audit would normally be conducted against a set of criteria considered to be acceptable by the designer and by the User but presently there is no standard criteria, so the assessment undertaken has to stand alone. The Chapter discusses certain aspects of the Software Safety Audit carried out on the product examined.

The analysis of the product can only be discussed briefly as the commercial confidence of the company, and of the product, must be maintained.

5.1 The Software Safety Audit

There is a probability of errors being created at each stage of the software development cycle. It is not practicable to remove all these errors from the software before the software is considered by the designer to be ready for operational use in a safety-related process or product. Since all software will have errors remaining after the testing stages have been completed then there needs to be some form of final check aiming to identify any unsafe aspects of the software. Such a set of checks on the software is called a Software Safety Audit.

The purpose of a Software Safety Audit is to give some measure of the operational safety when using the software and to make some subjective judgement whether the software is, or can be made, operationally safe, given the safety criteria being used. If the software is not, or cannot be made, operationally safe then the software should not be used until the software has been modified to meet the criteria or the User is prepared to acknowledge and accept the features identified as being unsafe.

In assessing the operational safety of the software it must be remembered that the software is only one component of a total system which includes the software, the computer hardware, the plant and associated hardware and the Operating Personnel. Once an assessment has been made of the software then the software should be maintained at the same safe state to which it was assessed.

There are three elements to analyse in the Software Safety Audit; the software, the system and the integrity of the system. The three elements form three levels of assessment in which the software element is the lowest level.

A Software Safety Audit could be developed starting with the software, followed by the system and its integrity. The safety

assessment of the software element can be considered to assist the person conducting the audit of the whole system. However, the assessment of the whole system assists the person conducting the audit of the software element and provides an insight into the functioning of the whole system.

An approach to Software Safety Audit needs to consider the software and the whole system and once the Software Safety Audit has assessed the software to be safe a mechanism of ensuring the continued integrity of the system should follow.

5.1.1 The Need and Form of a Software Safety Audit

The Health and Safety at Work etc Act 1974, [2], Section 6 (1), states "It shall be the duty of any person who designs, manufactures, imports or supplies any article" to meet certain requirements towards safety and, specifically, towards the testing of the article. Section 7 of the Act requires "every employee while at work" to take due regard for health and safety of himself and of others. Section 8 states "No person shall intentionally or recklessly interfere with or misuse anything provided in the interests of health, safety or welfare". The Act, therefore, makes all parties involved in the design, manufacture, installation and operation of safety-related equipment responsible for ensuring the health and safety of those affected by its operation.

The aim of a Software Safety Audit is to give a measure of the operational safety involved in using the software in the system being audited and to assist in rendering as impotent any errors detected in the software or the system which might otherwise jeopardise safety. In accordance with Sections 6, 7 and 8 of the Health and Safety at Work Act the interpretation is that a Software Safety Audit may be requested by

- a manufacturer of a new product at the design stage or before

it is offered for sale

- by the supplier of a product before it is supplied to the end-user
- by an end-user who requires a Software Safety Audit to be carried out on each stage of the development of a contracted system or
- by an end-user before the contracted system is permitted to become operational.

Whoever requests the Software Safety Audit is called the Requesting Authority and those undertaking the Safety Audit are called the Auditors.

The fact that an item of software and its system has been assessed for safety before becoming operational can afford the designer some measure of confidence in its use. If the designer has the knowledge that the software was ultimately found to be safe then the confidence will be high. When the designer has the knowledge that a number of iterations were necessary before a safe assessment was achieved then confidence in the software may be reduced.

The user's knowledge of a Software Safety Audit will influence confidence in the software. When a Software Safety Audit is carried out on the software before it becomes operational then the user's confidence will be high even though a number of iterations may have been necessary before being assessed as safe. When a Software Safety Audit of the software is undertaken retrospectively after being operational, even though the software has been assessed to be safe, then the user's confidence in the software and its system will be less since the inference is that there is cause for concern by the manufacturer or supplier. The confidence of the designer and the user in the software is influenced by knowledge of the Software Safety Audit having been carried out and this knowledge may have

commercial consequences. Therefore, a Software Safety Audit ought to be conducted in confidence with the control of the knowledge of a Software Safety Audit being conducted and the use of the results of the Software Safety Audit resting in the hands of the Requesting Authority. If the Requesting Authority is a commercial concern then the information on the Software Safety Audit may be suppressed. Alternatively, if the Requesting Authority is not a commercial concern, for instance a Trade Union or a group representing the public interest, then the information may be made public in order to cause some action to be taken, for example a public inquiry.

The ordering of the assessment procedures used for the Software Safety Audit can be fixed by legislation, but the order should be changeable such that the assessment procedures can take account of research developments.

On finding an error or inconsistency in the software or the system a change may be made. The change may alter some feature which had been checked previously. To retain credibility the Software Safety Audit should repeat the preceeding procedures. To do any other may lower the credibility of the Software Safety Audit in the view of the Requesting Authority. The number of instances where it is necessary to repeat a number of the earlier procedures can be minimised by the ordering of the Software Safety Audit.

The ordering of the procedures used in the assessment for this research was considered to give an increased probability of finding errors early in the assessment process and to reduce the number of revisions required as a result of finding errors.

The structuring of the software has been shown to influence safety, so before any other assessment can be carried out the structuring of the software should be examined.

The Software Safety Audit proposed in this Chapter starts with

an assessment of the structure of the software. Software Fault Tree Analysis and Software Event Tree Analysis methods are used later in the Audit to examine for particular failure conditions. Once the software has been assessed at the system level by means of Fault Tree Analysis and Event Tree Analysis, the software is assessed for the perturbation of variable and constant names, followed by a calculation of the Fallibility Index and the Plexus. The final activity of the Software Safety Audit would normally be to assign an Integrity Lock to each program. This was not possible in the assessment for this thesis as only the listings were available for the assessment and not the actual programs.

The ordering of the assessment procedures used in the Software Safety Audit for this thesis was:

Software Structural Analysis

Fault Tree Analysis

Event Tree Analysis

Perturbation of Variable Names

Calculation of the Fallibility Index

Calculation of the Plexus

5.2 An Example Software Safety Audit

To demonstrate the method of conducting a Software Safety Audit a genuine industrial-based application needed to be tested rather than one artificially created for the purpose. It was considered necessary to approach a manufacturer or supplier of electronic products who was known to incorporate a computer with some application software into a commercially available product which was intended for use in a safety-related control application.

There is a wide range of products on the market which can be shown to meet the criteria. A major manufacturer of medical electronic products was approached and it was agreed with the manufacturer that a particular product could be assessed provided that confidentiality was maintained at all times regarding the identity of the company, the product and the outcome of the assessment.

For reasons of commercial confidence the manufacturer must remain anonymous in this thesis. So that a description can be given of the product assessed, the product will continue to be referred to simply as 'the product' meaning the commercial product and 'device' to mean particular components of the product.

5.2.1 The Product

The product is used extensively in the medical service, particularly where hospitalisation of the patient is necessary, for instance Intensive Care Units and Surgical Wards. Earlier versions of the product have been in use for many years in the United Kingdom and abroad employing hard-wired logic to monitor and control the products function. Due to market pressures the company chose to develop the product further and to incorporate a microprocessor into the product to take over the control functions from the hard-wired logic. The product's principle function is to monitor and control

flow rates of drugs at a desired value within a medically acceptable tolerance level. The product was suitable for the trial application of a Software Safety Audit as it incorporated a microprocessor, it was used in a safety-related situation and the amount of application software involved was not too large for assessment by manual methods; approximately 4000 bytes of computer memory.

A full description of the product is not possible without breaking the confidence of the company. However, a brief summary can be given.

The product is used to monitor and control the flow rate of drugs to a patient and is made operational by an action, on the part of the nursing staff, indicating to the product the desired flow rate. The flow rate is indicated by means of a set of thumb-wheel switches and is input to the microprocessor when the 'START' button is pressed. A device then adjusts the flow rate to the desired flow rate value. Once the desired flow rate is reached a device monitors the flow rate to ensure that it remains within the medically accepted tolerance. If at any time during the operation of the product an alarm condition is reached then a set of actions are available to the microprocessor ranging from the sounding of alarm buzzers to rapidly closing off the flow. The product can be stopped or reset by pressing the 'STOP/RESET' button. The flow rate sensor is an optical device placed in the drug supply line.

Physically the product contains two decade thumb-wheel switches, 'START' and 'STOP/RESET' buttons and four status indicators all mounted on the front facia panel. Extending from the device are the mains electricity supply cable and the monitoring sensors. Within the product there is an electro-mechanical device designed to rapidly close off the flow rate in the case of a particular alarm.

The product uses a commonly available 8-bit microprocessor with

the application software being written in the Assembler language for that microprocessor, though the initial development of the software was written in Pascal. The use of Pascal was abandoned as the management of the company felt that Assembler language would yield a more compact object code and would have greater speed of operation. The software development was done on a microprocessor development system.

In discussion with the company it emerged that the software had been developed by competent electronic engineers, who had no formal training in software engineering, without the benefit of any form of design specification. Software modules were written as it was felt necessary to meet the overall design objectives set by the Designer. Testing was limited to testing what the company called the "functionality of the modules and inter-module communication".

5.3 Analysis of the Example Application

In discussion with the company, the company suggested that the criteria for the Software Safety Audit should be to examine the system for the catastrophic failure "an excessive flow rate without an alarm being raised such that life could be put at risk". It was the view of the company that this was the maximum credible incident for the product and this failure condition was used as the basis for the assessment.

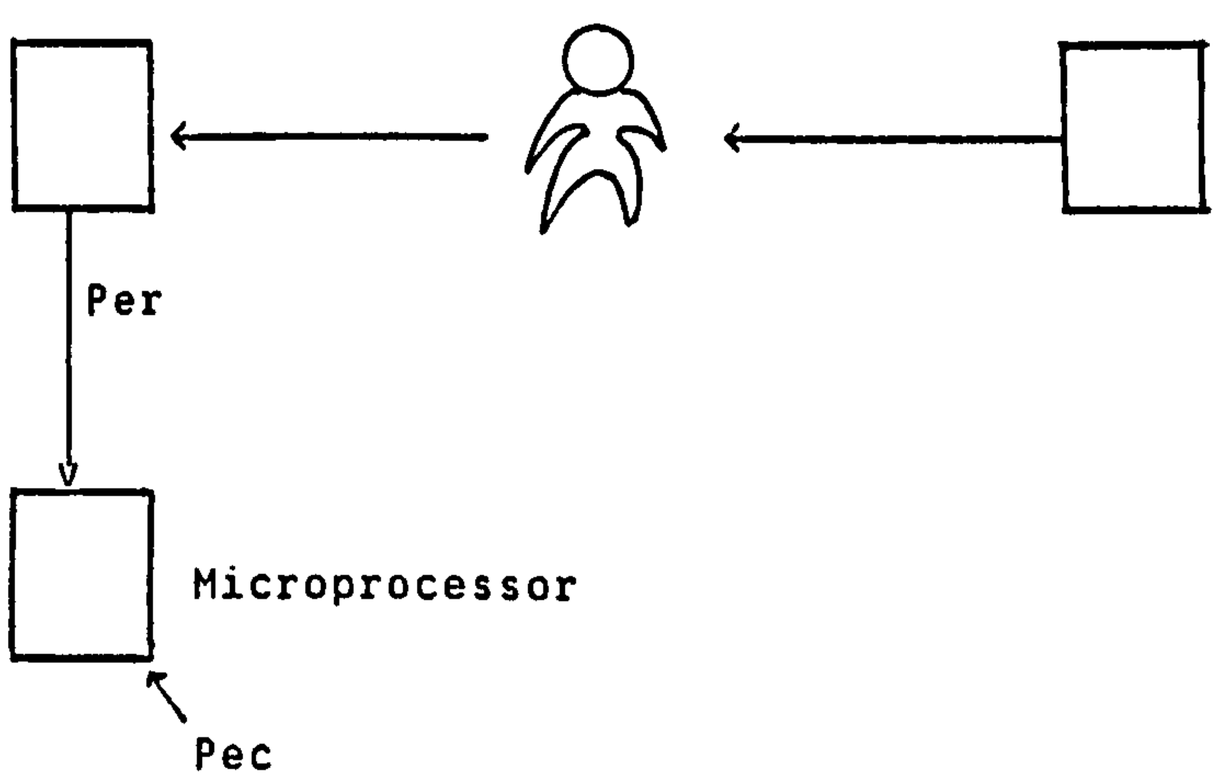
5.3.1 Input Mechanism

The product assessed relied upon the medical staff, most likely a nurse, dialling the desired flow rate on a pair of decade thumb-wheel switches. The flow rate was to be in the range 1 to 99 with no display of the value read by the microprocessor from the thumb-wheel switches.

The absence of a microprocessor-driven display showing the value read by the microprocessor may be considered to be unsafe as there is a probability that the system will incorrectly read the desired value. As a means of entering the desired flow rate a number of options were available to the designer in addition to thumb-wheel switches. Two such options were thumb-wheel switches with some display or a display counter. Considering the three methods of input mentioned as being available to the designer it can be assumed that the probability of an error caused by the nursing staff incorrectly reading the displayed value and the probability of error in computing the value read, does not change. As an exercise the use of thumb-wheel switches, or one of the two options mentioned, has been studied to see how a Software Safety Audit can be carried out on the equipment. The exercise also shows how the inter-relationships between the software, the hardware and people is considered in a Software Safety Audit.

a) Thumb-wheels alone

In this scheme the nurse enters the desired value on the thumb-wheel switches which are then read by the microprocessor. If the nurse has dialled the wrong value then it is expected that the error will be noticed by the nurse and corrected by entering the correct value. The scheme can be represented diagrammatically as



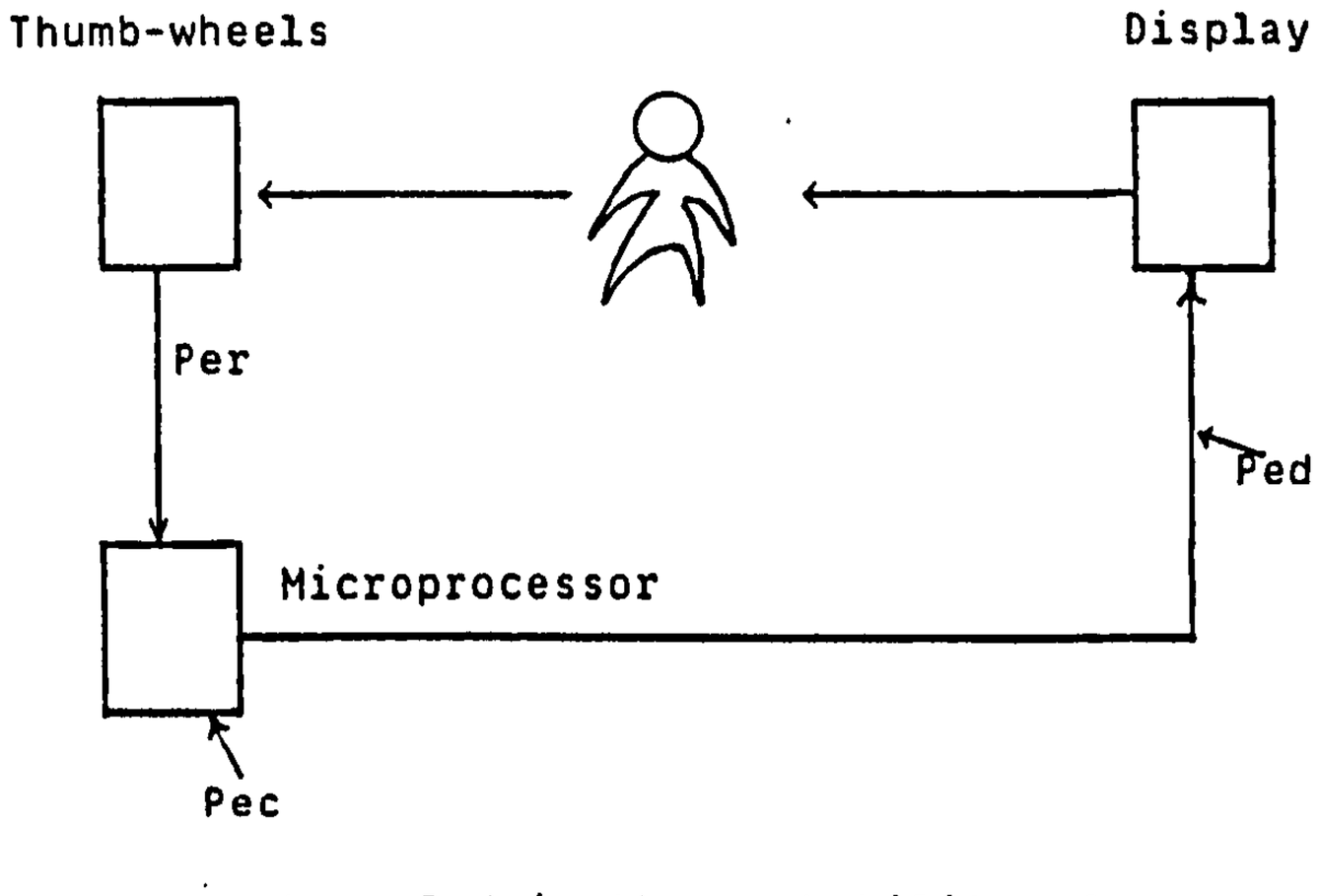
where Per is the probability of an error in the thumb-wheel switches when the input is read and Pec is the probability of an error in the microprocessor when computing the value read

The reliability of the system is therefore influenced by the probability of an error when the value is read by the microprocessor from the thumb-wheel switches.

b) Thumb-wheels with display

In this scheme the nurse enters the desired value on the thumb-wheel switches which are then read by the microprocessor. The microprocessor interprets the value and displays it to the nurse by, say, a Liquid Crystal Display. Any errors are observed by the nurse noting a differance between the thumb-wheel switches and the displayed value. The correction is made by the nurse altering the dials on the thumb-wheel switches accordingly. The scheme can be

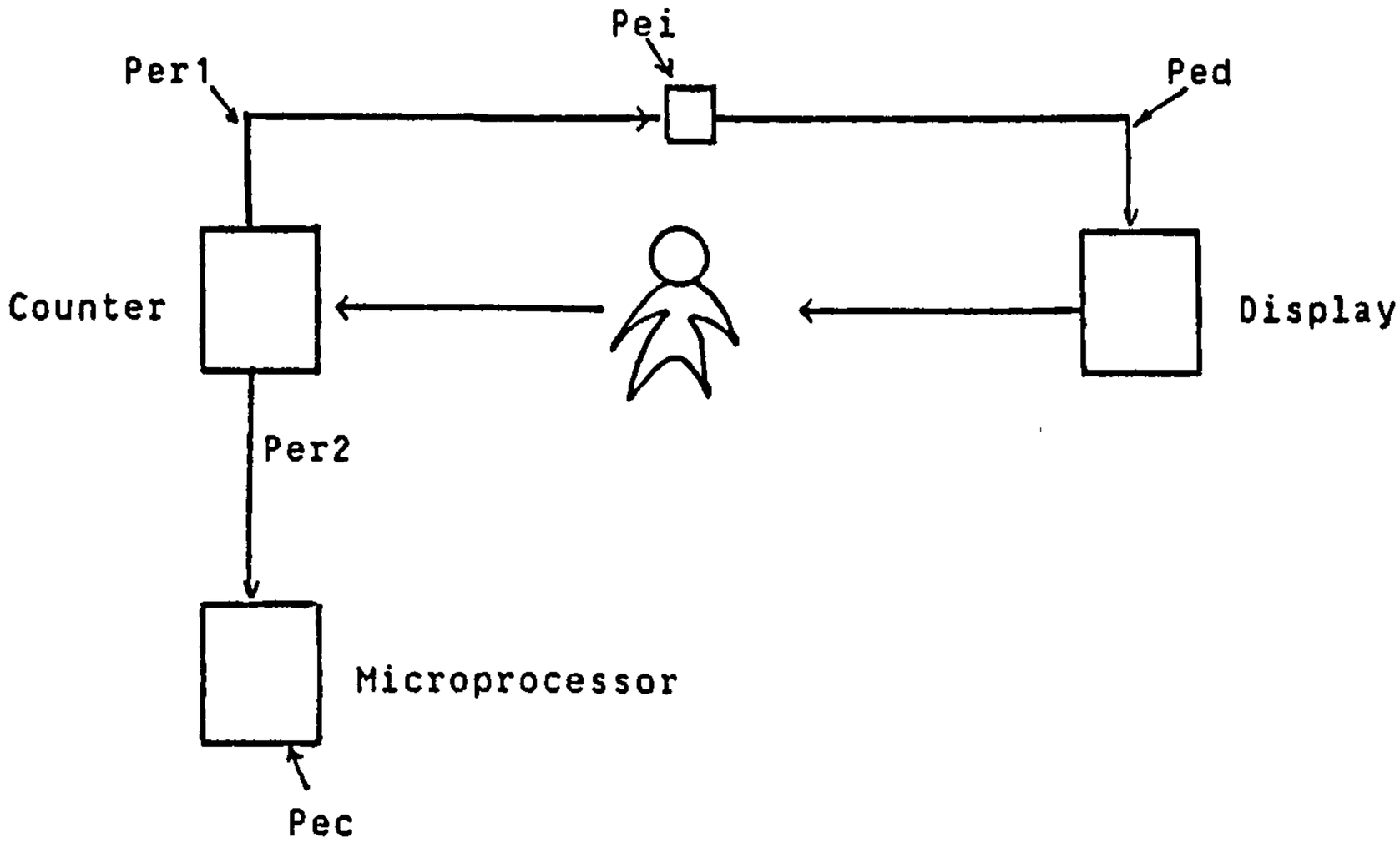
represented as:



where Ped is the probability of an error in displaying the value read by the microprocessor

c) Display Counter

A display counter would work by the nurse pressing a button according to the desired value. The number of times the button was been pressed would be accumulated by some internal electronics which would display the accumulated value to the nurse by, say, a Liquid Crystal Display. Once the desired value was reached an 'Accept' button would be pressed by the nurse causing the microprocessor to read the value. Such a scheme can be represented as



where Pei is the probability of an error in the internal electronics.

If it is assumed that when the microprocessor reads the desired value the probability of an error is the same for each of the three options discussed, then the use of thumb-wheels alone has the lower risk attached to it. In the context of a man-machine interface, the scheme could leave the user, in this instance a nurse, in doubt whether the value shown on the thumb-wheels was the value that the computer had read and was using in its calculations.

The use of a display counter is less reliable than thumb-wheels but would engender, in the user, a greater sense of safety since the desired value is only input to the microprocessor when the accumulated value is seen to be the same as the desired value.

In the context of human confidence the use of thumb-wheel switches with some display would seem to be desirable since the nurse can observe any differences between the value set on the thumb-wheel switches and that displayed. However, before taking up this option it is necessary to examine the likelihood of the nurse noting the error. Consideration must also be given to the probability that the microprocessor refreshes the display more than once and that the value displayed remains the one being used in calculations. If the microprocessor creates the display value only at the time of reading the thumb-wheel switches then there is the additional probability of an error due to data corruption of the internal value being used.

The controlling influence on safety is the desire to achieve a particular function with a low risk. In the case of the product the probability of an error when the microprocessor reads the value is influential to the safety of the control function. The use of thumb-wheel switches alone has the lowest probability of error, given that there are fewer components contributing to the probability of

error. Therefore, given the options examined, the use on the product of thumb-wheel switches alone can be considered to be the safer option.

5.3.2 Software Structural Analysis

The software in the product comprised three main modules; Standby, Start-up and Background with a fourth module being concerned with interrupt handling. Shared data was the method used for passing values between modules. Other routines were included in the system as function calls.

The structure of the Assembler code was such that the interrupt handler determined which entry point of the code to use dependent upon the interrupt being raised.

The software catered for two alarm conditions; 'System Alarm' and 'Functional Alarm'. The mechanisms concerned with ensuring safety were incorporated into the control modules such that the failure of the control module could result in the failure of the system to maintain safety.

Beizer, [1], p.237, recounts that a contributory factor to the disaster at Three Mile Island was the belief of the Operators that an actuator's real position could not be at variance with its position reported by the computer system.

In the case of the product's functional alarm, the alarm condition required the software to command the microprocessor hardware to release a mechanical device to rapidly close the flow. It is not reasonable to assume that once the release command, which is a safety procedure, has been sent to the output port that the action will be effected as requested, since there is a probability that the output port will not function as commanded. A checking mechanism could be implemented in such cases to determine that the action has actually been effected. If the required action has not

been effected then some alternative procedure should be adopted, such as sounding an additional alarm buzzer. When assessing the safety of the product it was found that no checking mechanism existed to ensure that this, or any other, safety actions had been effected once initiated. In the product assessed, a release command could be issued and the software would assume that the mechanism had been released even though, in fact, it had not. If the reason for the release of the mechanism was to prevent an unsafe condition from existing then, without some form of mechanism to check that the action had effected, the unsafe condition would persist.

A similar situation was found in the case of an alarm condition requiring movement from a pinch-wheel via a stepping motor; once the command had been issued by the software to the output port requesting the movement to the 'home' position there was no checking mechanism within the software to ensure that the motor was actually in motion, in the direction requested. Also when the motor was being used as part of the control system there was three independent routines called to effect movement of the motor, rather than just one. If just one motor control module had been used then the possibility of 'deadlock' would have been greatly reduced.

From the analysis undertaken it was apparent that features such as the motor control and release mechanism, which are essentially safety modules, were incorporated into seperate modules. As discussed in Chapter 3, the system is a more safe construction when the software is seperated into safety modules and control modules. A recommendation to the company, therefore, could be that both the alarm modules should be extended to incorporate checks that the requested actions had been effected, rather than allowing the software to make an assumption on the operation.

5.3.3 Risk Analysis of the Product

The application of Software Fault Tree Analysis and Software Event Tree Analysis was discussed in Chapter 3 in the context of high-level languages. However, the product assessed had been written in an Assembler language so it was helpful to this research to code the software into a Pascal-like language from the Assembler before applying Fault Tree Analysis and Event Tree Analysis.

A complete SFTA/SETA of the system would undoubtedly reveal the product's identity and as the analysis was only possible provided confidentiality was maintained only a small part, the Functional Alarm Module FALARM, will be illustrated.

5.3.3.1 Software Fault Tree Analysis

The assembler code for the Functional Alarm Module (FALARM) was:

```
FALARM:   DI                    ;disable_maskable_interrupts
          POP   IY              ;save address of calling_module in IY
          LD    SP,RAMTOP+1     ;reinit. stack_pointer
          LD    A,11110100B     ;reset RT_latch
                                ;release pull_in_solenoid
                                ;visual_run    OFF
          OUT   (PORTC),A       ;audible_alarm ON
                                ;visual_alarm  ON
                                ;visual_stdbye OFF
                                ;wdog          OFF
FALRLP:   CALL  STOPRD          ;stop/reset pressed ?
          JP    Z,STDBYE        ;IF_YES go to standbye_mode
                                ;ELSE flash visual_alarm and
                                ;pulse audible_alarm
          LD    A,11100100B     ;audible_alarm OFF
                                ;visual_alarm  ON
                                ;visual_stdbye OFF
          OUT   (PORTC),A       ;wdog          OFF
          CALL  DELAY3
          CALL  STOPRD          ;stop/reset pressed ?
          JP    Z,STDBYE        ;IF_YES go to standbye
                                ;ELSE
          LD    A,11110000B     ;audible_alarm ON
                                ;visual_alarm  OFF
                                ;visual_stdbye OFF
          OUT   (PORTC),A       ;wdog          OFF
          CALL  DELAY3
          JR    FALRLP          ;loop till stop/reset pressed
          CALL  SALARM          ;s/w trap
```

which was recoded into a Pascal-like language for analysis:

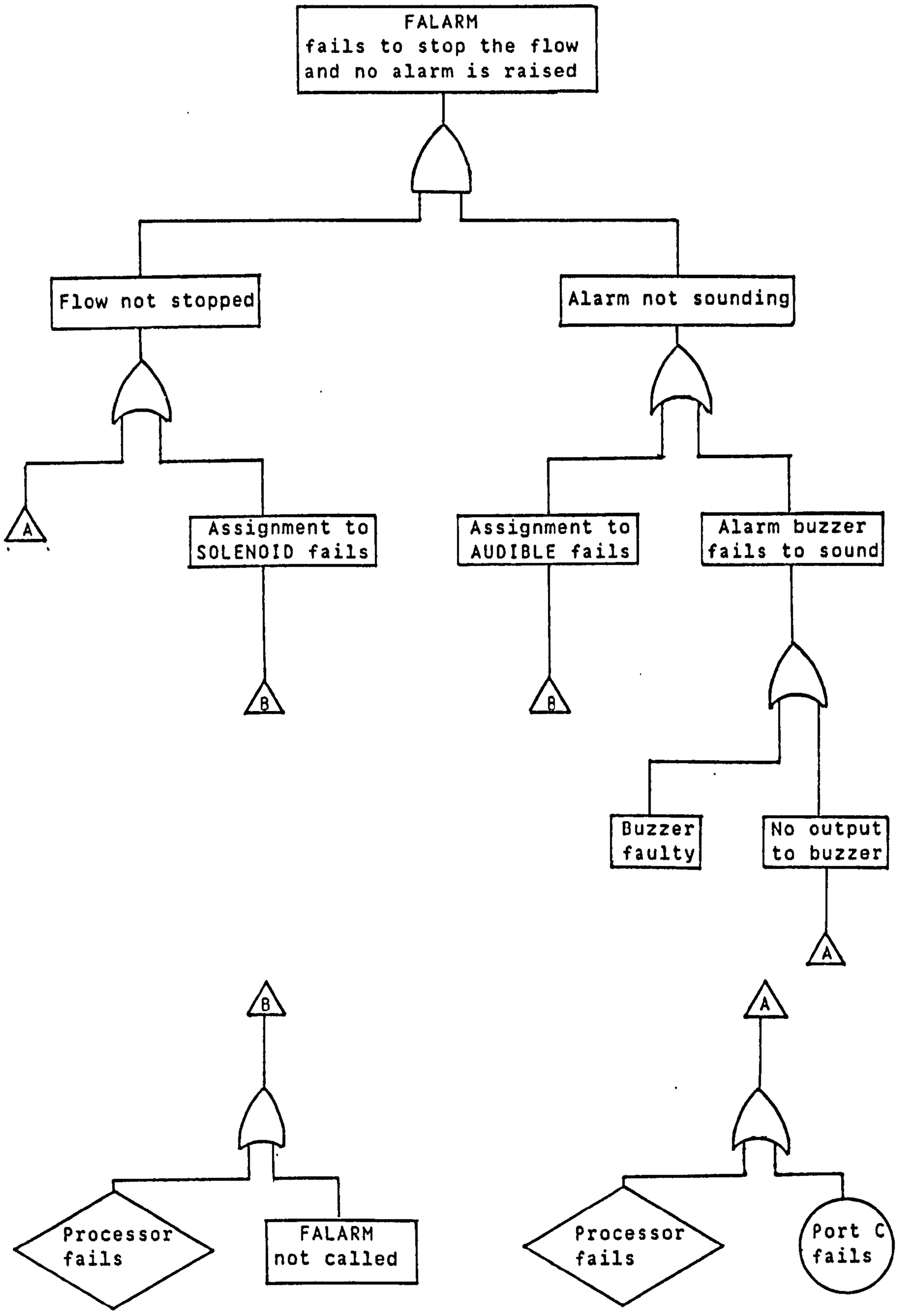
```
solenoid := FALSE;
audible  := TRUE;
visual   := TRUE;
WHILE NOT stopped DO
      BEGIN
         audible := FALSE;
         visual  := TRUE;
         delay3;
         IF NOT stopped THEN
            BEGIN
               audible := TRUE;
               visual  := FALSE;
               delay3;
            END;
      END;
   standby;
```

where STANDBY, STOPPED and DELAY3 are predefined functions.

When the module was analysed for the failure "failure to stop the flow" the SFTA diagram, summarised in Figure 5.3.3.1.1, was developed.

From the analysis it was observed that a failure of Port C, which controls both the alarms and the solenoid release mechanism, would cause the flow to continue uninterrupted without an alarm condition being signalled to the nursing staff. Using the knowledge of the probability of a failure of Port C, recommendations could be made to the company to put the solenoid release mechanism onto a different output port from the one which contained the alarms and also to include a check that the solenoid had actually been released when requested.

Figure 5.3.3.1.1 SFTA of the Product's Functional Alarm



Following the recommendations, the Functional Alarm Module could be altered to the following program:

```
solenoid := FALSE;
delay3;
IF NOT (solenoid) AND NOT (solenoid_released)
   THEN klaxon;
audible  := TRUE;
visual   := TRUE;
WHILE NOT stopped DO
      BEGIN
         audible := FALSE;
         visual  := TRUE;
         delay3;
         IF NOT stopped THEN
            BEGIN
               audible := TRUE;
               visual  := FALSE;
               delay3;
            END;
      END;
   standby;
```

where KLAXON is a function call to initiate some additional alarm action.

5.3.3.2 Software Event Tree Analysis

The Software Event Tree Analysis, SETA, of the whole of the software was also done as a Pascal-like version of the Assembler code and produced an event tree which took many sheets of paper. The event tree did, however, demonstrate that SETA could be used.

As an illustration of how SETA was applied to the whole of the software, the FALARM Module used for SFTA will be used with statement numbers added;

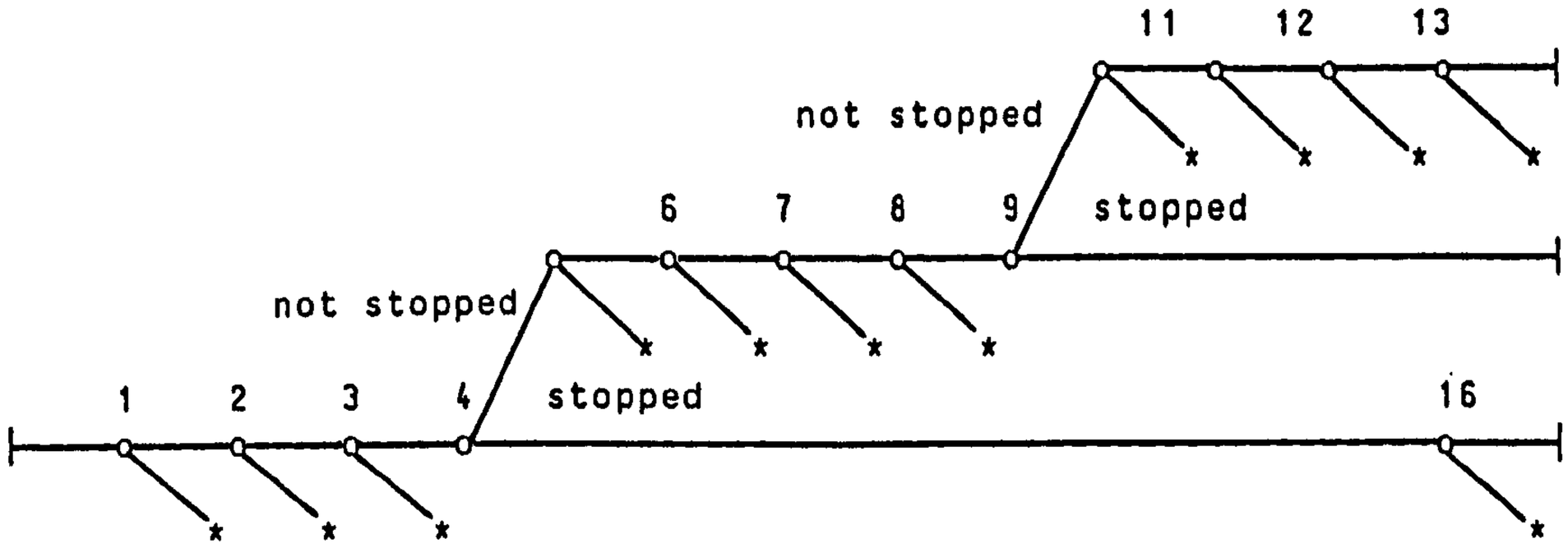
```
 1 solenoid := FALSE;
 2 audible  := TRUE;
 3 visual   := TRUE;
 4 WHILE NOT stopped DO
 5       BEGIN
 6          audible := FALSE;
 7          visual  := TRUE;
 8          delay3;
 9          IF NOT stopped THEN
10             BEGIN
11                audible := TRUE;
12                visual  := FALSE;
13                delay3;
14             END;
15       END;
16 standby;
```

In SETA the emphasis is on the control flow and the BEGIN and END statements form bounds to statement blocks and only indirectly

control the flow, so they are not considered.

The SETA for the module FALARM is;

Figure 5.3.3.2.1 SETA of the Product's Functional Alarm



To the SETA would normally be applied the probabilities of success, which at present have not been determined. If the probabilities were attached then the failure of the solenoid and Port C would be taken into account in the probability calculations associated with the procedure SOLENOID. In which case, the failure branch from node 1 would be correspondingly large and so be noticeable as a cause for further examination and concern.

It was apparent during the assessment that some research needs to be undertaken to determine typical failure probabilities for programming statements as these were difficult to obtain for the current analysis.

5.3.4 Application of Metrics

The manual application of the metrics developed in Chapter 4 was found to be time consuming. It would have been more efficient if some software tools had existed so that the actual Assembler code could have been mounted onto a computer for automatic analysis. The development of software tools to analyse the assembler code would reduce the time needed to conduct an assessment and would also provide more scope for analysis. Using the few tools that had been

developed as part of the research and, in the case of Fallibility Index, some manual calculation the following analysis was obtained:

a) Perturbation of Names

Number of Variable Names = 51

Total Number of Variable References = 388

Number of Variables with a Hamming Distance of one = 10

Number of Undetectable Alternates = 298

Number of Undetectable Omissions = 3

Number of Undetectable Insertions = 57

b) Fallibility Index

Number of Alternates, Na = 95616

Number of Omissions, No = 1016

Number of Insertions, Ni = 19676

Number of Poss. Errors, Npe = 2271168

Fallibility Index = 5.12%

c) Plexus

Number of Characters = 17743

Number of control statements = 1819

Plexus = 11024

Decision Content = 106458

Relative Redundancy = 89.65%

Error-Proneness = 10.35%

The interpretation of the analyses supports the hypothesis that any measurement of safety is conjoint and dependent upon empirical observations. To place the analysis in context it can be compared with the values determined for the example program in Chapter 4. Such a comparison shows that the product's Fallibility Index and Error-proneness could be considered as being reasonable given that it has a large Plexus value.

From the analysis of the perturbation of names it can be noted

that variables with a Hamming distance of one, 10 in number, had a high frequency of usage. This high frequency of usage influenced the number of undetectable alternates. If those variables with a Hamming distance of one were changed to have a Hamming distance of two or more, then for a relatively small change a large response would be achieved in the number of undetectable alternates. The change would also be reflected in a favourable change in the Fallibility Index.

It may be possible to automate the procedures for detecting variables with a Hamming distance of one prior to compilation and to recommend to the prpgrammer the necessary changes. The advantage of such automation would be to reduce the time needed for testing the software and also to reduce the Fallibility Index.

5.3.5 Integrity of the Product

The creation of a unique identity, Integrity Lock, for the product's software was not practicable since it was only possible to have access to the program listings and not to the actual Assembler code. To generate an Integrity Lock manually from the listings was considered to be too time-consuming for the research. The generation of an Integrity Lock is more efficiently achieved by automation using the internal representation of the software as it is this representation which needs to be protected and checked at run-time.

a Safety Audit could be considerably reduced by the removal of the constraints experienced and priority being given to the development of software tools. It was also apparent that access to design documents, test records and design personnel would have made a Software Safety Audit more comprehensive.

In order for the product to be certified as safe the lack of checking mechanisms within the software on initiated actions and the assumption on the correct working of devices would need to be re-examined by the company.

It is not possible to place any importance on the metric values obtained as little is known about the relationship between the values and safety. In order to place the values on a scale of values some research is needed to calibrate values obtained for a number of real systems. However, the form that a calibration exercise would take is difficult to envisage without returning to the use of a subjective judgement on the ordering of the values within the scale of values. This subjectivity endorses the view that safety will remain subjective until some mapping function is found to relate the definition of safety and the set of scales of values. Until such a mapping function is found the use of procedures, like those illustrated, are needed to guide the Auditor towards a judgement on the safety of the software or the system.

The use of non-quantitative procedures to guide judgement based on experience and what is called 'best world practice' are found in other engineering fields, for example shipping insurance. The analysis of the product has shown that such procedures can also be aplied to the analysis of software.

5.4 Discussion

The reason for conducting the Software Safety Audit was to examine the practicability of the procedures and not simply to derive some values for a system. To this end it has been possible to show that the procedures are practicable and incidentally to derive some values from a real system.

The Software Safety Audit was carried out over a period of approximately five weeks which was longer than originally anticipated, mainly because of the manual calculation of the Fallibility Index. The effort required to conduct the Software Safety Audit was influenced by many constraints, principally, unfamiliarity with the programming language, lack of knowledge of the system being assessed, the lack of appropriate software tools and very limited access to the design personnel. The most significant of these was the lack of familiarity with the language and this needed to be addressed with care.

The effort expended on the assessment could be reduced by the development of software tools and with more experience in the application of the procedures. The problem of familiarity with the programming language used in the system being assessed could be minimised if the software toolset included tools for analysing a language syntax and which could be generated relatively easily for uncommon languages. Such a tool could be created with the aid of tools such as YACC, which is part of the UNIX Operating System.

It was evident during the Software Safety Audit that whilst it is possible to apply SETA to application software there was a need for further research into SETA before it could be considered to be as useful as SFTA.

The method of conducting a Software Safety Audit worked well given the constraints mentioned above. The time taken to carry out

5.5 References

1. Beizer,B., Software Testing Techniques, Van Nostrand Reinhold, 1983

2. H.M.S.O., The Health and Safety at Work etc. Act 1974, Her Majesty's Stationery Office, London

CHAPTER 6

Conclusions

The control of industrial processes by computer has new risks associated with it. One of the new risks is the incorporation of software into the control system of a controller.

It has been shown in this thesis that measurements of software reliability and measurements of software safety do not have the same goals. Software reliability and software safety have been shown to be related subjects and that software safety is a seperate and distinct subject.

The terms hazard, risk, danger and safe have been defined in terms of industrial control and a relationship between these terms has been postulated. Though an attempt has been made to discuss the philosophy of safety it is evident that there is considerable scope for further work. A formal definition of software safety has been proposed and the terms 'safe' and 'unsafe' have been shown to be subjective judgements.

It is the hypothesis of this thesis that software influences the safe operation of industrial-based controllers incorporating software and that the risk can be assessed and quantified.

The subjectivity of safety has been examined and it is suggested that an assessment of safety is a conjoint measurement.

An examination of the factors affecting the software development process and the metrics available for measuring the influence of these factors has shown that there are many influences affecting software but that there are few metrics available.

The research for this thesis found that the software incorporated into industrial-based controllers has an influence on the safety of the control system and that there are many aspects to assessing the safety of software used for industrial control. It

has been shown that some of these aspects can be quantified but there is no evidence that metrics have yet been proved to measure the safety of software used for industrial control.

The research for this thesis applied analytical methods, taken from other engineering disciplines, to assess the risk of using an item of software. It was found that the structure of the software can be examined using Software Fault Tree Analysis and Software Event Tree Analysis but that further research is needed into Software Event Tree Analysis before its usefulness can be exploited.

The research also examined the use of State Transition Diagrams as a method of determining erroneous states and found that the number of such states can become unmanageable when all the possible failure conditions are considered, even for relatively simple control systems.

Three categories of danger have been proposed for the ocassions when software is used in industrial-based control systems; minor, Major and Serious. The structuring of the software for safety has also been examined and a suggestion has been made on the use of Safety Modules and a mechanism called an Integrity Lock.

The research for this thesis has shown that software errors can be introduced at each stage of the development of the software and two methods of measuring the possibility for error have been proposed; Plexus and Fallibility Index. Further rigorous development of the Plexus metric and the Fallibility Index is required before the meaning of the measurements is known.

It was found from an experiment that variables declared with seven characters had a significantly better probability of correct interpretation than for variables declared with more than or less than seven characters.

The combination of the risk analysis, structural analysis and

software analysis into a set of assessment procedures has been called a Software Safety Audit. A Software Safety Audit was done on a commercial product and it was found that further development of the assessment method is needed.

'Loss Containment' is a term often used in the Process Industries to describe procedures for containing the consequences of the loss of safe-working within acceptable criteria. The loss containment of software requires requires a judgement to be made on the course of action to take when the software or the system becomes 'unsafe'. Such a judgement will need to consider both the economic and the social consequences of the action, the practicability of such action and the time needed for the action to achieve a state which is considered to be 'more safe'.

In some systems it may be possible to determine the possible unsafe states and make a prior judgement on the appropriate action to take for each unsafe state.

In some systems it will not be possible to determine the possible unsafe states as the number may be unmanageable. Similarly, it may not be possible to make a judgement on the appropriate actions to take as these may be too numerous, may be subject to a large number of variables or may be indeterminate. In such systems some method is required which will allow a judgement to be made based on the current safety practice, the current unsafe state, available states, time available to respond to the current unsafe state and possibly many other variables. The development of what are called Intelligent Knowledge-Based Systems may be applicable and research could be conducted into the use of these systems as monitors of safety-related systems.

The research for this thesis has made a start on the subject of assessing and quantifying the safety of software used for industrial

control. The research has identified subject areas which, with more research, could produce methods and metrics to quantify the safety of software.

APPENDIX 1

Database Program

The following dBase II program was used in the experiment reported in Chapter 4.3.

```
SET TALK OFF
SET FORMAT TO SCREEN
USE B:PSYCHO.DBF
ERASE
DO WHILE T
?
?
?
?
?
?
?' ARE YOU PREPARED TO HELP A RESEARCH STUDENT '
?' WITH A SIMPLE EXPERIMENT?'
?
?
?' If so, then press any key. '
?
?
WAIT TO ACTION
ERASE
DO WHILE T
?
?
?' Thank you for agreeing to assist in this simple experiment to'
?'try and determine the level of difficulty people experience in the'
?'use of mnemonics in the place of lines of text.'
?
?' You are asked to suggest a suitable mnemonic for each line of'
?'text presented to you. As an example you might suggest that a '
?'suitable mnemonic for the text LENGTH OF STRING IN METRES could be'
?'METRELENGTHS. Please limit your suggested mnemonic to no more '
?'than 20 characters. You will be asked to suggest eight such '
?'mnemonics.'
?
?' After which the screen will clear and you will be asked to '
?'restate the mnemonics from memory. Finally, you will be asked to'
?'input your understanding of the mnemonics.'
?
?' Terminate each input with a RETURN before starting the next'
?'question.'
?
?' When you are happy that you understand what is to happen, '
?'signal your readiness by pressing any key'
?
WAIT TO ACTION
ERASE
?
ACCEPT "What is your SURNAME? " to d:name
ACCEPT " and your INITIALS " to d:inits
ACCEPT " State which OU Summer School " to d:venue
ACCEPT " Please enter todays date " to d:date
?
```

```
?
?
?'    Please suggest a mnemonic for the following:'
?
ACCEPT "TIME OF DAY " to m:1
ACCEPT "VALVE 8 POSITION " to m:2
ACCEPT "AMBIENT TEMPERATURE" to m:3
ACCEPT "LIQUID FLOW IN LITRES/MINUTE" to m:4
ACCEPT "MOTOR SPEED IN R.P.M" to m:5
ACCEPT "WEIGHT OF PRODUCT IN TONNES" to m:6
ACCEPT "DISTANCE FROM THE VALVE CONTROLLER TO THE VALVE IN METRES" to
                                                                    m:7
ACCEPT "PERCENTAGE OF SULPHUR DIOXIDE IN THE ATMOSPHERE" to m:8
ERASE
?
?
?
?
?
?
?
?'    Can you now re-enter the mnemonic you suggested for'
?
ACCEPT "TIME OF DAY" to t:1
ACCEPT "LIQUID FLOW IN LITRES/MINUTE" to t:4
ACCEPT "WEIGHT OF PRODUCT IN TONNES" to t:6
ACCEPT "VALVE 8 POSITION" to t:2
ACCEPT "PERCENTAGE OF SULPHUR DIOXIDE IN THE ATMOSPHERE" to t:8
ACCEPT "AMBIENT TEMPERATURE" to t:3
ACCEPT "DISTANCE FROM THE VALVE CONTROLLER TO THE VALVE IN METRES" to
                                                                    t:7
ACCEPT "MOTOR SPEED IN R.P.M." to t:5
ERASE
APPEND BLANK
REPLACE SURNAME with d:name, INITIALS with d:inits
REPLACE DATE with d:date, VENUE with d:venue
REPLACE MNEMONIC1 with m:1, MNEMONIC2 with m:2
REPLACE MNEMONIC3 with m:3, MNEMONIC4 with m:4
REPLACE MNEMONIC5 with m:5, MNEMONIC4 with m:4
REPLACE MNEMONIC5 with m:5, MNEMONIC6 with m:6
REPLACE MNEMONIC7 with m:7, MNEMONIC8 with m:8
REPLACE TEXT1 with t:1, TEXT2 with t:2, TEXT3 with t:3
REPLACE TEXT4 with t:4, TEXT5 with t:5, TEXT6 with t:6
REPLACE TEXT7 with t:7, TEXT8 with t:8
?
?
?
?'     Can you now try and give a short description for the following'
?'mnemonic you suggested:'
?
@ 12,5 SAY MNEMONIC1 GET ANSWER1
@ 13,5 SAY MNEMONIC2 GET ANSWER2
@ 14,5 SAY MNEMONIC3 GET ANSWER3
@ 15,5 SAY MNEMONIC4 GET ANSWER4
@ 16,5 SAY MNEMONIC5 GET ANSWER5
@ 17,5 SAY MNEMONIC6 GET ANSWER6
@ 18,5 SAY MNEMONIC7 GET ANSWER7
@ 19,5 SAY MNEMONIC8 GET ANSWER8
READ
USE
DO A:PSYCHO
```

APPENDIX 2

Program Analyses

The programs analysed below are linear expansions of those referred to in Chapter 4.4. A comparison of the measures is plotted in Graph 3 to Graph 5. The values were calculated using the program given in Appendix 3.

Program 1

```
PROGRAM                          Halsteads Length, N^ = 12.75
VAR:x;                            Halsteads Volume, V = 36.49
VAR:y;
BEGIN                             Decision Content, H = 282.1
  x:=2;                                     Plexus, P =  29.8
  y:=95;                           Error Proneness, E =  10.6%
  x:=x+y;                    Number Characters, Nc =  53
  PRINT:x;
END
```

Program 2

```
PROGRAM                          Halsteads Length, N^ = 12.75
VAR:x;                            Halsteads Volume, V = 50.53
VAR:y;
BEGIN                             Decision Content, H = 319.3
  x:=2;                                     Plexus, P =  34.6
  y:=95;                           Error Proneness, E =  10.8%
  x:=x+y;                    Number Characters, Nc =  60
  x:=x+y;
  PRINT:x;
END
```

Program 3

```
PROGRAM                          Halsteads Length, N^ = 12.75
VAR:x;                            Halsteads Volume, V = 64.57
VAR:y;
BEGIN                             Decision Content, H = 356.57
  x:=2;                                     Plexus, P =  39.4
  y:=95;                           Error Proneness, E =  11.0%
  x:=x+y;                    Number Characters, Nc =  67
  x:=x+y;
  x:=x+y;
  PRINT:x;
END
```

Program 4

```
PROGRAM
VAR:x;
VAR:y;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  x:=x+y;
  x:=x+y;
  x:=x+y;
  PRINT:x;
END
```

```
Halsteads Length, N^ = 12.75
 Halsteads Volume, V = 78.61
 Decision Content, H = 393.8
           Plexus, P =  44.1
  Error Proneness, E =  11.2%
Number Characters, Nc =  74
```

Program 5

```
PROGRAM
VAR:x;
VAR:y;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  x:=x+y;
  x:=x+y;
  x:=x+y;
  x:=x+y;
  PRINT:x;
END
```

```
Halsteads Length, N^ = 12.75
 Halsteads Volume, V = 92.64
 Decision Content, H = 431.1
           Plexus, P =  48.9
  Error Proneness, E =  11.3%
Number Characters, Nc =  81
```

Program a

```
PROGRAM
VAR:x;
VAR:y;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  y:=x+y;
  PRINT:y;
END
```

```
Halsteads Length, N^ =  12.75
 Halsteads Volume, V =  50.5
 Decision Content, H =  319.3
           Plexus, P =   34.6
  Error Proneness, E =   10.8%
Number Characters, Nc =   60
```

Program b

```
PROGRAM
VAR:x;
VAR:y;
VAR:p;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  p:=x+y;
  PRINT:p;
END
```

```
Halsteads Length, N^ = 16.36
 Halsteads Volume, V = 54.0
 Decision Content, H = 351.25
           Plexus, P =  43.2
  Error Proneness, E =  12.3%
Number Characters, Nc =  66
```

Program c

```
PROGRAM
VAR:a;
VAR:x;
VAR:y;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  a:=x+y;
  PRINT:a;
  a:=x+a;
  x:=1025;
  x:=y+x;
  y:=a+x;
  PRINT:y;
END
```

Halsteads Length, N^ = 20.3
Halsteads Volume, V = 120.5

Decision Content, H = 548.2
Plexus, P = 70.9
Error Proneness, E = 12.9%
Number Characters, Nc = 103

Program d

```
PROGRAM
VAR:a;
VAR:b;
VAR:x;
VAR:y;
VAR:p;
BEGIN
  x:=2;
  y:=95;
  x:=x+y;
  p:=x+y;
  PRINT:p;
  a:=x+p;
  b:=1025;
  x:=y+b;
  y:=a+x;
  PRINT:y;
END
```

Halsteads Length, N^ = 28.75
Halsteads Volume, V = 131.46

Decision Content, H = 612.0
Plexus, P = 95.7
Error Proneness, E = 15.6%
Number Characters, Nc = 115

APPENDIX 3

BASIC Program

The BASIC Program that follows was written to enable the analysis of the example programs given in Chapter 4.4 and also to provide the data included in Appendix 2 and Table 3.

```
10 DIM VARFREQ(25)
20 DIM DIGFREQ(25)
30 DIM PROCFREQ(25)
40 REM
50 REM -------------------- Inputs for Halstead ---------
60 REM ---------- Variables are: N1, N2, NN1, NN2 --------
70 REM
80 INPUT "Halsteads Number of Unique Operators, n1 = ";N1
90 INPUT "Halsteads Number of Unique Operands, n2 = ";N2
100 INPUT "Halsteads Number of Operators, N1 = ";NN1
110 INPUT "Halsteads Number of Operands, N2 = ";NN2
120 REM
130 REM ------- Inputs for Decision Content & Complexity ----
140 REM ----- are: LOD, LOP, NC, NV, NP, ND, NSC, NSD, VS ---
150 REM
160 INPUT "Number of Characters in Character Set, Cs = ";CS
170 INPUT "Number of Declarations, D = ";LOD
180 INPUT "Number of Control Statements, Sc = ";LOP
190 INPUT "Number of Characters in the Program, Nc = ";NC
200 INPUT "Tot. No. of Unique Variables, Nv = ";NV
210 INPUT "Tot. No. of Digit Values, Nd = ";ND
220 INPUT "Tot. No. of Relational Ops + Procedures, Np = ";NP
230 INPUT "No. of Available Control Statements, Nsc = ";NSC
240 INPUT "No. of Avail. Declarative Statements, Nsd = ";NSD
250 INPUT "No. of Possibilities for Variables, Vs = ";VS
260 REM
270 REM -------- Calculate Halsteads Length & Volume --------
280 REM -- Variables are: LENGTH, VOLUME, N1, N2, NN1, NN2 --
290 REM
300 LENGTH =(N1 * (LOG(N1)/LOG(2))) + (N2 * (LOG(N2)/LOG(2)))
310 VOLUME = (NN1 + NN2) * (LOG (N1 + N2)/LOG (2))
320 REM
330 REM ---------- Input Frequency of Variables ------------
340 REM ----- Variables used are: NOVAR, NV, VARFREQ(I) -----
350 REM
360 NOVAR = 0
370 FOR I = 1 TO NV
380 INPUT "Input Frequency of Variable ";VARFREQ(I)
390 NOVAR = NOVAR + VARFREQ(I)
400 NEXT I
410 REM
420 REM ------ Input Frequency of Digital Values ------------
430 REM ----- Variables used are: NODIGS, ND,DIGFREQ(I) -----
440 REM
450 NODIGS = 0
460 FOR I = 1 TO ND
470 INPUT "Input Frequency of Digit ";DIGFREQ(I)
480 NODIGS = NODIGS + DIGFREQ(I)
490 NEXT I
500 REM
```

```
510 REM - Input Frequency of Relational Ops and Procedures --
520 REM --- Variables used are: NOPROCS, NP, PROCFREQ(I) ----
530 REM
540 NOPROCS = 0
550 FOR I = 1 TO NP
560 INPUT "Input Frequency of Operator/Procedure";PROCFREQ(I)
570 NOPROCS = NOPROCS + PROCFREQ(I)
580 NEXT I
590 REM ------------------ Clear Screen ---------------------
600 PRINT CHR$(12)
610 REM
620 REM --------- Calculate for the Declaration Part --------
630 REM -- Var's used are: TEMP, LOD, NSD, NV, VS, I, IDEC --
640 REM
650 TEMP = 0
660 PRINT "Plexus, P =";
670 IDEC = 0
680 IF LOD = 0 OR NSD = 0 THEN GOTO 830
690 PRINT LOD;"Log";NSD;"+";
700 FOR I = 1 TO NV
710 TEMP = TEMP + (LOG (VS - (I - 1))/LOG (2))
720 PRINT "Log";VS-(I-1);
730 IF I < NV THEN PRINT "+";ELSE PRINT
740 NEXT I
750 IDEC = LOD * (LOG (NSD)/LOG (2)) + TEMP
760 REM
770 REM ----------- Calculate for the Procedure Part --------
780 REM -- Var's are: TEMP1, NV, VARFREQ(I), NOVAR, TEMP2 ---
790 REM -- TEMP3, NP, PROCFREQ(I), NOPROCS, ND, L, NSC, LOP -
800 REM ------- and DIGFREQ(I), NODIGS ----------------------
810 REM
820 REM
830 TEMP1 = 0
840 PRINT "                          +";LOP;"Log";NSC;"+";
850 FOR I = 1 TO NV
860 TEMP1 = TEMP1 + VARFREQ(I)*(LOG(NOVAR/VARFREQ(I))/LOG(2))
870 PRINT VARFREQ(I);"Log";NOVAR;"/";VARFREQ(I);
880 IF I < NV THEN PRINT "+";ELSE PRINT
890 NEXT I
900 TEMP2 = 0
910 PRINT "                          +";
920 FOR I = 1 TO ND
930 TEMP2 = TEMP2+DIGFREQ(I)*(LOG(NODIGS/DIGFREQ(I))/LOG(2))
940 PRINT DIGFREQ(I);"Log";NODIGS;"/";DIGFREQ(I);
950 IF I < ND THEN PRINT "+";ELSE PRINT
960 NEXT I
970 TEMP3 = 0
980 PRINT "                          +";
990 FOR I = 1 TO NP
1000 TEMP3=TEMP3+PROCFREQ(I)*(LOG(NOPROCS/PROCFREQ(I))/LOG(2))
1010 PRINT PROCFREQ(I);"Log";NOPROCS;"/";PROCFREQ(I);
1020 IF I < NP THEN PRINT "+";ELSE PRINT
1030 NEXT I
1040 REM
1050 REM ------- Calculate the Plexus as a Sum ----------
1060 REM -- Var's used: INFO, LOP, NSC, TEMP1, TEMP2, IDEC --
1070 REM
1080 INFO = LOP * (LOG(NSC)/LOG(2))+TEMP1+TEMP2+TEMP3+IDEC
1090 REM
1100 REM -------- Calculates the Decision Content ----------
```

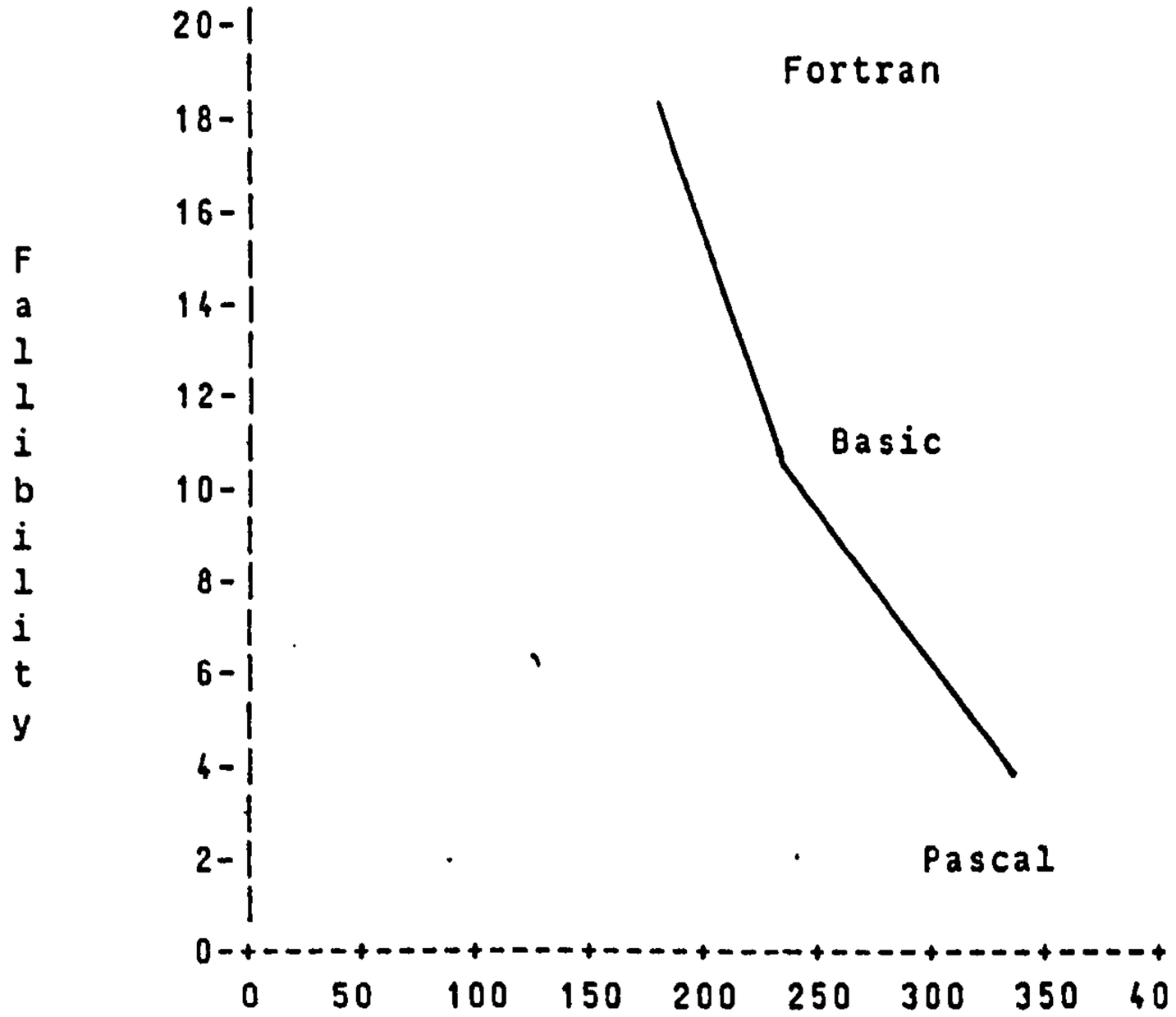
```
1110 REM -------- Variables used are: DECISION, NC ----------
1120 REM
1130 DECISION = NC * (LOG (CS)/LOG (2))
1140 REM ------------ Clear Screen --------------------------
1150 REM PRINT CHR$(12)
1160 REM
1170 REM ---------- Display the Results ---------------------
1180 REM -- Var's used: INFO, DECISION, REDUNDANCY, ERRORS --
1190 REM --------- and N1, N2, NN1, NN2, VOLUME, LENGTH -----
1200 REM
1210 PRINT
1220 PRINT "       Plexus, P =";INFO
1230 PRINT "   Decision Content =";DECISION
1240 REDUNDANCY = (DECISION - INFO) / DECISION * 100
1250 PRINT "Relative Redundancy =";REDUNDANCY;"%"
1260 ERRORS = 100 - REDUNDANCY
1270 PRINT "    Error-Proneness =";ERRORS;"%"
1280 PRINT
1290 PRINT
1300 PRINT "Halsteads Length, N^ =";N1;"Log";N1;" + ";N2;
1310 PRINT "Log";N2;" = ";LENGTH
1320 PRINT "Halsteads Volume,  V =";NN1+NN2;"Log";N1+N2;
1330 PRINT " = ";VOLUME
1340 PRINT
1350 PRINT
1360 PRINT
1370 GOTO 80
```

APPENDIX 4

Graph 1

Fallibility Index against the Number of Characters (Nc) to write a Program

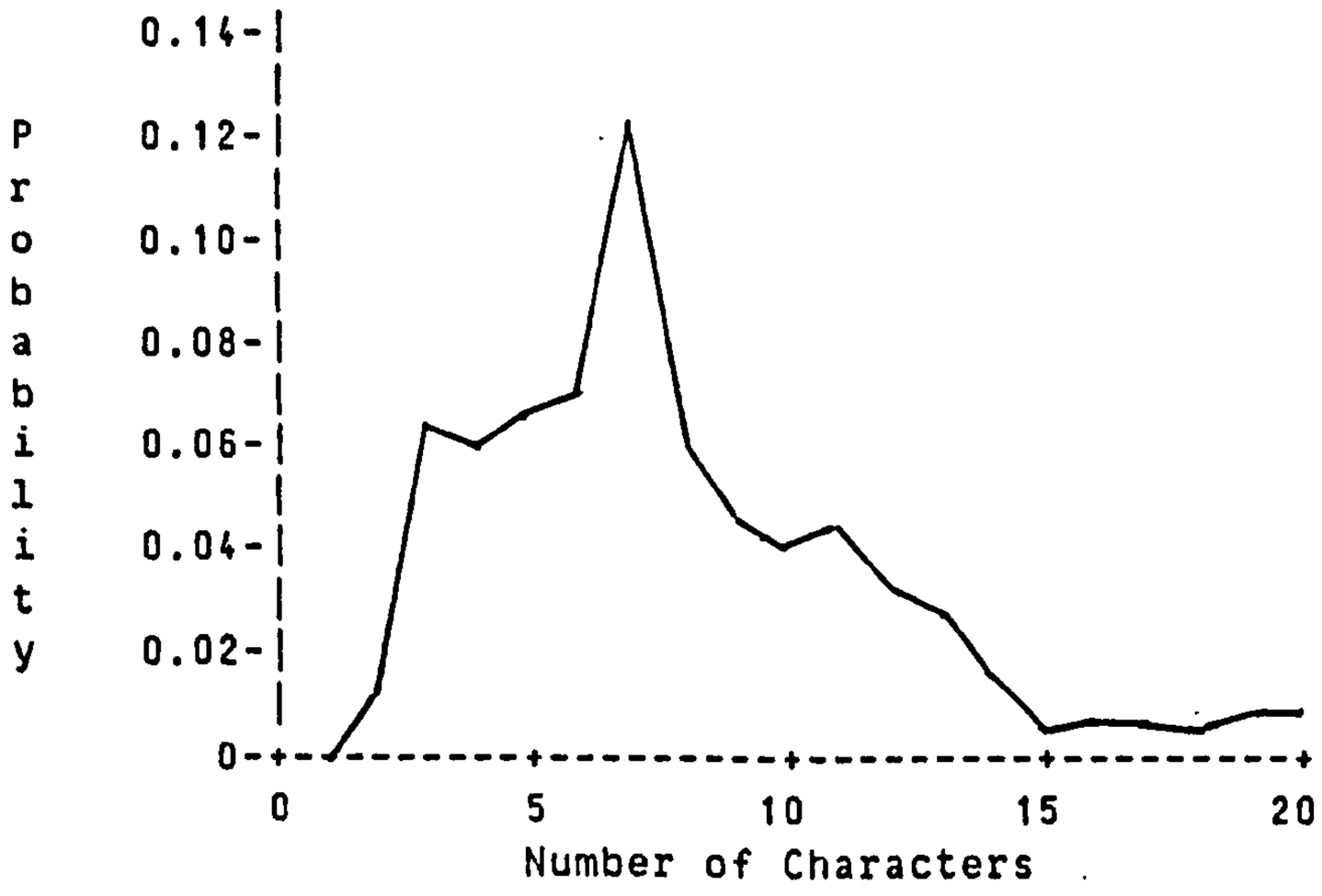
[Source: Chapter 4.2]



Graph 2

The Probability of a Correct Interpretation of a Variable for Varying Numbers of Characters

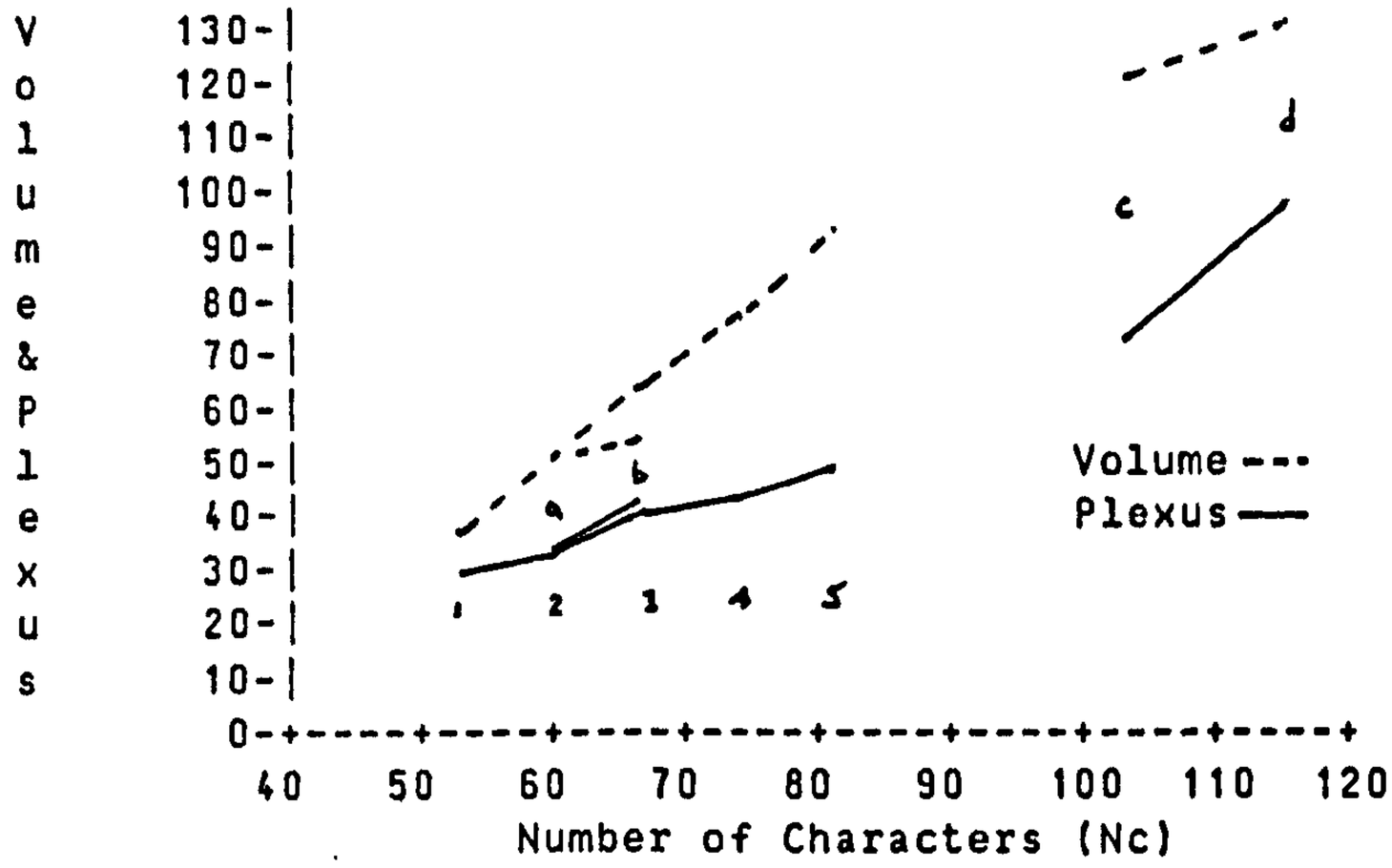
[Source: Chapter 4.3 and Table 2]



Graph 3

The Plexus Metric and Halsteads Volume Metric against the Number of Characters (Nc) to write a Program.

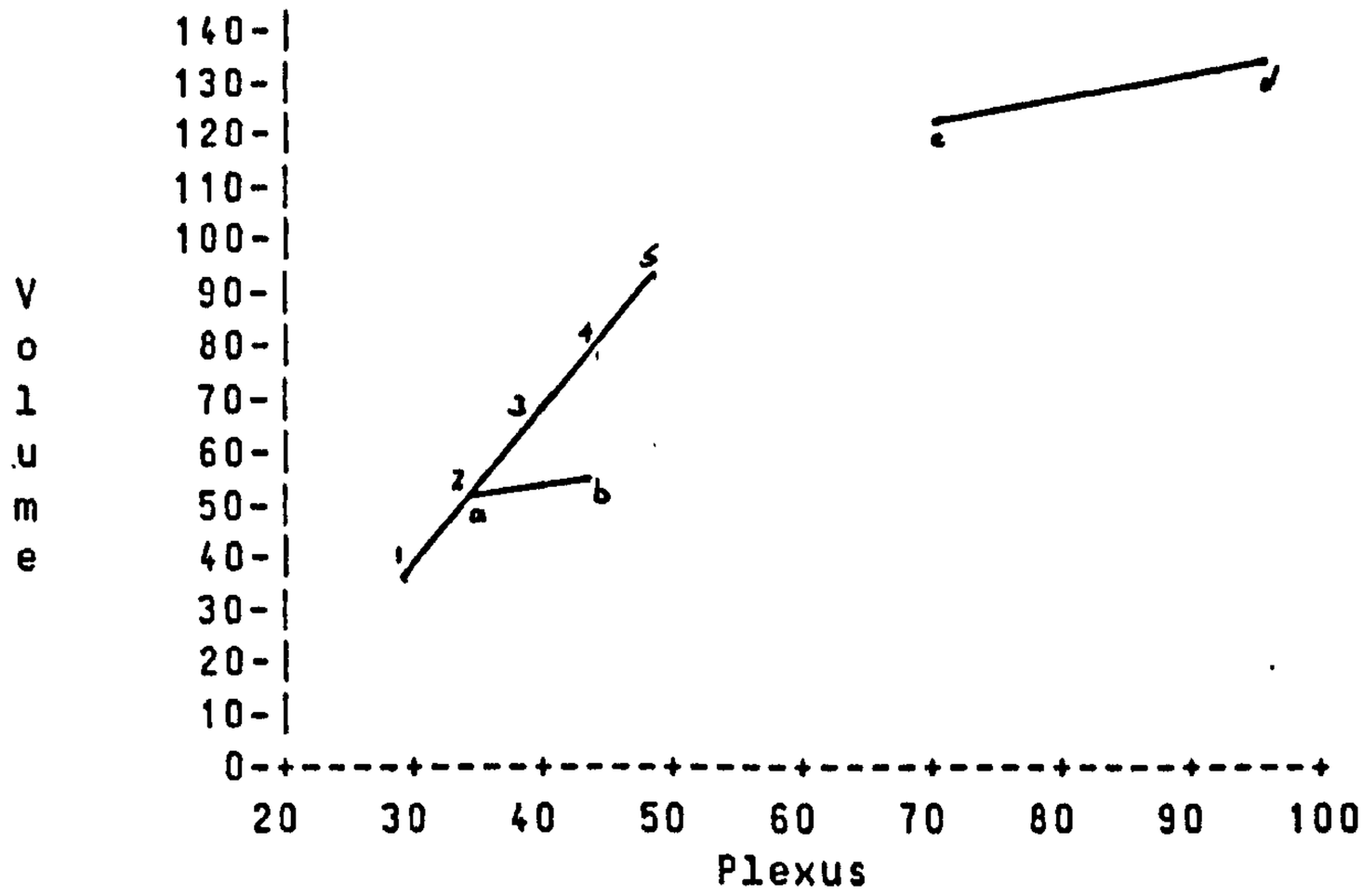
[Source: Chapter 4.4 and Appendix 2]



Graph 4

Comparison of the Plexus Metric and Halsteads Volume Metric.

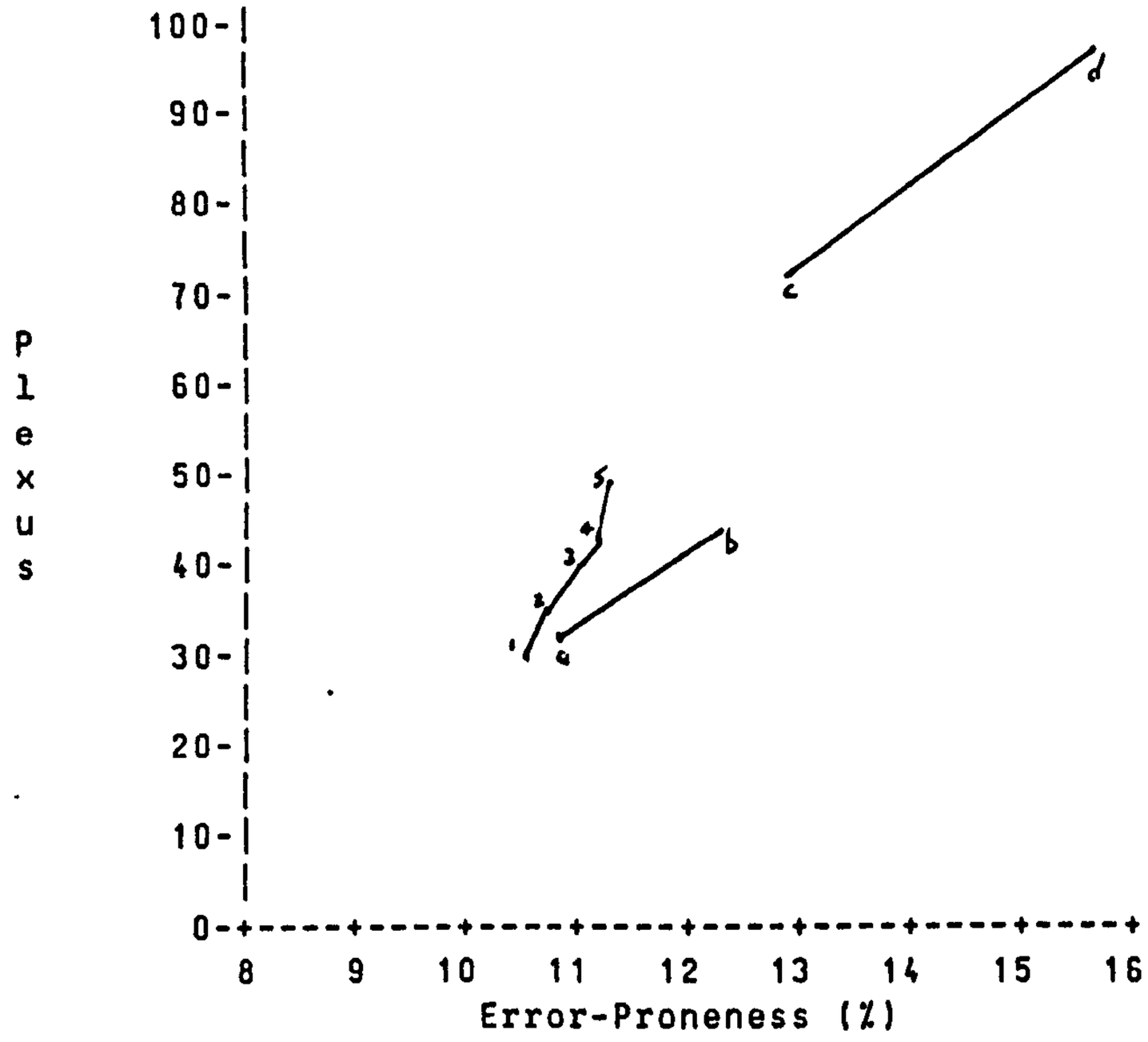
[Source: Chapter 4.4 and Appendix 2]



Graph 5

The Plexus Metric and Error-Proneness

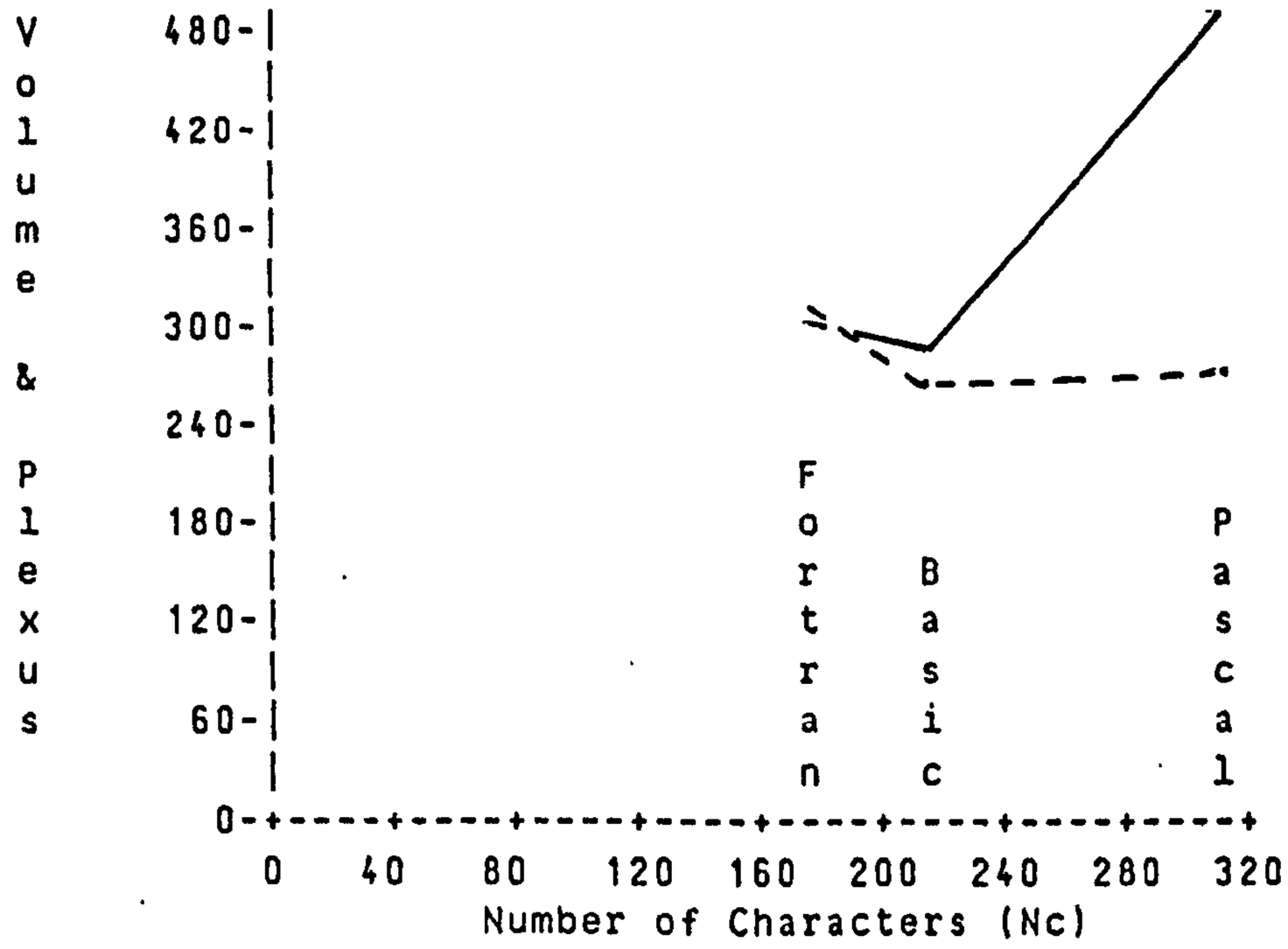
[Source: Chapter 4.4 and Appendix 2]



Graph 6

The Plexus Metric and Halstead's Volume Metric plotted against the Number of Characters to write a Program - Declaration Part Included.

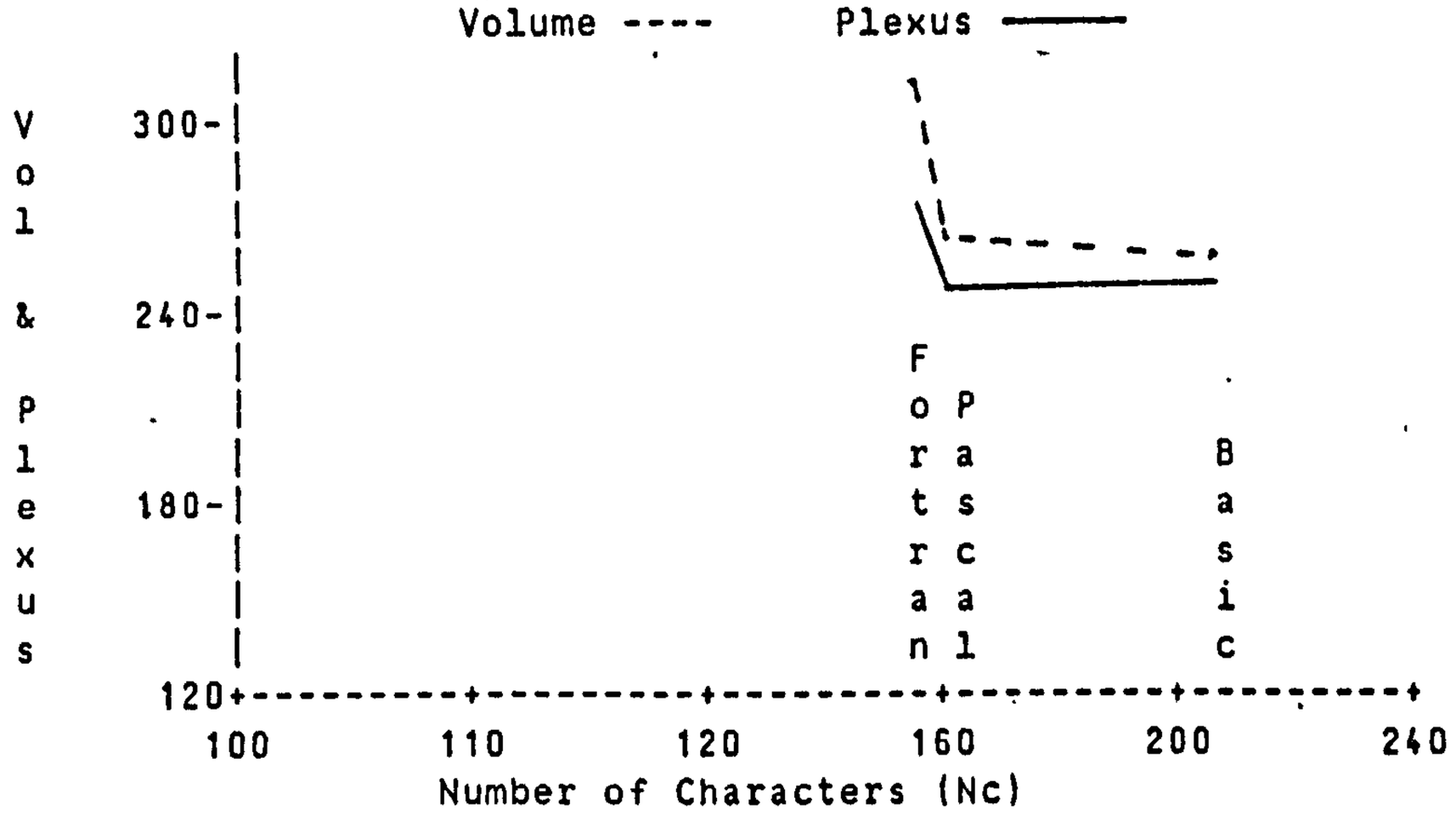
[Source: Chapter 4.4 and Table 3]



Graph 7

Halstead's Volume Metric and the Plexus Metric against the Number of Characters to write a Program - Declaration Part Omitted.

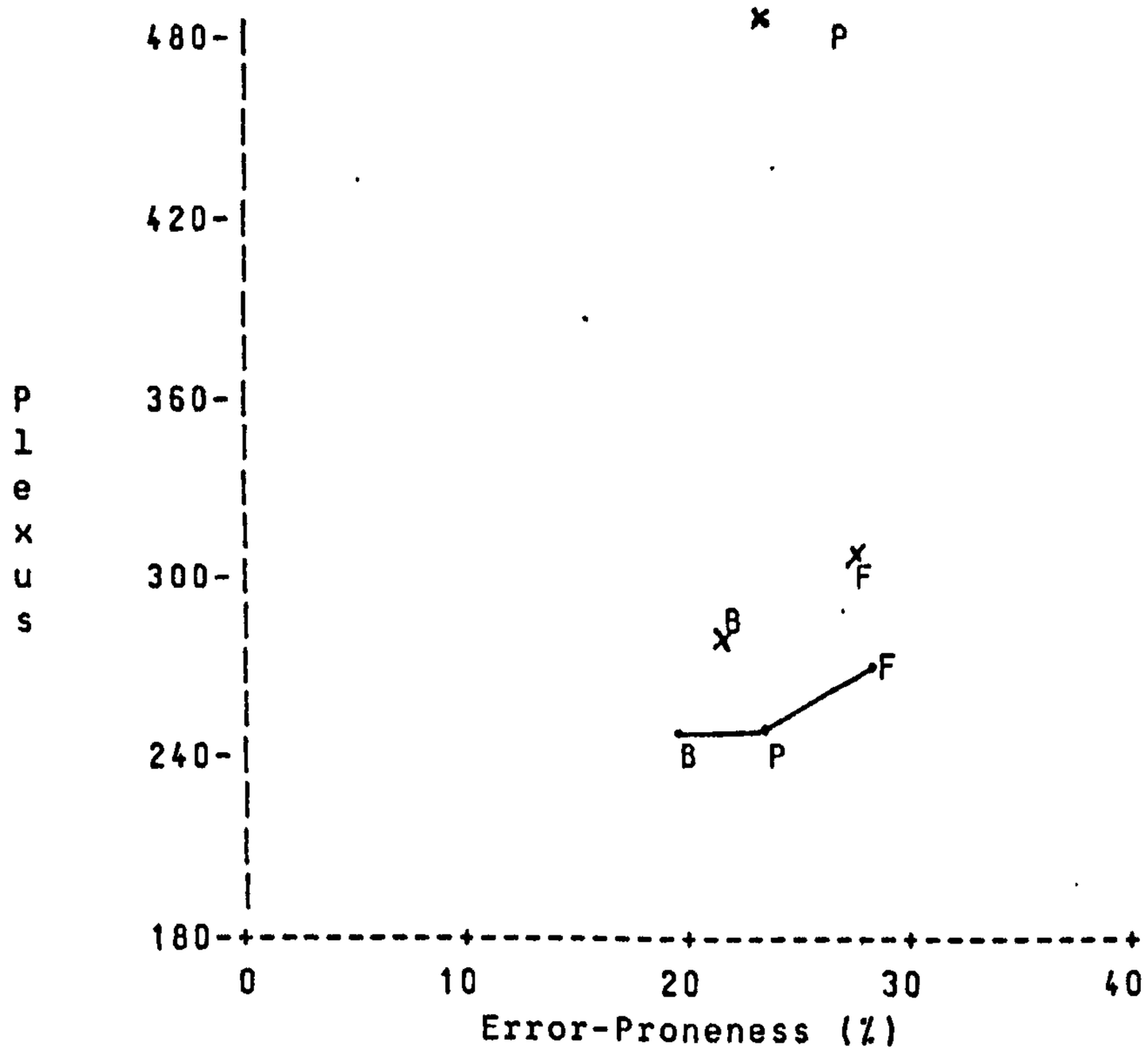
[Source: Chapter 4.4 and Table 3]



Graph 8

Plexus Metric plotted against Error-Proneness where the Declaration Parts are Included and Omitted.

[Source: Chapter 4.4 and Table 3]
with Declarations (x) without Declarations (.)

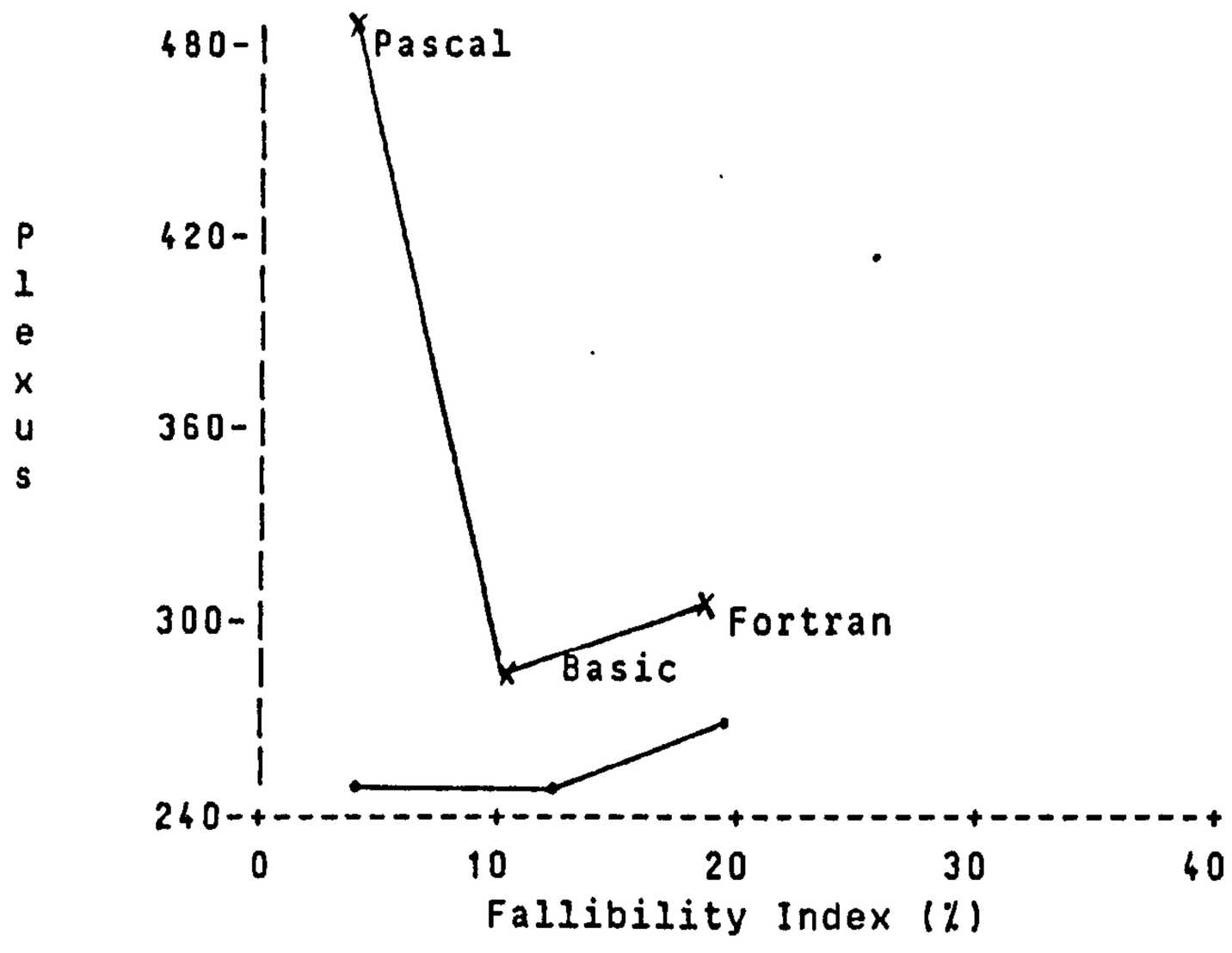


Graph 9

The Plexus Metric plotted against the Fallibility Index

[Source: Chapter 4.4]

with Declarations (x) without Declarations (.)



APPENDIX 5

Table 1. Analysis of the Literature Survey

Application by Risk Level / Factors	Scientific inc. Academic	DP	Infra structure	Indust. Control	High Risk	Sub-Total	Def.	Metric	Total
1. Specification/Design	(2)	(14)	(7)	(6)	(11)	(40)	(11)	(6)	(57)
1.1 Formal	2	8	5	3	7	25	5	2	32
1.2 Functional		2	1	2	2	7	3	2	12
1.3 Language				1	2	3	2	2	7
1.4 Structure		4	1			5	1	-	6
2. Languages	(10)	(17)	(10)	(8)	(21)	(66)	(5)	(3)	(74)
2.1 ADA				1	3	4			4
2.2 RTL/2 & CORAL				1		1			1
2.3 BASIC	1					1			1
2.4 PASCAL	1		1	1	4	7			7
2.5 FORTRAN	3	3	1		4	11			11
2.6 Assembler			1	1		2			2
2.7 Structured Programming	4	9	3	3	3	22	3		25
2.8 S/W Redundancy		1	1	1	3	6			6
2.9 S/W Metrics	1	4	3		4	12	2	3	17
3. Support Environment and Testing	(10)	(17)	(5)	(9)	(25)	(66)	(6)	(8)	(80)
3.1 Host Systems			3	1		4			4
3.2 Toolsets		4		1	1	6			6
3.3 Test Models	5	6		1	8	20	4	7	31
3.4 Correctness Proofs		1	1	1	8	11	2	1	14
3.5 Validation & Verifications	1		1	2	2	6			6
3.6 Path & Program Proving	3			1	3	7			7
3.7 Simulation	1	6		2	3	12			12
4. Operational Env't & Personnel	(5)	(5)	(2)	(5)	(5)	(22)	(3)	(4)	(29)
4.1 Psychological	3	2	2	1	3	11			11
4.2 Personal Environment	1	1			2	4			4
4.3 Hardware Failure	1			2		3	3	4	10
4.4 Documentation		2		2		4			4
TOTALS	(27)	(53)	(24)	(28)	(62)	(194)	(25)	(21)	(240)

Table 2. Analysis of Data Gathered from the Experiment

Analysis of CORRECT Answers

No. Characters	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Total
Mnemonic:																					
1 - Frequency	0	3	33	20	14	6	18	6	2	1	2	0	0	1	1	0	0	0	0	0	107
2 - Frequency	0	2	5	6	16	6	20	10	8	4	5	0	1	1	0	0	0	0	0	0	84
3 - Frequency	0	5	5	5	8	12	43	24	0	2	0	1	0	0	0	0	0	0	0	0	105
4 - Frequency	0	0	3	4	3	11	7	6	9	6	5	9	6	1	2	2	1	0	0	0	75
5 - Frequency	0	0	12	6	10	10	11	6	10	7	5	6	4	2	0	1	1	0	0	1	92
6 - Frequency	0	0	0	5	5	6	4	2	6	4	8	1	4	3	1	1	0	1	1	0	52
7 - Frequency	0	1	1	1	1	6	5	0	1	7	9	6	6	1	3	1	3	2	5	5	64
8 - Frequency	0	0	2	10	6	11	11	2	7	8	8	8	2	5	1	1	1	2	2	2	89
All - Frequency	0	11	61	57	63	68	119	56	43	39	42	31	23	14	8	6	6	5	8	8	668

Analysis of INCORRECT Answers

No. Characters	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Tot.
Mnemonic:																						
1 - Frequency	0	0	0	4	3	1	0	2	1	1	0	0	0	0	0	0	0	0	0	0	0	12
2 - Frequency	0	0	2	9	0	5	5	6	3	3	0	0	1	0	1	0	0	0	0	0	0	35
3 - Frequency	2	0	1	1	4	2	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	14
4 - Frequency	0	0	0	4	1	1	4	10	5	4	4	2	3	1	3	2	0	0	0	0	0	44
5 - Frequency	0	0	1	5	4	5	3	2	2	3	0	1	0	0	0	1	0	0	0	0	0	27
6 - Frequency	1	0	2	3	6	7	20	5	4	5	3	4	1	4	2	0	0	0	0	0	0	67
7 - Frequency	3	1	0	3	3	1	3	7	5	5	1	7	3	2	1	0	1	1	1	2	5	55
8 - Frequency	2	1	0	1	5	4	0	1	3	6	3	0	1	1	0	0	2	0	0	0	0	30
All - Frequency	8	2	6	30	26	26	35	36	23	28	11	14	9	8	7	3	3	1	1	2	5	284

Analysis of ALL Answers

No. Characters	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Tot.
Mnemonic:																						
1 - Frequency	0	0	3	37	23	15	6	20	7	3	1	2	0	0	1	1	0	0	0	0	0	119
2 - Frequency	0	0	4	14	6	21	11	26	13	11	4	5	1	1	2	0	0	0	0	0	0	119
3 - Frequency	2	0	6	6	9	10	12	46	24	1	2	0	1	0	0	0	0	0	0	0	0	119
4 - Frequency	0	0	0	7	5	4	15	17	11	13	10	7	12	7	4	4	2	1	0	0	0	119
5 - Frequency	0	0	1	17	10	15	13	13	8	13	7	6	6	4	2	1	1	1	0	0	1	119
6 - Frequency	1	0	2	3	11	12	26	9	6	11	7	12	2	8	5	1	1	0	1	1	0	119
7 - Frequency	3	1	1	4	4	2	9	12	5	6	8	16	9	8	2	3	2	4	3	7	10	119
8 - Frequency	2	1	0	3	15	10	11	12	5	13	11	8	9	3	5	1	3	1	2	2	2	119
All - Frequency	8	2	17	91	83	89	103	155	79	71	50	56	40	31	21	11	9	7	6	10	13	952

The Probability of the Correct Interpretation of a Variable

Nc	Pc	Nc	Pc	Nc	Pc	Nc	Pc
1	0.000	6	0.071	11	0.044	16	0.006
2	0.012	7	0.125	12	0.033	17	0.006
3	0.064	8	0.059	13	0.024	18	0.005
4	0.060	9	0.045	14	0.015	19	0.008
5	0.066	10	0.041	15	0.008	20	0.008

where Nc is the number of characters in the variable and

Pc is the probability of a correct interpretation of the variable.

Table 3. Data Used in the Plexus Calculations

	n1	n2	N1	N2	Cs	D	Sc	Nc	Nv	Nd	Np	Nsc	Nsd	Vs	Fvi	Fdi	Fpi	N^	V	H	P	r	E
Lang. (1)	3	5	4	6	64	2	5	88	3	2	1	12	2	26	2 1	1	1	16.36	30.0	528.0	41.8	92.1	7.9
Lang. (2)	3	6	4	7	64	1	3	43	3	3	1	15	3	26	2 1 1	1 1 1	1	20.26	34.9	258.0	38.0	85.3	14.7
Pascal (1)	5	18	19	38	96	14	14	312	13	5	10	26	7	33696	2 4 3 5 3 2 4 1 1 3 1 1 1	1 2 1 2 1	12 4 1 1 1 1 1 1 1	86.70	257.8	2054.5	483.0	76.5	23.5
(2)								161 (Declarations not included)										86.70	257.8	1060.2	248.2	76.6	23.4
BASIC (1)	5	16	19	39	68	1	14	216	7	9	9	36	9	26	5 4 2 3 2 5 3	1 2 4 1 1 1 3 1 1	4 1 1 2 12 4 1	75.60	254.7	1314.9	283.7	78.4	21.6
(2)								206 (Declarations not included)										75.60	254.7	1254	248.9	80.2	19.8
FORTRAN (1)	6	19	19	48	68	1	14	181	7	12	10	29	9	26	5 2 2 3 2 6 2	2 6 2 2 1 1 1 5 2 1 1 2 1	1 1 1 4 1 1 11	96.22	311.1	1101.8	304.1	72.4	27.6
(2)								156 (Declarations not included)										96.22	311.1	949.6	269.3	71.6	28.4

Note: The definition of the column headers is to be found in Chapter 4.4.