THE UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

UNIVERSITY OF
NEWCASTLE UPON TYNE

A Trade-off Model Between Cost and Reliability During the Design Phase of Software Development

by

Robert C Burnett

PhD Thesis

October 1995

Abstract

This work proposes a method for estimating the development cost of a software system with modular structure taking into account the target level of reliability for that system. The required reliability of each individual module is set in order to meet the overall required reliability of the system. Consequently the individual cost estimates for each module and the overall cost of the software system are linked to the overall required reliability.

Cost estimation is carried out during the early design phase, that is, well in advance of any detailed development. Where a satisfactory compromise between cost and reliability is feasible, this will enable a project manager to plan the allocation of resources to the implementation and testing phases so that the estimated total system cost does not exceed the project budget and the estimated system reliability matches the required target.

The line of argument developed here is that the operational reliability of a software module can be linked to the effort spent during the testing phase. That is, a higher level of desired reliability will require more testing effort and will therefore cost more. A method is developed which enable us to estimate the cost of development based on an estimate of the number of faults to be found and fixed, in order to achieve the required reliability, using data obtained from the requirements specification and historical data.

Using Markov analysis a method is proposed for allocating an appropriate reliability requirement to each module of a modular software system. A formula to calculate an estimate of the overall system reliability is established. Using this formula, a procedure to allocate the reliability requirement for each module is derived using a minimization process, which takes into account the stipulated overall required level of reliability. This procedure allow us to construct some scenarios for cost and the overall required reliability.

The foremost application of the outcome of this work is to establish a basis for a trade-off model between cost and reliability during the design phase of the development of a modular software system. The proposed model is easy to understand and suitable for use by a project manager.

Key-Words: *Software Cost Modelling, Trade-off Cost-Reliability, Development Cost, Software Testing, Markov Analysis*

Acknowledgements

First and above all I thank Almighy God. On the many occasions that I resorted to Him, I always found spiritual comfort and, somehow, reassurance in my doubts.

I express my heartfelt thanks to Prof. Tom Anderson, my supervisor, for his co-operation and invaluable and constructive criticisms which provided the foundation that enabled me to accomplish this task.

My deepest gratitude to Dr. Chris Phillips and Dr. Chris Woodford for their attention and their very helpful comments throughout this work.

I also wish to thank Alcides Calsavara, Antonio Marinho Barcellos, Ram Chakka, Shirley Craig, Trevor Kirby and Raimundo Macedo for their help and support.

Thanks to the Centro Federal de Tecnologia do Paraná - CEFET PR - (Federal Centre of Technology of Parana-Brazil) and Ponticia Universidade Católica do Paraná -PUC PR - (Pontifical Catholic University of Parana-Brazil) for their support and interest in this research.

My special thanks to Ana Lucia, my wife and friend, for being so supportive and understanding through all these years.

Finally, I acknowledge the financial support from the CNPq - Conselho Nacional de Pesquisa - Brazil, through the grant 200487/92-2, which allowed me to undertake this research.

I dedicate this work to my wife Ana Lucia, my sweetheart daughter Eliana

and

my parents Osmarina and George Burnett

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a software project the possibility of control and measurement stems from the fact that software development exhibits a characteristic of behaviour that enables us to predict in advance its various steps, having pre-defined phases which any project should follow. This characteristic can help in avoiding procedures that could result in overly expensive projects [58] and, in principle, enables us to establish effective cost control. These phases in the general structure of software system development may be characterised as follows [55]: requirement analysis, requirement specification, system design, implementation (coding), testing (testing and debugging) and operation (use). Within this framework the implementation and testing phases, for complex software systems, frequently present difficulties for project management with regard to cost control [7, 55]. There is a real risk that, during these phases, the cost of the software system could run out of control.

Software project managers recognise the value of adopting techniques which help to estimate the effort (for example: man-months) and cost needed to complete the development of a software system. Then, taking into account the estimation outcome and a specific upper limit for cost, the project cost could be brought under control.

Project managers need to be able to estimate how long a software project will take and how much it will cost.

In order to make that estimation it would be helpful to know, before beginning the implementation phase and taking into account the project profile (such as a required level of reliability), the amount of effort that should be allocated during implementation and testing. Indeed, the cost should be estimated as accurately as possible so that an implementation team with suitable skills can be selected in order to address the constraints of cost and required reliability. To help in this task of software cost estimation, methods have been proposed [7, 12, 43, 57] which yield an estimate of the amount of effort required for development, such as the number of people needed and the development schedule.

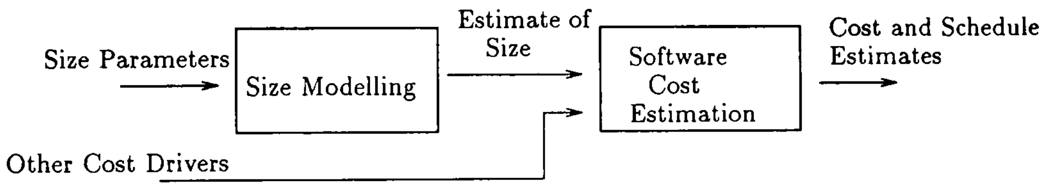

Figure 1.1: Software cost estimation

Several different methods for estimating the cost of developing a software module have been devised, such as those cited above; all of these methods basically utilize the structure of figure 1.1, where:

Size Modelling represents a group of checklists and/or algorithms, which support estimating the size of a module; [1]

Software Cost Estimation is the activity of estimating the cost that will be expended with the software development;

[1]Document [65] is a good example of this modelling approach, containing an extensive list of guidelines and mechanisms for estimating software size reasonably accurately.

Size Parameters refers, for example, to the number of functions to be executed, the number of physical files to be manipulated or the number of inputs/outputs that a module has to handle;

Estimate of Size is an estimate of the module size, which is usually obtained either in lines of code or function points, as discussed in section 2.2.5 ;

Other Cost Drivers refer to the use of software tools, reusability, complexity and characteristics of utilization (such as distributed processing) in the implementation of the module;

Cost and Schedule Estimates are the amount of time and person-period (for instance, man-months) necessary to develop the module.

Currently available software cost estimation methods do not usually consider the desired level of reliability as a cost driver, as will be discussed in chapter 2. However, as illustrated in [2, 45, 80], the cost of a software system is strongly influenced by reliability requirements; that is, a system with high required reliability needs more development effort and consequently costs more than if the same system had lower reliability requirements. These papers confirm that the cost of a system increases when the level of reliability required from it increases.

In spite of there being many software reliability models very few of them provide any guidance, before these phases begin, on how much effort (in relation to desired reliability) should be spent during the implementation and test phases. Research has concentrated on "release policies" for software systems, i.e., when to stop testing and deliver the system [44, 45, 53, 80, 83]. These policies do not assist the project manager in advance of implementation; they only provide guidance during testing.

In addition, matters dealing with reliability in a *modular* software system are scarcely treated in the literature; very few papers address the modularity issue, as is emphasized in [35, 45].

In this context, this work proposes a method for *estimating* the cost of development of a modular software system, given a desired level of reliability for that system, as well as a budget that represents the upper cost limit. Estimation is carried out during the design phase, that is, before implementation begins using historical data and the expected number of software faults for each module.

In this work the cost of development C_{dev} of a module is taken to be specifically the cost spent during the coding and testing phases only. As analysed subsequently, the estimate of the cost of testing is based on the level of reliability R required for the module, which is linked to the number of faults (introduced during the coding phase) which need to be removed (found and fixed) during the testing phase.

To connect cost and reliability a line of argument is developed which suggests that achieving a required level of reliability is strongly associated to the effort devoted to testing, which in turn depends on some intrinsic characteristics of the module under consideration. Based on this reasoning, we plan to estimate C_{dev} using formulas that enable us to represent the "chain" of relationships depicted in figure 1.2.

- Estimated Cost of Development (C_{dev}) is formulated here as the estimated cost that will be expended on the coding and testing phases. The latter, as shown subsequently, is associated with achieving a specific reliability level.

- Required Reliability for the System represents the overall level of reliability required for the software system, which is assumed to be known in advance of the design phase;

- Required Reliability of a Module is set at a level which enables a given overall required reliability for the system to be achieved, as developed in chapters 4 and 5;

- Characteristics of a Module are obtained from the requirement specification

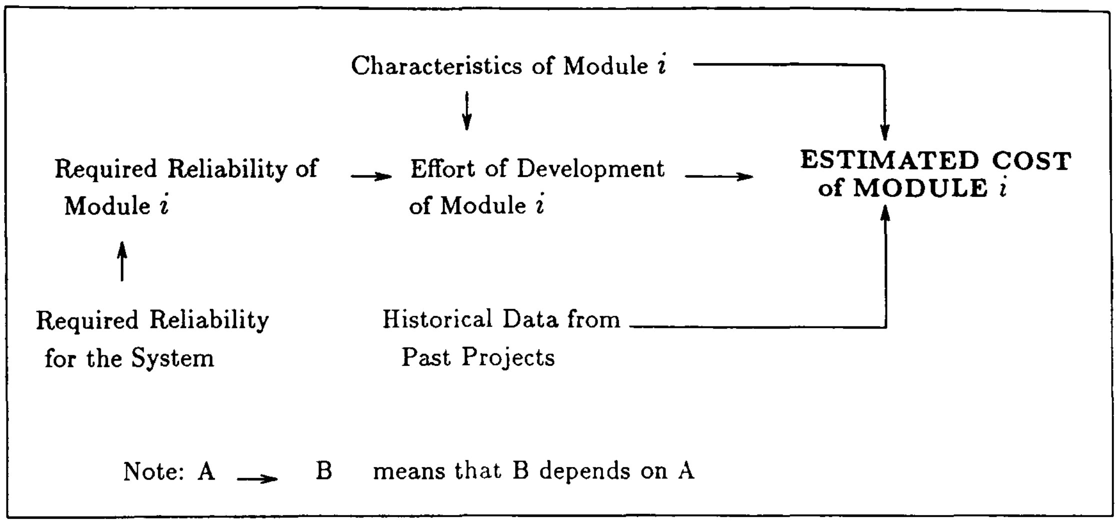
Characteristics of Module i
Required Reliability of Module i
Effort of Development of Module i
ESTIMATED COST of MODULE i
Required Reliability for the System
Historical Data from Past Projects
Note: A → B means that B depends on A

Figure 1.2: Relationships analysed in cost of development

for the software, and include various factors which can influence the cost of development, for instance, module size and the expected number of faults to be removed during testing;

- Historical Data from Past Projects allow us to formulate fundamental correlations between some of the data, using regression analysis;
- Effort of Development of Module is the effort needed for coding and testing the module, taking into account the number of faults that have to be found and fixed during the testing phase so that a required reliablity will be achieved.

Applying the procedures for estimated cost of development and overall estimated system reliability a trade-off model between cost and reliability is defined. Using the results of this trade-off model a project manager could plan the allocation of resources to the implementation and testing phases so that the estimated total system cost does not exceed the project budget and the estimated system reliability meets the required target.

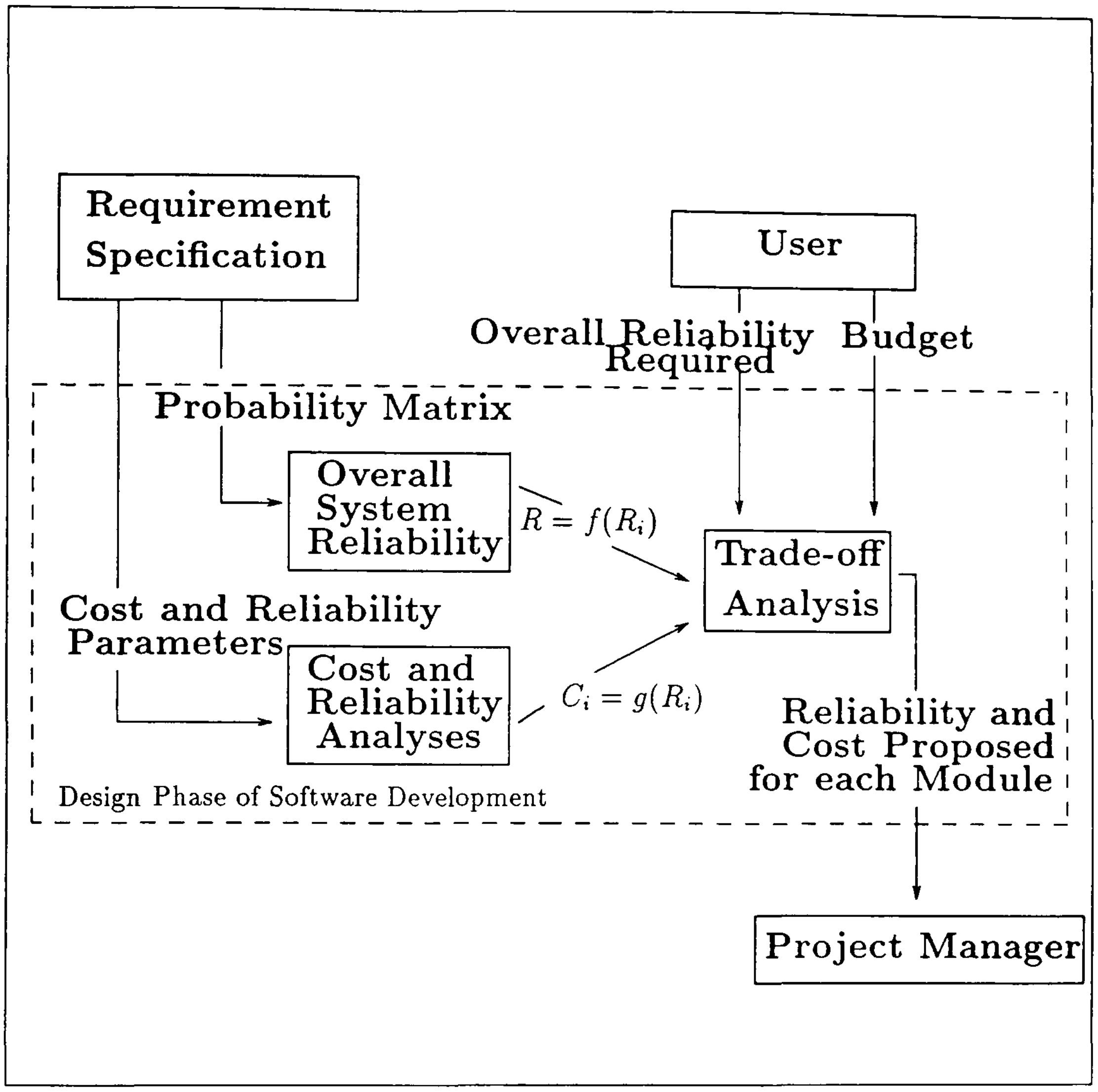
Requirement Specification
User
Overall Reliability Required
Budget
Probability Matrix
Overall System Reliability
$R = f(R_i)$
Trade-off Analysis
Cost and Reliability Parameters
Cost and Reliability Analyses
$C_i = g(R_i)$
Reliability and Cost Proposed for each Module
Design Phase of Software Development
Project Manager

Figure 1.3: Overview of the procedures in the trade-off model

The overall procedure to set up a trade-off between cost and reliability during the design phase is envisaged as shown in figure 1.3, where the

- *Probability Matrix* is the result of the analysis of the expected pattern of usage of the system under consideration, derived from a requirement specification. This transition probability matrix expresses the envisaged pattern of interaction between the modules.
- *Overall System Reliability* is the definition of a formula that will enable us to estimate the system reliability based on the target reliability of each module.
- *Cost and Reliability Analyses* represent the study of how to link cost and reliability during the design phase of software development. This link between cost and reliability is made through the number of faults that need to be removed during the testing phase so that the overall required reliability will be achieved.
- *Trade-off Analysis* represents a process of minimizing the total cost of development. As a result of this process the cost and target reliability for each module are obtained.

As shown in [35], the testing phase has to be managed wisely. To do this a project manager should have sufficient information to allow him to control overspend. The point of this work is to aid progress in this direction.

To address the points noted above this thesis is arranged as follows:

- Chapter 2 considers some of the methods available for estimating the cost of software development, focusing on how reliability requirements influence the estimate and how the effort of coding can be obtained, based on some published methods. This chapter also discusses methods applied in advance of the coding phase as well as during the testing phase.

- Chapter 3 defines how the cost of development is dealt with in this work. This will enable us to establish a trade-off between cost and reliability during the design phase of software development. A formula is proposed that links the number of faults to be fixed to a required level of reliability based on the expected number of faults present in the module before starting the testing phase. In addition, the factors that are involved in the effort of testing are defined and a proposal is made for estimating the effort of finding and fixing one fault during the testing phase. Then, a formula for the estimated cost of development is established using the results achieved in the previous sections.

- Chapter 4 proposes a formula to estimate the overall reliability of a modular software system, which is linked to the reliability of each module and the probabilities of transition between modules, using Markov Analysis.

- Chapter 5 develops a procedure to allocate the reliability requirement for each module utilizing the outcomes obtained in chapters 3 and 4 and using a minimization procedure, which enables us to construct some scenarios for an estimated cost and the overall required reliability. An example is also presented, using hypothetical data, which serves to illustrate the procedures developed.

- Chapter 6 focuses on the analysis in the formulas of cost of development and overall estimated reliability for the parameters employed in those formulas. A numerical expression of these sensitivities is made and a comparison of the sensitivity among the parameters is presented.

- Chapter 7 provides the conclusion and outlines future work.

Chapter 2

Cost Estimation Methods

2.1 Introduction

Software cost estimation is mainly concerned with providing a cost estimate before any lines of code have been written, taking into account the many factors that are thought to have a direct effect on cost. The outcome is an estimate of the effort, such as the number of man-months, required to develop the software, which converts directly into cost. The factors that enable us to estimate the effort are primarily the estimated size of the software and some adjustment factors called *cost drivers* [7, 12, 33, 43]. These cost drivers are applied so that the cost estimation takes into account some of the software characteristics, such as desired software reliability, software complexity and the programming language to be used, all of which are thought to influence the software cost.

We have chosen to widen the above characterization of software cost estimation in order to include also the estimation performed after the coding phase, even though these methods cannot strictly be considered as cost estimation methods. Nevertheless, these estimation techniques provide some interesting alternatives for associating cost with reliability. Hence it could be said that there are two ways of dealing with software cost estimation:

- *before* the coding phase, which aims to provide a cost estimate when no lines of code have been written or tested, and is based on historical data derived from previous projects; and
- *after* the coding phase, that is, during the testing phase using data that are collected during this phase (after this phase has progressed for a while), thus aiming at estimating the testing time in this phase, such that a desired value of software reliability can be attained. Based on this testing time, the estimated cost to complete the testing phase may be calculated.

In examining these two approaches, we focus on the factor that is well-known to be of paramount importance for cost estimation, namely, the *software reliability requirement.* This factor, considering a modular software system, can be defined as the probability that a module will operate according to its specification when called and will transfer control correctly when finished.

In the first approach (cost estimation before the coding phase) the reliability requirement is used to adjust the estimate of effort and consequently the cost (see section 2.2). In the second approach, cost estimation is treated by software release policies (section 2.3) which employ a required level of reliability in order to establish when to stop the testing. As shown subsequently, these policies can also be used in the estimation of cost using reliability as a major factor.

This chapter considers some of the methods available for estimating the cost of software development focusing on how reliability requirements influence the estimate. It may also help in understanding why certain solutions were employed in the model developed in this work; this model, as discussed on the following chapters, combines features of the two above mentioned approaches, estimating the cost before the coding phase, without requiring any data from the testing phase, but using the desired level of software reliability as a key parameter.

2.2 Cost Estimation Before the Coding Stage

In this section we summarize how software cost is currently estimated before the coding phase, using different approaches, and discuss how a reliability requirement is/is not taken into account in these approaches. A discussion on software size estimation methods is also presented. It should be emphasized that this chapter is *not* a review of the literature on cost estimation models but specifically a survey as to how the reliability requirement is/is not handled in this estimation. More details about cost estimation models can be found in [7, 12, 16, 27, 32, 43].

2.2.1 How to estimate the cost

The cost of developing software, here represented by C_{dev}, is directly related to the effort E_{dev} (expressed, for example, in man-months) spent in that development. This effort has in turn been said to be mainly dependent on the size of the software [7, 12, 32, 55].

This dependency is represented in two ways: either using essentially historical data from past projects or based on so-called "theoretical data". The first of these is discussed below, whereas the latter is treated in section 2.2.6. Other known and well-referenced cost estimation models, which have some similarities with the models analysed here and in section 2.2.6, are outlined in section 2.2.7.

In sequence we briefly outline how an estimate of the effort of development E_{dev}, which is utilized for estimating the development cost C_{dev}, can be obtained. The outline given here, which is based on published works, should be seen as a very brief introduction on how to obtain this parameter. An example is also presented on how to estimate the effort of development using (hypothetical, but realistic) historical data. A view in more depth can be acquired from the references indicated.

2.2.2 Expression for effort of developing

It is well-known that the effort needed to implement a software module, here represented by E_{dev}, is strongly related to the size of the module, as can be seen in most of the software cost models that are available in the literature [7, 12, 43]. The effort E_{dev} can be expressed as:

$$E_{dev} = (a \cdot S^b) \cdot D(\underline{X}) \tag{2.1}$$

where

- a and b are constants usually derived by regression analysis between E_{dev} and the software size S of previous projects. It might be suggested that one difficulty in applying these models is that either a or b might not be suitable for a particular user's installation, and this would produce inaccurate effort estimates. Thus, it would be recommended that the user should establish a and b for their own software systems. The result would be a model calibrated for the user's installation. A typical method of carrying out this task is to use standard regression analysis[1] to determine these values, using the precedent of past projects, where effort in person-period and system size is available.

- S is the system size as estimated during the design phase, either in function points (FP) or thousands of line of code (KLOC). In section 2.2.5 there is a

[1]For details, see, for example, [12].

discussion of how the parameter S can be estimated.

- $D(\underline{X})$ is an adjustment multiplier that depends on cost-drivers, represented here by the vector $\underline{X}$. Development effort is clearly influenced by other attributes referred to as effort (or cost) drivers, for example, module application domain[2] and technical complexity [7]. Each of these attributes should be evaluated on a suitable scale producing a figure $D(\underline{X})$ that represents the level of influence of drivers on the effort. Depending on this value for $D(\underline{X})$ the effort E_{dev} can be adjusted appropriately.

  An example of this multiplier can be seen in Boehm's COCOMO model [7, page 118]. Recent works, such as [33], suggest that in many development environments a small sub-set of Boehm's cost-drivers would be sufficient.

  In this work we merely assume that there exists some $D(\underline{X})$, corresponding to Boehm's cost-drivers and tailored for the environment, that enables an appropriate adjustment of E_{dev}. Subsequently, we analyse how software cost models deal with a reliability requirement, discuss in detail some characteristics of $D(\underline{X})$, and more specifically examine how software reliability is handled.

2.2.3 Example of how to obtain E_{dev}

An example of how to establish a and b using hypothetical (but realistic) data is outlined below.

Suppose that an installation has a historical precedent providing data on coding effort in man-months and program size in KLOC for a series of programs. Typical values are shown in Table 2.1, which has been adapted from [7]. All programs (used in Table 2.1) belong to the same category (for example: business application) and

[2]For instance, business and real-time applications.

Program	Coding Effort	Size
1	4	5.0
2	3	4.5
3	5	6.5
4	11	12.5
5	5	7.0
6	3	4.2
7	11	13.5
8	12	15.0
9	7	8.9
10	7	9.0

Table 2.1: Values from past projects

with $D(\underline{X}) = 1.0$[3].

Applying regression analysis between effort and size we obtain the following formula for the effort of coding (because we are using coding effort as historical data):

$$E_{dev} = 0.634 \cdot S^{1.09} \tag{2.2}$$

To examine the accuracy of this formula[4] we can calculate the differences between the actual values for effort that are indicated in Table 2.1 and those estimated using the formula (2.2). These differences are shown in Table 2.2.

The average magnitude of the absolute relative error [16, 30] is a suitable measure

[3]So, we are hypothesizing that the influence of each cost driver on the effort is "nominal" [7].

[4]We are assuming zero-intercept, that is, there is no constant overhead.

Program	Estimated E_{dev}	Difference	% Difference
1	4.336	0.336	+6.7
2	2.733	-0.267	-5.6
3	5.123	0.123	+2.5
4	5.106	0.106	+2.1
5	4.173	-0.287	-6.0
6	2.97	-0.03	-0.6
7	11.18	0.18	+3.6
8	11.87	-0.13	-2.6
9	7.131	0.131	+2.6
10	7.046	0.046	+0.9

Table 2.2: Differences between actual values and estimates

for determining the quality of performance of the formula (2.2). This value is given by

$$\frac{1}{n} \cdot \sum_{i=1}^{n} \left| \frac{Y_i - Y_i^{est}}{Y_i} \right| \tag{2.3}$$

where

Y_i is the actual effort of development (Table 2.1) and Y_i^{est} is the estimated value of Y_i (Table 2.2);

Applying formula (2.3), the foregoing absolute average relative error magnitude is then 0.04135, i.e., about 4%.

This value is much less than the 0.25 that is recommended in [16, page 148]. Hence, for this example, the predictive quality of the formula (2.2) is quite good, and as a consequence it may be suggested that the hypothetical user of this estimation formula may have reasonable confidence in predictions obtained using the formula for other programs with similar characteristics of development.

In this example the formula used for E_{dev} was simply $E_{dev} = a \cdot S^b$, because, as hypothesized, $D(\underline{X}) = 1.0$. It can be observed that the value obtained for b is of the same magnitude as has been found in other cost models[5], such as are shown in [7, 12, 43]. However, the value obtained for a is considerably smaller. This is because we are analysing just the *effort of coding* while the references cited (usually) estimate the effort that will be spent in total during design, coding and testing. In other words, the effort of development shown is equal only to the effort of coding, hence the term "coding effort" in Table 2.2.

2.2.4 Reliability requirement as a cost driver

Examining the expression (2.1) for the effort of development E_{dev}, we observe that if we stick to this formulation and we want E_{dev} to take the reliability requirement into account, then there is only one opportunity–which is to use this requirement as a cost driver.

As can be seen subsequently, there is no reference in the software cost models analysed as to the influence that the software reliability requirement might have on software size during the coding phase. It might be suggested that the influence, if any, of reliability requirement on software size should be dealt with during the requirement specification phase, that is, prior to the design phase. This aspect is beyond the scope of this work.

To analyse the foregoing alternative, we focus our discussion on Boehm's COCOMO model [7]. The reason for doing so is that COCOMO is the best known

[5]We should bear in mind that b may have a wide variation among different environments (due to non-linearity between effort and size), as can be seen in [4].

and best documented cost model, widely used as "inspiration" for other cost models [12, 32, 43], and constitutes a typical specimen of a cost model that uses the concept of cost drivers.

The reliability requirement, which is termed "Required Software Reliability" in COCOMO, is one out of 15 cost drivers defined there[6]. The effect of the cost drivers in the cost estimation is determined by assigning ratings for each factor on a scale (very low, low, nominal, high, very high and extra high), and then associating a numerical value[7] to each rating, as shown in Table 2.3 . These numerical values, whose product constitutes $D(\underline{X})$ and consequently serves as a multiplier for E_{dev}, allow us to adjust the cost depending on the significance of the cost driver to the software being estimated (once obtained, these 15 numerical values (scores) are multiplied together to give $D(\underline{X})$).

To characterize the reliability requirement as a cost driver Boehm uses a concept of "nature of loss". The requirement has to be rated depending on the loss that a user would suffer if the software yielded a wrong outcome. The scaling factors proposed in COCOMO to incorporate the reliability requirement are presented[8] in Table 2.3.

Despite the expression "Required Software Reliability" used in COCOMO, which refers to the general behaviour of the software, rather than to a specific value for the required reliability, there is no unequivocal correlation between "Impact of a wrong outcome for a user" with a desired level of reliability.

However, it may be conceded that (without undue concern for accuracy) a high or low figure for the required reliability is likely to be influenced by the consequences

[6]COCOMO has the expression for cost estimation as in equation (2.1).

[7]These numerical values were established by Boehm empirically.

[8]There is no "Extra High" rating for the reliability requirement in COCOMO.

Impact of a wrong outcome for a user	Ratings	Numerical Values
Slight inconvenience	*Very Low*	0.75
Easily recoverable loss	*Low*	0.88
Moderate loss	*Nominal*	1.00
High financial loss	*High*	1.15
Risk to human life	*Very High*	1.40

Table 2.3: Classification of reliability requirement, according to COCOMO

that a wrong outcome could yield for the user. On this basis, working with ratings would only give us a very crude idea of the influence of a required reliability in the cost of development.

Indeed, it may be noted that even if two software modules required different levels of reliability, they could be classified with the same rating[9]. If this was the case they would have the same adjustment factor for the estimated effort (when only the effect of the reliability requirement considered). Nevertheless, in order to increase slightly the reliability of a software system, more testing would need to be undertaken, which would presumably imply more effort and consequently more cost. The ratings proposed in COCOMO are too coarse to accommodate this situation.

The kind of relationship between ratings and numerical values, shown in figure 2.1, can arguably be attributed to the fact that as the reliability requirement becomes "tighter" (tends to "Very High"), a great deal more effort (and consequently cost) is needed than would be the case if the requirement was less stringent. Hence, increasing the reliability requirement from "Nominal" to "High" is less costly than

[9]Furthermore, we see in Table 2.3 that there exists only a 1.87 : 1 ratio between "*risk to human life*" and "*slight inconvenience*", which seems to be rather too small.

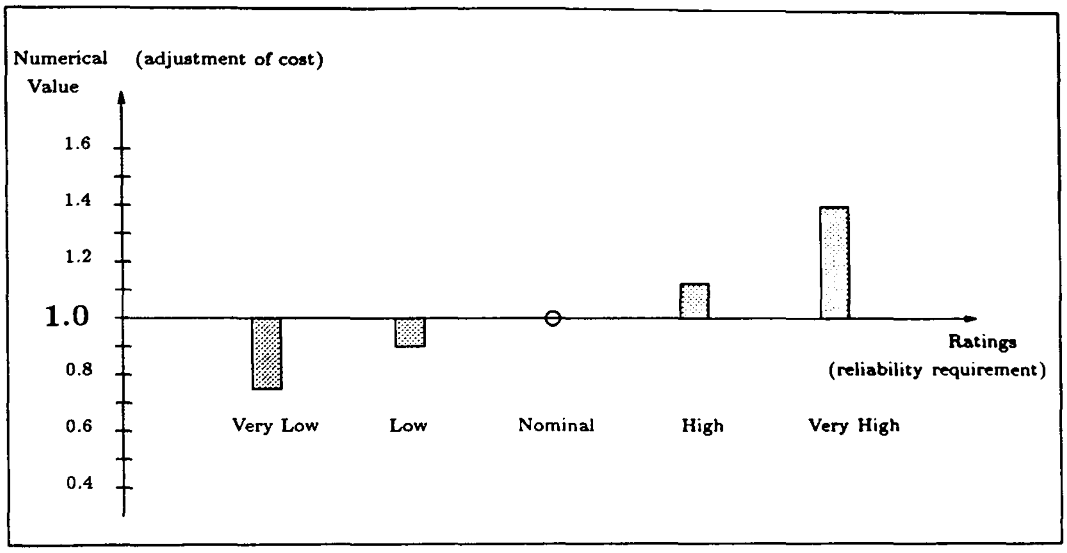

Figure 2.1: Relationship between reliability and cost, according to COCOMO

from "High" to "Very High". In other words, there is a non-linear variation between "Very Low" and "Very High" which could be justified by the fact that the latter would demand more testing effort than the former which, obviously, would mean more cost.

It can reasonably be inferred that if the ratings were replaced by a more precise notion, such as an explicit figure for the required reliability, then we could obtain a more accurate relationship between the precise values for reliability and the factors utilized for cost adjustment.

2.2.5 Software size estimate

As shown in the previous sections, the software size is a key factor in the cost estimation methods. Thus, it is worth discussing how the software size is estimated during the design phase.

An estimate of the size of software is usually made using one of two units—lines of code (LOC) or function points (FP). The latter is discussed in some detail here, and then the former is briefly summarized.

Besides its popularity as a software metric (as has been emphasized in [24]), the reason for our interest in the function point approach is that the FP method allows us to take some size drivers into consideration in estimating software size–the reliability requirement may be a case in point.

However, there is an obvious difficulty in estimating the size of software during the design phase when some details are not known about its characteristics. In addition there is the problem of defining what source lines of code means [59]. To avoid this problem, one solution is to base the total development effort estimates on the "functions" that a software system has to perform instead of using the LOC as a measure of software size. The FP method tries to provide a less "subjective" manner of estimating software size. Besides, it should be emphasized, the number of LOC can be estimated from the number of FP, as will be shown later.

The FP method enables us to estimate the software size through "function points" based on some characteristics of the information processing and a technical complexity factor [1, 28, 29, 60, 73, 74]. In addition FP are available for measurement rather than estimation at an earlier stage in the development process than the source lines of code.

Another reason why LOC seems to be a weaker method than FP is that it is difficult to use analysis and analogy for sizing a software system as is usually employed for estimating the size in lines of codes, because often our past experience may not give us the capability for predicting the software size. It may be suggested that a case in point is when we are distributing functions that were previously centralized.

A brief summary of the best known FP methods follows.

- **Albrecht's function point method**

To cope with the problems of estimating software size, Albrecht [1, 73, 55] proposed that the "functions" the software should perform could be used to measure software size, where for each function a weight is attributed. Taking into account the relative weights, the outcome of Albrecht's method indicates the software size in a unit called *function points*. This unit does not directly quantify the program size in the way that lines of code do, and is said to be a measure of the *problem size* [73].

Albrecht developed a method of estimating the number of functions (as seen by the end-user) that a software system has to carry out, based on the number of inputs utilized, outputs produced and the number of interfaces employed. The components selected to define the software size in the Albrecht model are: number of user inputs, number of user outputs, number of user inquiries, number of files and number of external interfaces.

Each component is associated with a weight[10], which can be classified as "simple, average or complex" depending on the number of data elements in each component, that reflects the relative value of the component to the user. The weighted sum of these components is called "Function Points". This total of *function points* obtained, according to Albrecht, needs to be adjusted further to take into account some characteristics of the development environment. After this adjustment the final outcome of *software function points* is obtained.

Because of the importance of this method for size estimation, some studies have been published [28, 29, 46, 49] that discuss how to obtain FP more accurately than by using Albrecht's method. Other studies [60, 73] show some extensions to the original Albrecht function point method and review its weak points [25, 73]. Despite the criticisms that have been made of its features, FP has become an established

[10]It should be noted that the weights were determined by "debate, trial and error".

Ratings	Numerical Values
Not Present or No Influence	0
Insignificant Influence	1
Moderate Influence	2
Average Influence	3
Significant Influence	4
Strong Influence	5

Table 2.4: Scale for evaluation of the characteristics

industry standard of size measurement for software (as stressed in [28]) and has generated a specific research field.

We focus here on how the adjustment factor works in Albrecht's method. The adjustment factor is composed of a group of 14 characteristics that can be considered as *size drivers*:

⋆ Data Communication; Distributed Functions; Performance; Heavily Used Configuration; Transaction Rate; On-line Data Entry; End-User Efficiency; On-line Update; Complex Processing; Re-usability; Installation Ease; Operational Ease; Multiple Sites; and Facilitate Changes[11].

Each one these characteristics has to be evaluated on a scale from 0 to 5 as shown in Table 2.4. The values attributed to the 14 characteristics are added together producing a total, which represents the overall degree of influence of the environment on the software size (estimated function points).

[11]Note that the software reliability requirement is not included in these characteristics.

LANGUAGE	LOC per FP
Assembly	300
C	130
Cobol	100
Pascal	90
Modula 2	80
Ada	70
Object-Oriented Language	30
Fourth-generation language	20
Code generators	15

Table 2.5: Number of lines of code to build one Albrecht function point

The adjustment factor is then formulated to be equal to $0.65 + 0.01 \sum_{i=1}^{14} V_i$, where V_i is the numerical value attributed for each characteristic. So, the overall degree of influence can give an adjustment factor of $\pm$ 35% in the number of FP.

However, we usually need to utilize LOC instead of FP as a direct measure of size. A straightforward way to do this would be to determine the size in FP and for each environment to use its own method to convert FP to LOC. This can be done using regression analysis between FP and LOC, as shown in [1].

Table 2.5, which is reproduced from [55], gives a *rough* estimate of the relationship between LOC and FP, for the main programming languages. It can be seen, for instance, that one function point would require 130 LOC in *C* language.

- **MARK II function points**

A well-known variant of the Albrecht function points is Symons' MARK II [73, 74] which offers a different approach to estimating the number of function points and to evaluating the adjustment factor. Let us examine particularly the adjustment factor since, contrary to Albrecht's method, this can be calibrated in the environment of the user.

- ⋆ The adjustment factor in MARK II is composed from the same 14 characteristics present in Albrecht's method plus five additional ones—requirements of other applications, special security features, direct access for third parties, documentation requirement and user training facilities.

Again, the reliability requirement is not explicitly cited. However, a key difference is that any other additional characteristic suggested by the user can be defined for this adjustment. The procedure of adjustment is carried out as in Albrecht's method as described previously.

Hence, we may include reliability requirement as a new characteristic, using the same ratings as defined in Table 2.4.

Examining the ratings shown in Table 2.4 and considering the inclusion of the reliability requirement as allowed in MARK II, the influence of the reliability requirement would be linear, which does not seem to match with the reality of software development.

• Function points and software science

Albrecht's function point method and MARK II are intended to deal with business applications, where the internal complexity of the system is mainly due to the process of validation and interactions with stored data. This could limit their validity for applying them in scientific or technological systems, where the internal complexity is the hard core.

To overcome this possible limitation, Reifer has proposed the *Asset-R* (Analytical Software Size Estimation Technique Real-Time), described in [60], which is thought to be useful in business data processing, but is mainly applicable to scientific and real-time systems, combining the concepts of function points and software science. A summary of this model follows.

The basic formula that estimates the size is

$$S = ARCH \cdot EXPF \cdot ((FP_{(adj)} \cdot LEC) + MVOL)^{RF}$$

where

- ⋆ S is the size of the system in source lines of code;
- ⋆ $ARCH$ is the *architectural constant* (which was derived empirically), a value depending on the system architecture, for example, centralized ($ARCH = 1.0$), distributed with central data-base ($ARCH = 1.8$) or fully distributed ($ARCH = 2.1$);
- ⋆ $EXPF$ is the *expansion factor for size drivers*, including the following drivers: requirements volatility, data base size, degree of real time, use of modern programming techniques, use of software tools, analyst capabilities, applications experience, environment experience and language experience. For each driver there is a range of numerical values associated with it, which were derived by Reifer "through extensive statistical analysis of existing sizing data bases". Note that the reliability requirement is not included as a size driver in Asset-R.
- ⋆ $FP_{(adj)}$ is the count of *function point adjusted*, which is obtained similarly to Albrecht's method, but using other weighting factors.
- ⋆ LEC is the *language expansion factor*, that is, the number of lines of code that are required to implement one function point. Thus, the product $(FP_{(adj)} \cdot LEC)$ provides the conversion of function points counts to lines of code. This factor LEC depends on the language that the software will be coded as shown in Table 2.5.

- ⋆ $MVOL$ is the *mathematical volume*, that is, the size estimators, either number of operators (any symbol or keyword that specifies an action) and operands (any symbol used to represent data, as well as, variables, constants, labels and most punctuation marks), or number of algorithms;
- ⋆ RF is the re-use factor. This factor is usually set to 1, but its calculation "is internal to the system" [60].

For scientific systems the number of function points adjusted $FP_{(adj)}$ is calculated using: the number of inputs, the number of outputs, the number of master files, the number of modes, the number of inquiries and the number of interfaces. For real-time systems two other parameters are included—the number of stimuli (response relationship) and the number of rendezvous are added. For these two types of application the formula does not contain any weighting factors or complexity adjustment factor[12].

If we need to estimate the size in lines of code for software classified as real-time or scientific and want to take advantage of the characteristics of Asset-R, we should bear in mind that the model does not take into consideration any reliability requirement as a size driver.

- **Size in lines of code**

If a home-made procedure for sizing software is preferred it is worth reading references [65] and [67]. The former is a document from the Software Engineering Institute (SEI), at Carnegie Mellon University [65]. which covers mechanisms for defining and estimating software size; the outcome achieved is a measure of the size

[12]Rook indicates in [62] that scientific systems should also utilize the weight factors, whereas Reifer states just for business data processing.

of the source code[13]. The latter deals with the subject of software metrics, where code metrics is a case in point.

It should be noted, as highlighted in [33], that if size is adjusted by some factors we should not use cost drivers based on those same factors, in order to avoid adjusting E_{dev} twice with the same factors.

2.2.6 Cost estimation using theoretical data

For this class of cost estimation model we have chosen to discuss Putnam's model [57, 58][14] since, as emphasized in [32], this model can be regarded as a typical and well-known method based on theoretical data; it enables us to make decisions about cost, time of development and the risks of software development.

The basic assumption in this model[15] is that manpower utilization during software development follows a Rayleigh-type curve[16]. This model is said to be theoretically based because it is supported by mathematical laws that the software development process is assumed to follow [12].

[13]Other references that may serve as starting points for creating a software sizing method are [23, 36, 72], which contain some checklists to help one make better estimations, and [78], which describes an approach to software size estimation, based on some factors that affect the software size.

[14]In [58], which represents a series of three articles about this model, Putnam describes his model in detail. A compact perspective on this model can be found, for example, in [12, 32, 43].

[15]The software product called SLIM, which stands for Software LIfe-cycle Methodology, was developed by Larry Putnam in the late 1970s and incorporates his approach to cost estimation, being a direct application of his model.

[16]The Rayleigh-curve is characterized by the fact that the curve increases rapidly towards a peak, after which it steadily decreases towards zero. A discussion of the behaviour of this curve can be found in [32, page 508].

A very brief summary of some results obtained from Putnam's model are:

- **Effort of Development (E_{dev})**

$$E_{dev} = 0.3945 * \frac{1}{T^4} * \left(\frac{S}{C}\right)^3$$

  ⋆ T is the development time in years;

  ⋆ S is the estimate of software size in LOC;

  ⋆ C is the technology factor. It reflects the effect on productivity of numerous factors, such as hardware constraints, program complexity, personnel experience levels and the programming environment. Putnam has proposed using a discrete spectrum of 20 values for C, ranging from 610 to $57,314$. A value for C may also be determined from historical project data.

  From the equation for E_{dev} above, the most important observation is that for a software product of a given size and fixed development environment the effort varies inversely as the fourth power of the development time.

  Despite the fact that it is well-known that human effort and time cannot be traded directly in a software development, it remains to be seen under what conditions the above relationship may be valid[17].

- **Difficulty Metric (D)**

  The constant D takes on discrete values corresponding to the difficulty of the software to be developed (hardware constraints and programming environment, for example).

[17]It is said in [32] that "this relationship has been strongly disputed by researchers. Putnam himself reported investigating 750 software systems and found that it held for only 251 of them".

With $K = \frac{E_{dev}}{0.3945}$ then

$$D = \frac{K}{T^2} = \frac{S^3}{C^3 T^6} = \left(\frac{S}{CT^2}\right)^3$$

According to Putnam's model, for a software of high complexity, the value of D would be about 7.3, while for relatively straightforward software the value of D would be about 27. The Putnam model has six discrete values for D ranging from 7.3 to 89.0.

Putnam's method intends to bring the problems of estimating, scheduling, and project control within reasonable limits, attempting to convert an estimate of system size into effort, and consequently cost.

We may clearly see Putnam's method as a COCOMO-like model, where the parameter C would have the same function as cost drivers. Once again, there is no discussion in this model as to how to include a software reliability requirement as a factor which could influence the cost. The difficulty metric D is not related to the reliability requirement either.

2.2.7 Other cost estimation models

In this section we highlight some other cost estimation models that have particular characteristics or are well-referenced.

Firstly we outline ESTIMACS, a software product which was developed for a consulting company, being a proprietary model. Therefore, its internal details, such as the equations used, are not available. Keremer [27] makes an assessement of ESTIMACS, indicating that it is one of the most used software cost estimation products. A summary of its characteristics, based on the description of [32, 62], follows.

As input ESTIMACS has some *size variables* which contain some similarities with the input parameters for Albrecht's method and MARK II, ***product variables***, which could be seen as the cost drivers, and other ***environment factors***.

Among the product variables there are constraints related to reliability requirements. However, the references [32, 62] do not contain any more detail about this aspect. Thus, ESTIMACS is cited here only as a reminder that this software product might include some treatment of the reliability requirement.

As output, ESTIMACS produces the effort (in man-hours), size in LOC and FP, and cost among some other results. There is no reference in these results to anything related to the reliability requirement.

Secondly, we very briefly mention the software product called SOFTCOST[18], which was developed at the Jet Propulsion Laboratory and which is said to be an attempt to gather the best features present in other models.

SOFTCOST assumes that there is a linear relationship between software size and the effort of development. This product has a great many more factors of effort adjustment (cost drivers) than COCOMO, but without, seemingly, producing a better result. The input required and outputs produced do not indicate any relationship with a reliability requirement.

Therefore, glossing over its features, it is merely noted that SOFTCOST does not have any special treatment for the reliability requirement.

The Walston-Felix Study [12, 32] is an early model of software estimation (developed in 1977), which works with the same structure for effort of implementation as that shown in section 2.2.2. This model is considered important because it identifies 29 characteristics that should be taken into account as possible cost drivers.

[18]A concise analysis of this software cost estimation model can be seen in [12, 32]. A comparison between SOFTCOST and COCOMO is included in [12].

This study and set of characteristics seem to have had a considerable influence on Boehm's model. However, as can be seen in [12, page 244-245], there is no evidence that the software reliability requirement is considered in this early model.

The Bailey-Basili meta model [3] is another known model (or rather, methodology) that contains some equations for effort and considerations for cost drivers. This model has some similarity with the concepts of COCOMO, where the latter could be seen as far more complete than the former. As emphasized in [32], its overriding contribution seems to be the suggestion of a methodology that may help in the task of building one's own software estimation model, which is also highlighted in [12] as being an alternative that is worth further exploration. As for the other models, there is no specific focus on reliability requirements for the cost drivers handled.

To avoid becoming too repetitive, we simply cite several other software products that also have features for software cost estimation but seemingly do not include any special features for dealing with software reliability requirements: PRICE SP [12, 32], MERMAID [61] and COPMO [12].

2.2.8 Summary

If some conclusions can be drawn from these analyses of some relevant software cost estimation models that are applied before the coding phase, we should stress:

- None of the models deal directly with a figure of required reliability.
- Some of the models enable some handling of the reliability requirement, but use a very limited approach as to how this requirement should be taken into consideration.
- Approaches for obtaining the software size are often utilized in cost estimation but none of these approaches use the reliablity requirement as a size driver.

- To allow for requirement reliability in software size estimation methods, it appears that the best option might be to build one's own sizing method.
- A software cost estimation procedure that makes use of relevant features of good existing models (the aim of SOFTCOST) but with the inclusion of a focus on reliability requirements would seem to be a valuable contribution to this subject of cost estimation.

The outcome of this thesis is a step in this direction.

2.3 Cost Estimation During the Testing Stage

This section surveys briefly those cost estimation procedures that are employed *during* the testing phase, and can make use of a value for the required software reliability. To do this, we outline three representative approaches for "software release policies", which enable us to estimate when to stop the testing phase and transfer the software to operational use, taking into account either an estimate of cost, or of required reliability, or both.

As shown subsequently, these policies may enable a cost estimate to be associated with reliability.

Determining when it is best to stop the testing phase, which is referred to in the literature as *the optimal software release policy*, is the main approach for verifying whether it is feasible or not to define a trade-off between cost and reliability during the testing phase [21, 37, 45, 53, 79, 80, 83]. In this approach a project manager verifies whether the required trade-off is feasible and when it is no longer relevant to continue the testing stage. The outcome reached by the chosen policy fixes the software release time, i.e., the total testing time, for which both reliability and cost requirements may be considered.

2.3.1 Software release policies

As is well-known, a significant matter of practical concern during the software testing phase of software development is to know how to achieve a compromise between the requirement for reliability and the cost of obtaining that reliability. In fact, this matter arises in finding out what level of reliability is achievable within the available budget. This can generically be termed a trade-off of cost against reliability during the testing phase.

As cited in [51], in any application the number of distinct input combinations that one would usually need to validate a software system is enormous, which is said to result [83] that the longer the software is tested the more reliable it tends to be. Thus, the testing phase can involve an enormous amount of effort, in terms of human resources and time, in order to produce more reliable software. Because of this it is vital to establish a deadline at which to stop the testing stage, taking into account the fact that there is a point beyond which the cost of obtaining significant improvements in reliability may rise significantly.

Since the early 1980s research has been conducted in this area, with the objective of determining the "optimum" time[19] when testing should stop and the system could be considered ready for operational use (see, for example, [34, 44, 51, 53, 79, 80, 82, 83]). To achieve this aim, two main criteria have been utilized—required reliability and expected cost; using these criteria, the optimum testing time for the system is obtained, i.e., when to stop testing and deliver the system to the user.

In determining the optimum release time, there are two main approaches:

i Reliability and cost criteria are considered separately. The testing time is

[19]This expression "optimum" only represents the optimum stopping time in the sense that it is meant to denote the best (minimum) required time to stop testing in order to achieve the required reliability, considering all of the factors involved.

established from either reliability or cost requirements. An example of this approach is given in [53];

ii The testing time is dependent on the relationship between cost and reliability. In these models a function is formulated considering cost and reliability requirements, so that the testing time obtained is that which yields the required reliability. An example of this approach is given in [80].

When the reliability criterion is to be utilized, there are two ways of working with the reliability requirements:

a Based on an acceptable number of remaining faults in the software. In this case testing is terminated when the estimated number of faults remaining is lower than a pre-established number [83];

b Based on an acceptable failure intensity level, that is, a specific value for reliability. In this case, the testing time problem is usually formulated using a software reliability growth model–SRGM[20] (see [6, 41, 42]).

Among the many software release policies that have been published (see references cited), we choose to summarize three of them, since these policies may be seen as typical of models in this area. However, each of these models assumes a particular distribution for the manifestation of faults, i.e., a specific software reliability growth model (SRGM). If an analysis in depth of were to be developed then this would need to consider other types of SRGM, such as those shown, for example, in [6, 42].

[20]A model used for software reliability assessment during the testing and operational phases is called a software reliability growth model. This is only true if the model assumes that software faults are fixed when found.

2.3.2 Goel-Okumoto model

The Goel-Okumoto model [21, 53] is an early and (perhaps) the most referenced approach to the optimum software release time policy; its concepts are widely employed in many other relevant software release models. This model does not take into account the reliability and cost criteria simultaneously. Hence, the user of the model has to estimate the testing time based on either reliability *or* cost requirements.

- **Using reliability criterion**

The criterion used is to stop testing when the predicted reliability at a specified time t, during the testing phase, is equal to some required value. Thus an equation is formulated to express the reliability required R, as a function of t, testing time T and the cumulative number of faults found and fixed $m(t)$. The required value T is then found, yielding the final formula.

It is shown that

$$m(t) = a\left(1 - e^{-bt}\right)$$

$$R = exp\left[-m(t)e^{-bT}\right] \tag{2.4}$$

where

⋆ t is the period of time during the testing phase that is utilized for making the estimation;

⋆ $m(t)$ is the expected number of software faults found and fixed by time t;

⋆ a represents the expected number of software faults to be found and fixed in total;

* ⋆ b is the fault detection rate during the testing phase[21];
* ⋆ R is the reliability required;
* ⋆ T is the testing time sought;

Then, rearranging the equation and solving for T, we have

$$T = \frac{1}{b}\left[\ln m(t) - \ln\ln\frac{1}{R}\right]$$

The user at time t (again, referring to the time spent in the testing phase) estimates the reliability based on data collected until that moment. T is measured in the same units as t, e.g., days, weeks, months, etc.

With the formula above for testing time the user can verify the sensitivity of T in relation to R and $m(t)$. In others words, the user can determine when a long testing time T is required to obtain a highly reliable software system, or when the compromise sought is not feasible.

As can be seen from the above formulation, there is no parameter of cost in this first approach. However, knowing the time T, i.e., how long the testing phase is estimated to last so that a required reliability R can be achieved, means that a value for the estimated cost in that phase may be obtained.

• Using cost criterion

The time spent in software testing and debugging delays the transfer of the system to the user and consequently results in a higher development cost. The aim in this second approach of the Goel-Okumoto model is to determine the optimum testing time to minimize the cost, considering all of the factors involved. Another aspect

[21]The term (ae^{-bt}) represents the expected number of remaining faults.

considered in this cost model approach is that the cost of finding and fixing a fault is supposed to be much less during testing than during operation.

The following variables are used:

- ⋆ $C_{opt}(T)$ is the estimated software cost;
- ⋆ P_{fix_tes} is the estimated cost of finding and fixing a fault during testing ;
- ⋆ P_{fix_ope} is the estimated cost of finding and fixing a fault during operation, where $P_{fix\_ope} > P_{fix\_tes}$;
- ⋆ P_{tes} is the estimated cost of testing per unit time;
- ⋆ P_{fix_tes}, P_{fix_ope} and P_{tes} are assumed to be known in advance;
- ⋆ t is the software life-cycle length;
- ⋆ T, $m(t)$, a and b are as defined previously.

The expression for $C_{opt}(T)$ is then

$$C_{opt}(T) = \underbrace{P_{fix\_tes}m(T)}_{1} + \underbrace{P_{fix\_ope}[m(t) - m(T)]}_{2} + \underbrace{P_{tes}T}_{3} \tag{2.5}$$

Where

1 → is the cost of finding and fixing faults during the testing phase;

2 → is the cost of finding and fixing faults during the operational phase;

3 → is the testing cost.

The objective is to find the best value of T that minimizes $C_{opt}(T)$. This is found in this model by differentiating $C_{opt}(T)$ with respect to T and equating the result with zero. This produces

$$T = \frac{1}{b} \ln \left[\frac{ab(P_{fix\_ope} - P_{fix\_tes})}{P_{tes}} \right]$$

As we must have $T > 0$, it can be noted that a solution only exists when

$$ab > \frac{P_{tes}}{P_{fix\_ope} - P_{fix\_tes}}$$

- **Observations on the Goel-Okumoto model**

  ⋆ Using the reliability criterion the estimated testing time is not linked to any cost constraints. Therefore, if the cost that would correspond to testing time T is required equation (2.5) should be used.

  ⋆ Conversely, the formulation using the cost criterion does not take any figure for reliability into consideration. The option that remains is to determine the value for R which would be achievable with testing time T (that would produce the cost $C_{opt}(T)$), using equation (2.4).

  Therefore, a straightforward trade-off model between cost and reliablity is not obtained using the Goel-Okumoto model.

2.3.3 Stopping rule considering cost and reliability

In this case the decision policies on the optimum software release times consider both software cost and software reliability criteria simultaneously. Presented here is a method proposed in [80], for an exponential SRGM:

$$R_{est}(x|t) = exp\left[-\sum_{i=1}^{2}\left(m_i(x)e^{b_i t}\right)\right]$$

where

⋆ $R_{est}(x|t)$, the estimated software reliability, is defined as the probability that a software fault does not occur in the time interval $(t, t + x)$, given that the last fault occurrence time is $t \geq 0$, $(x \geq 0)$;

- ⋆ b_i is the fault detection rate at time t;
- ⋆ $m_i(t)$ represents the expected number of faults of type i to be found and fixed during time interval $(0, t)$ (similarly to section 2.3.2);
- ⋆ $i = 1, 2$ means types i of faults–it is assumed that there are just two types of fault: type 1 faults which are "easily" found and fixed, whereas type 2 are "difficult" to find and fix (a clear explanation of how to define an "easy" or "difficult" fault is not provided by [80]);

The software cost $C_{opt}(T)$ is given by the same expression as in equation (2.5). However, in this approach we seek to determine the optimum software release time which minimizes $C_{opt}(T)$ subject to the condition that $R_{est}(x|T)$ is not less than the required reliability R.

The optimal software release problem is formulated as follows.

For specified operational time $x \geq 0$, minimize $C_{opt}(T)$, subject to $R_{est}(x|T) \geq R$ and $T \geq 0$. As shown in [80], the optimum software release time T is then obtained as a result of the minimization procedure.

- **Observations on cost and reliability combined**
  - ⋆ This model employs the same concepts as the Goel-Okumoto model but in this model both cost and reliability are considered simulteanously;
  - ⋆ The formulation in this model leads to an analysis of the trade-off between cost and reliablity during the testing phase. However, some period of time during the testing phase must have passed, so that we can estimate the parameters required in the model (defined in section 2.3.2), before any trade-off can be studied.

⋆ As this model enables us to analyse the required trade-off, one very tempting thought is to explore whether an adaptation of this model or its concepts may be accomplished, so that we can apply it during the design phase. It is clear that we cannot use this model directly in the design phase since some parameters in the formulas are obtained during testing. A combination of the concepts of this model and those of the software estimation models could turn out to be a very reasonable approach for the trade-off of cost against reliablity during the design phase.

2.3.4 Release policies with modular structure

As is emphasized in [45], the influence of modular structure on the software release time has been largely ignored in previous research. The following model, which is proposed in [45], is a policy for determining the release time of software systems composed of modules, during the testing phase, taking into account the amount of use of the modules during their execution.

Main problem: after a period of time T, should the software system be released or should testing be continued?

The concept behind this model is to work with the probable profit that earlier release may produce. So, to answer this question, a function $p(\tau)$ is proposed in [45] describing the profit obtained by releasing the software system (which is composed of modules) after further testing of duration τ. A very brief summary follows.

$$p(\tau) = V_1(T+\tau) - V_2(\tau) - V_3(T+\tau)$$

where

⋆ $V_1(t)$ is the value of the software system at time t, based on the cost of each module;

- ⋆ $V_2(t)$ is the average system cost due to undetected software faults and depends on the period of time in which a module is executed; the period of time in which the software system is in use during the test period; the length T of the testing period; and the number of faults found and fixed in each module.

- ⋆ $V_3(t)$ is the cumulative running cost of software testing up to time t (considering all modules involved) when the release time is t.

The value $p(0)$ and $p(t)$ for some $t > 0$ are then compared. If $p(0) \geq p(t)$, the test is stopped and the software system is released at time T. Otherwise the test is continued until time $T + t$, at which time, by replacing T by $T + t$, this comparison procedure is repeated. This procedure is continued until the software release time is determined.

In [45], some statistical procedures are developed for estimating the number of software faults for individual modules, and an algorithmic procedure is described for determining the values $V_1(t)$, $V_2(t)$ and $V_3(t)$; the determination of the software release time is then discussed in detail.

- **Observations on policies for modular structure**

  - ⋆ This model does not contain a software release policy considering cost and reliability simultaneously. However, the formulation proposed considers a modular software system, which does not happen in the previous models described.

  - ⋆ The equations for $V_1(t)$, $V_2(t)$ and $V_3(t)$, as shown in detail in [45], are rather complex for a project manager to handle. As noted in [36], project managers often refuse to use any model that contains other than "simple arithmetic formulae".

  - ⋆ In spite of above issues, this model appears to be closer to the practical aspects involving the "real life" of software development and more comprehensive than the other two.

2.4 Conclusion

It may be concluded that the current software cost estimation models available in the literature, either by themselves or through sizing software models, do not have any special approach to deal with the trade-off between cost and reliablity before the coding phase.

As outlined, a real discussion on the trade-off between cost and reliability can be established during the testing phase through software release policies. However, the input data for these policies are obtained only during the testing phase which may represent an impediment for their utilization in the earlier phase of the life-cycle of software development, such as at the design phase.

Chapter 3

Trade-off between Cost and Reliability

3.1 Cost of Development: Structure

As stated in chapter 1, we are interested in estimating the development cost of a modular software system during the design phase, taking into account a required level of reliability for each module. In this section a method of estimating this development cost is proposed, based on some factors that are related to the cost and reliability of a module. The acquisition of some of the factors employed in the method were considered in chapter 2.

It has to be stressed that we are not analysing the entire cost of developing a module[1] but, more specifically, the cost spent during the coding and testing phases.

To accomplish this task we use the most common technique for costing any engineering development project [55], that is, to employ effort estimation. Firstly,

[1]If this were the case, we should also need to consider the cost spent during the design and previous phases, and the cost of putting the software system into operation.

the number of person-periods (the effort) needed to perform coding and testing (including debugging) is estimated and then a cost is associated with each unit of effort so that an estimated cost is obtained.

A project manager knowing the outcome of this estimation, namely, effort and cost, and considering a required reliability, could then plan the resource allocation for the coding and testing phases, aiming at avoiding the known problem of cost overrun in these phases.

The estimated cost of development of a software module is defined here as the cost to implement all functions identified in the requirement specification taking into account a required level of reliability R, so that the module can be considered ready for operation. This cost quantifies the effort spent during the coding and testing phases and is represented by

$$C_{dev} = P_{dev} \cdot E_{dev}$$

$$C_{dev} = P_{dev}(E_{cod} + E_{tes}) \tag{3.1}$$

where

- C_{dev} is the estimated cost of development of a module (taken to be the cost of coding plus testing, taking into account a reliability level R), based on the effort of development E_{dev}. The total development cost of a system will be the sum of each individual module cost for all of the modules in the system.

- P_{dev} is the cost of development per person-unit time. In this thesis, it is assumed that the cost per person-unit time spent in either the coding or testing phases is indistinguishable. Hence, the single cost P_{dev} is used here to quantify the cost of both the coding and the testing phases. However, if they are to be regarded as different, this simply means that P_{cod} and P_{tes} would have to be estimated separately.

The value of P_{dev} will be a characteristic of each user's installation, and is related to the skill of the programming team allocated to those phases. It is assumed here that P_{dev} includes the direct costs of human resources (salary, tax, etc.), as well as other costs such as use of computational resources, support staff (management, secretary, etc.) and overheads (heat, light, rent, etc.). The value P_{dev} must be available for each skilled person that can be allocated during those phases. The unit of P_{dev} is pounds per unit time.

During the design phase the project manager must be able to indicate which profile of human resource he is going to employ to develop each module. Based on this information the cost P_{dev} is then assigned[2].

- E_{cod} is the effort required to implement (i.e., code) the module in, for instance, man-months. It is estimated based upon an analysis of the cost of previous projects within the organization and using data from "similar developments"[3] so that a correlation can be substantiated between the effort expended on those projects, taking into account various module sizes and functions.

  When we need to recover information from previous projects (a case in point is module size versus the effort that has been expended in their implementation)

[2]In the case that team members have distinct costs $(P_{dev})_i$ then considering m to be the number of team members

$$C_{dev} = \sum_{i=1}^{m} (P_{dev})_i (E_{dev})_i$$

[3]As suggested in [13], "the user defines what this means". A comprehensive approach on how to define similar software can be seen, for example, in [55, 64].

we need to ensure that we are employing data from previous software which resembles, as far as possible, the software that is currently being estimated. To establish that resemblance we have to define some sort of classification that allows us to homogenize the data.

However, there does not seem to be an obvious definition for similar development, which is suitable for any software project and that allows us to establish a precise classification of a software application. An initial attempt for such a classification is to classify the software to be developed by the *type* of application for the software. A suggestion of potential macro-categories to enable a crude classification, as proposed in [55], would be: system software, real-time software, business software, engineering and scientific software, embedded software, personal computer software and artificial intelligence software.

In addition to this macro-classification, it is suggested that the user, namely, the project manager, should split each category into sub-categories as necessary, to adapt further the classification within his environment.

- E_{tes} is the effort spent in verification and validation of a module during the testing phase. A comprehensive definition for the terms verification and validation[4] can be obtained from [55]:

    i. *Verification* is defined as being "the set of activities that ensure that the software correctly implements a specific function". Boehm in [7] says that verification aims at answering the following question: "Are we building

[4] *Verification and Validation* (*v&v*) can have a wider meaning. As defined in [84], "*V&V* is a collection of analyses and testing across the *full life cycle* and complements the efforts of other quality engineering functions". Here, however, we are using the term *v&v* just for activities performed during the testing phase.

the product right?";

ii. *Validation* "refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements". The question that validation tries to answer, according to Boehm, is "Are we building the right product?".

Even though the words testing and debugging are often casually used with the same (or similar) meaning[5], they are, in fact, distinct activities, as is emphasized in [55, 68]. Testing is the activity of finding situations in which the results do not match those expected, that is, a failure of the software has occurred, while debugging is the activity of diagnosing and correcting the fault (bug) that produced the failure, as depicted in figure 3.1[6].

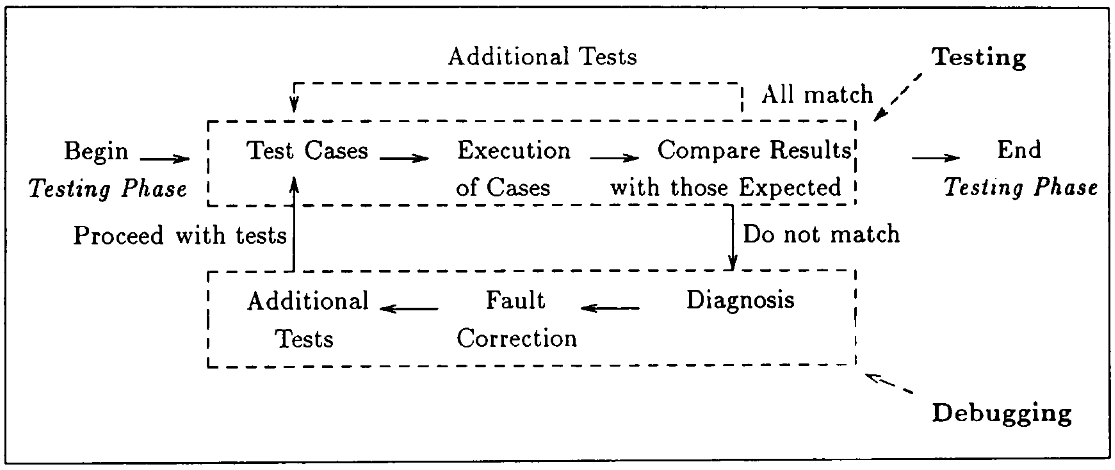

Figure 3.1: Activities during the testing and debugging phase

However, E_{tes} is used here to represent the sum of the effort of testing (considered separate from debugging activities) and the effort of debugging. Thus, the testing

[5]Software release policies [44, 45, 53, 80, 83] deal with testing and debugging as a single activity.

[6]In figure 3.1, the test cases and additional tests are prepared by the developer.

and debugging activities are dealt with as a single task. On page 61 some arguments are presented which may clarify why we adopt this approach.

Subsequently the components of equation (3.1) are expanded, where, it should be highlighted once more, we are interested in finding a way of expressing C_{dev} in relation to a required level of reliability R for an individual software module.

3.2 Underlying Relationship for the Required Reliability

Three quantities of software faults are utilized in our analysis of cost and reliability, as indicated in figure 3.2, where

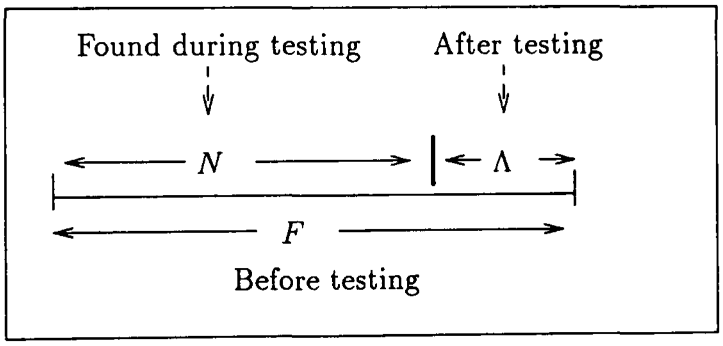

Figure 3.2: Quantities of faults

- F is the estimated number of faults that will be introduced into the module during coding;
- N is the estimated number of faults that will be found and fixed (correctly) during the testing phase;

- Λ is the estimated number of faults that will remain in the module after finishing the testing phase. It is assumed that $\Lambda = F - N$ (perfect debugging).

3.2.1 Relationship between N and required reliability

It is generally agreed that the reliability R of a module is conceivably dependent on the number of faults Λ that remain in the module after testing and debugging have finished; as Λ decreases the probability that the module works according to its specification will increase, that is, R will vary inversely with Λ.

For a given level of reliability R we need to estimate the testing plus debugging effort needed to achieve R. This effort is determined by N, the number of faults needed to be fixed to reduce the faults from F to Λ, where Λ is sufficiently low that the reliability is R.

Thus we need to estimate N, based on a relationship between N and R.

To make this association the following arguments are considered:

1) Suppose a module with S lines of code has Λ remaining faults. Assuming that each line of code can hold just one fault, we have Λ faulty lines among the S lines.

2) Let β be the probability of an individual faulty line being executed and causing a failure during an invocation of the module, where we assume that β is the same for each faulty line. β then represents the probability that for one run of the module a specific faulty line will produce a failure. So, for example, $\beta = 0.005$ (considering just one faulty line) means that for 1000 executions of the module (using different input data), on average five failures will be caused by this particular faulty line. This parameter is briefly analysed below.

   This same assumption is adopted, for instance, in the early software reliability models of Jelinski-Moranda, Shooman and Musa [6, 76]. Results obtained with this assumption have been claimed to yield, in many situations, an overly optimistic estimate for the behaviour of faults in a software module, as analysed, for example, in [42]. Many later models have been proposed to overcome this deficiency. However, it should be stressed that these models rely on data collected during the testing phase. This data is not available to be used in the model developed here.

In spite of the clear limitations of the above assumption, the results obtained from the whole model developed in this work may still be sufficiently valid, in this early phase of the life-cycle of a software module.

3) We also assume that the manifestation of a fault does not depend on the occurrence of other faults (i.e., the remaining faults occur independently).

In general, using $P(B_i)$ to denote the probability that fault i does not manifest itself for one run of the program, then the probability that none of the faults $1, \cdots, \Lambda$ occur during one run of the program is equal to

$$P(B_1) \cdot P(B_2|B_1) \cdot P(B_3|B_1B_2) \cdots P(B_\Lambda|B_1B_2 \cdots B_{\Lambda-1})$$

and it would then be necessary to estimate the conditional probabilities $P(B_i|B_1B_2 \cdots B_{i-1})$ (probability that ith fault does not occur given that 1st, 2nd, $\cdots$, $i-1$th faults do not occur).

With the simplifying assumption of independence, we have $P(B_i|B_1B_2 \cdots B_{i-1}) = P(B_i)$, and, therefore, $P(B_1) \cdot P(B_2|B_1) \cdot P(B_3|B_1B_2) \cdots P(B_\Lambda|B_1B_2 \cdots B_{\Lambda-1})$ reduces to $P(B_1) \cdot P(B_2) \cdots P(B_\Lambda)$.

Of course, in practice there often is a knock-on effect; the occurrence of one fault may give rise to the occurrence of another fault. However, we cannot build this possibility (the conditional probabilities) into our analysis during the design phase, because it is completely unknown at the design phase, how a fault (which has not yet been created) might cause other faults to occur.

Then, considering the reliability of a software module to be the probability that when called the module will operate according to its specification and will transfer control correctly when finished (as will happen if none of the Λ faults contained in the module occurs), we then have that the reliability R of the module is given by

$$R = (1 - \beta)^{\Lambda} \tag{3.2}$$

So,

$$\ln R = \Lambda \ln(1 - \beta)$$

As $\Lambda = F - N$, then

$$N = F - \frac{\ln R}{\ln(1 - \beta)} \tag{3.3}$$

Observe that since the right-hand side of equation (3.3) must be non-negative, a very small value for β will imply a very large value for R.

As defined previously, the parameter F represents the estimated number of faults that will be present in a module after coding. As analysed in [38, 39, 75], there are some factors, related to characteristics of development, that have a direct effect on F; these factors can be called fault drivers, for example, difficulty of programming, program-team's skill (programming experience of each member of the programming-team) and module size. One method, described in [75], is to estimate F using regression analysis between the factors cited above and the expected number of faults. Another approach, discussed in section 3.2.3, is to use an adjustable formula, as proposed in [71], which is related to the module size and some parameters that may characterize its complexity.

3.2.2 Considerations on β

We do not derive an explicit expression for β in this work. Rather we assume that β is estimated during the design phase based on the historical data of past projects of the same category as the module under development.

We may estimate β from previous project data if we know Λ and the failure rate per run for the modules of earlier projects. For a specific module, let η represent the failure rate/run, so that

$$\beta = \frac{\eta}{\Lambda}$$

Example: Suppose a module contains five faults. Suppose that in 1000 runs of the module 10 failures occur, that is, $\eta \approx \frac{10}{1000} = 0.01$ failures per run. Then $\beta = \frac{0.01}{5} = 0.002$. This means that if we have "similar" (see page 45) software we might estimate the "failure rate per fault" (the probability that a remaining fault will produce a failure) as being $\beta = 0.002$.

Further work should address the issue of assessing which, if any, factors have a strong influence on β, so that an expression for β can be established. It might be suggested, for example, that factors such as programming language, software size and technical complexity might exert a direct influence on β.

Based on these factors and using regression analysis (see an example of regression analysis in section 2.2.1) an expression for β as a function of these parameters may be defined.

3.2.3 How to estimate the expected number of faults

The parameter F represents the number of faults initially present (that is, introduced) in a module. This value can be estimated during the design phase, if we take account of the evidence [19, 38, 54] that there is a close relationship between the module size and number of faults.

a) Using software science

Research on this topic [19, 38, 39, 71] enables an estimate to be made of the fault density, that is, the relationship between the number of faults and the number of lines of code based on factors that are available during the design phase. As the number of lines of code can be estimated, it follows that the number of faults can be estimated.

Using the formulation proposed in [38] and improved in [71], which appears to produce results which are close to the actual number of faults, we have the following expression for F

$$F = \frac{S \cdot \kappa}{3000} \cdot \log_2 \left(\frac{8 \cdot \kappa \cdot S}{1 + 8 \cdot \log_2(\kappa \cdot S) - 9 \cdot \log_2(\log_2(\kappa \cdot S))} \right) \qquad (3.4)$$

where

- S is the number of lines of code, which should be estimated as discussed in chapter 2;
- κ is a constant expressing the average number of operators and operands used per line of code[7]. This constant can be acquired from a table pro-

[7]These concepts were proposed by Halstead [12] and they have the following meaning: an operator is any symbol or keyword that specifies an action, for example, add, multiply and move; an operand is any symbol used to represent data, including variables, constants and labels.

posed by Halstead that associates κ with the programming language utilized for implementing the module, or κ can be evaluated locally for similar developments using the same programming language[8].

For example: an algorithm implemented in assembler language may have $\kappa = 2.67$, whereas if it were developed in Fortran, we would expect $\kappa = 7.5$ [54].

At first glance it would seem that line for line a Fortran program has more faults than an assembler program, an observation that is hard to believe. However, it has been said that in assembler language [38] "four times as many lines of code are needed to implement a given algorithm as compared to Fortran". So, for the same algorithm, the assembler program could be four times bigger in size, meaning that it would have more faults than a similar program implemented in Fortran, as expected.

The outcome of applying equation (3.4) is reasonably precise, as is demonstrated in [38, 71].

Example:

Number of faults F expected for a module with the following characteristics:

$S = 4000$ lines of code; the programming language is Fortran, so in this case the parameter κ is 7.5, as stated in [54].

Evaluating equation (3.4) in this case we obtain $F = 115$ faults. That is, we can expect 115 faults to be introduced during the coding of that module, which implies that almost 3% of the lines of code will be defective.

b) Using a direct, local method

[8]It is worth stressing that there exist some criticism about Halstead's theory, as can be seen in [66].

Another way to estimate F, as proposed in [75], is through analysis of the relationship between faults in a program and the factors that may have a direct effect on the number of faults to be introduced, instead of just size and programming language.

However, this method requires some data, such as frequency of program specification change and volume of program design document, that will only be available after finishing the design phase. Therefore, it would not serve our purpose to produce the estimate during the design phase. For this reason, this alternative is not considered here.

3.3 Effort of Testing

It is argued in this work that the magnitude of the effort of testing can be related to the required level of reliability R. We have to find and fix N faults in order to achieve the level of reliability R required and we can estimate N from R using equation (3.3), if we have estimates for F and β. We can estimate F using equation (3.4), by estimating S and establishing a value for κ; β must be estimated from prior data.

Next, we need to establish a suitable expression for the effort of testing (finding and fixing N faults during the testing phase).

The overall effort E_{tes} to be spent during the testing phase, so that the remaining faults Λ correspond to the desired reliability R, is thus hypothesized to depend directly on two factors: the number of faults N to be removed (we will use "removed" to mean "found and fixed" in this work) and the effort τ_j to remove the jth fault during testing. On this basis, we express the effort of testing as being the sum of the effort of removing the first, second, ... , Nth faults. Therefore,

$$E_{tes} = \tau_1 + \tau_2 + \cdots + \tau_N \tag{3.5}$$

3.3.1 Estimated effort to remove one fault

The underlying assumption that enables us to estimate τ_j, as analysed in [54], is that the expected average amount of effort required to remove the jth fault is proportional to the total effort required to implement the module divided by the expected number of faults. This assumption apparently gives a very reasonable approximation for the sought effort, according to examples shown in [54], and is based on the hypothesis that "if there are F bugs expected in a software module, one would have to understand to some degree $\frac{1}{F}$ of the program on average for each bug found".

This assumption is represented here by

$$\tau_j = \alpha_j \cdot \frac{E_{cod}}{F} \tag{3.6}$$

where:

- E_{cod} is the effort required to implement (coding) the module, which can be estimated as analysed in section 2.2.3;
- F is the expected number of faults, as analysed in section 3.2.3;
- α_j is a factor of proportionality that is employed here to characterize the effort required to remove the jth fault.

Since no detailed data are available about α_j, we make an assumption based partly on intuition and partly on mathematical convenience which allows us to establish an expression for this parameter:

- The effort to remove a fault is assumed to increase during the testing phase, that is, the effort of removing the jth fault is greater than the effort of removing the $(j-1)$th fault. So, $\alpha_1 < \alpha_2 < \cdots < \alpha_N$.

  At the beginning of testing, for an allocated level of effort (for example, man-months), a certain quantity of faults are removed leading to a sharp decrease in the number of remaining faults. Later in the testing, for the same allocated effort, many less faults will be removed, because they are more "hidden".

  As analysed in [68], once the curve shown in figure 3.3 begins to approach a vertical asymptote, this means that testing is either nearing completion or has become transfixed, without achieving any further significant improvements. It does not mean that all faults have definitely been removed, but that the current test method has almost achieved the maximum number of corrections possible with that method. Following our line of reasoning, each fault will require a different increasing effort to be removed.

  Thus, it can be said that removing faults during the final stages of testing requires a great deal more effort than at the beginning in a behaviour that is clearly not linear, as envisaged in figure 3.3.

  The graph of figure 3.3 shows τ_j increasing exponentially with j during the testing phase.

  On the basis of the above points, it is proposed that the parameter α_j should vary exponentially in relation to j, taking on the behaviour roughly depicted in figure 3.3.

Now we have to find a manner of expressing α_j so that the stated conditions are fulfilled. A suitable expression which fulfils all the required properties is:

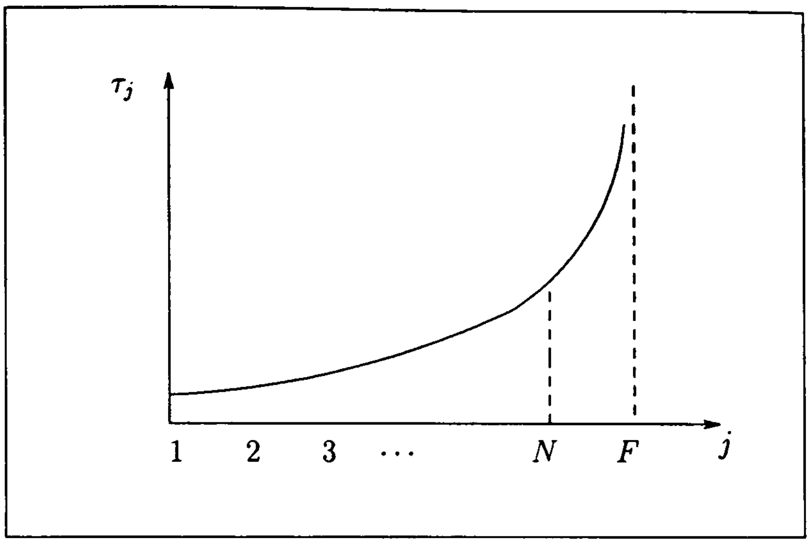

Figure 3.3: Each fault requires a different effort to fix

$$\alpha_j = \rho \cdot e^{j \cdot s} \tag{3.7}$$

where s and ρ are constant parameters that control, respectively, the steepness and amplitude of the curve showed in figure 3.3.

Hence,

$$\tau_j = \rho \cdot \frac{E_{cod}}{F} e^{j \cdot s} \tag{3.8}$$

It should be noted that the assumed exponential behaviour of τ_j during testing roughly agrees with the data collected in [68]. A likely explanation for this behaviour, in the real environment of testing, is that in the beginning of testing with a small amount of effort (person-period) one fault can be removed. After that, the amount of effort needs to be increased in order to remove one fault, because, as mentioned previously, the faults will be more obscure. Therefore, in the final phase of testing, to analyse the circumstances that a fault occurred, understand the failure and determine a proper correction will sharply require more manpower than in the beginning of the testing, which can explain that exponential behaviour.

To establish a final expression for τ_j, expressions for ρ and s need to be derived.

- **Expression for s**

An expression for s (the rate at which τ_j increases) is now derived using the following line of argument.

From equation (3.8) we see that the effort of fixing the Nth fault is given by

$$\tau_N = \rho \frac{E_{cod}}{F} e^{N \cdot s}$$

And the effort of fixing the first fault is given by

$$\tau_1 = \rho \frac{E_{cod}}{F} e^{s}$$

Then,

$$\frac{\tau_N}{\tau_1} = \delta = \frac{e^{N \cdot s}}{e^{s}} = e^{(N-1) \cdot s}$$

$$s = \frac{\ln \delta}{N-1} \tag{3.9}$$

where it is assumed that there exists a parameter $\delta = \frac{\tau_N}{\tau_1}$ (which can be used for the software under estimation) which reasonably represents the ratio between the effort of finding and fixing the Nth and first faults[9]. To simplify the representation of δ, it is not denoted here as being δ_N.

Further work should address this issue, in order to establish an expression for δ based on data that are available during the design phase.

[9]In [7, page 40] it is suggested that $1 < \delta < 10$, for smaller software projects.

- **Expression for ρ**

To obtain an expression for ρ, we introduce a factor that enables the estimator (for instance, the project manager) to set an upper bound for the estimated effort that will be expended during the testing phase. So, we can say that

$$\underbrace{\sum_{j=1}^{F} \tau_j}_{\star} = \underbrace{\gamma \cdot E_{cod}}_{\star\star} \tag{3.10}$$

where the factor marked with ($\star$) represents the effort required to remove all F faults and the factor marked with ($\star\star$) represents the maximum effort of testing that is envisaged by the estimator. The latter is expressed in terms of the effort of coding E_{cod}, where γ characterizes the relationship between the effort of coding and the maximum effort of testing[10]. In the same way as observed for the parameter δ, it is assumed that the parameter γ is available during the design phase to be employed for the software under estimation. Then, substituting equation (3.8) in equation (3.10), we have

$$\rho \frac{E_{cod}}{F} \underbrace{\sum_{j=1}^{F} e^{j \cdot s}}_{\dagger} = \gamma E_{cod}$$

where the term marked with ($\dagger$) is the sum of a geometric progression with rate e^s. Then,

$$\rho \frac{E_{cod}}{F} \left(e^s \frac{e^{Fs} - 1}{e^s - 1} \right) = \gamma E_{cod}$$

$$\rho = \gamma F \left(\frac{e^s - 1}{e^{Fs} - 1} \right) \frac{1}{e^s} \tag{3.11}$$

[10]Some of the evidence shows that $0.4 < \gamma < 2.5$, as can be seen, for instance, in [31, 55]. Therefore, according to these figures, $0.4E_{cod} < (E_{tes})_{max} < 2.5E_{cod}$.

3.3.2 Expression for E_{tes}

Considering the expression for E_{tes} in equation (3.5), we have that

$$E_{tes} = \sum_{j=1}^{N} \tau_j$$

$$E_{tes} = \sum_{j=1}^{N} \gamma F \left(\frac{e^s - 1}{e^{Fs} - 1} \right) \frac{1}{e^s} \frac{E_{cod}}{F} e^{j \cdot s}$$

$$E_{tes} = \gamma E_{cod} \left(\frac{e^s - 1}{e^{Fs} - 1} \right) \frac{1}{e^s} \underbrace{\sum_{j=1}^{N} e^{j \cdot s}}_{\ddagger}$$

where the factor marked with (‡) is again the sum of a geometric progression with rate e^s. Then,

$$E_{tes} = \gamma E_{cod} \left(\frac{e^s - 1}{e^{Fs} - 1} \right) \left(\frac{e^{Ns} - 1}{e^s - 1} \right)$$

$$E_{tes} = \gamma E_{cod} \frac{e^{Ns} - 1}{e^{Fs} - 1} \tag{3.12}$$

One may argue why not link cost and reliability considering testing and debugging separately. An argument follows.

The effort expended during the testing phase is composed of two factors:

i) The effort required to check that the software is working according to its specifications. This effort may be seen as being:

   - mandatory effort required to attend to the demands of *v&v* during the testing phase, without taking into account any figure for the required reliability as an element of decision;

– supplementary effort of verification and validation to find sufficient software failures and faults to achieve the desired reliability.

ii) The effort of relating each software failure to a software fault, which will result in fixing the fault.

The former we can associate with the effort of testing in itself, whereas the latter can be associated with the effort of debugging.

If testing and debugging efforts were considered separately, it can be concluded that new parameters should be introduced, and therefore estimated, in order to establish separately the efforts of finding and fixing a fault. If it were the case, it can be said that all procedures developed here to establish E_{tes} would be roughly the same; however, where we call "effort of removing" should then be separated in two different efforts, perhaps "effort of detecting" and "effort of debugging". Yet, effort of detecting plus effort of debugging must be equal to E_{tes}!

Thus, there does not seem to be any significant advantage in using this approach. This approach would instead produce complicated formulas with more difficult parameters to be estimated, and without a clear advantage in the final result.

3.4 Estimated Cost of Development

Substituting the value for E_{tes} obtained in equation (3.12) in equation (3.1) we now obtain the final expression for the cost of development.

$$C_{dev} = P_{dev}E_{cod}\left(1 + \gamma\frac{e^{Ns}-1}{e^{Fs}-1}\right) \tag{3.13}$$

where $N = F - \frac{\ln R}{\ln(1-\beta)}$ (equation (3.3)).

Considerations on equations (3.3) and (3.13):

- Values known in advance:

  R and P_{dev}

  The cost P_{dev} must be available, based on the type of human resources to be allocated in the coding and testing phases.

  The level of required reliability R for each module is known in advance and is linked with the overall reliability of the software system, as described in [9] and analysed in the next chapter.

- Values estimated using formulae developed in this work

  N (equation 3.3) and s (equation 3.9).

- Values estimated using data from the design specification and past projects

  E_{cod}, F, β, γ, δ and S.

  In section 3.2.3 the estimation of the parameter F was briefly outlined, using equation (3.4) to relate it to size S.

  There must be historical data of previous projects, using similar characteristics of development in the installation under consideration, that allow us to estimate the parameters above.

- The relationship between the development cost C_{dev} and the required level of reliability R is roughly depicted in figure 3.4 (in chapter 5 there is a more precise graph and an explanation for why there is a "flat" section in this curve). As we might expect, the cost rises markedly when the required reliability approaches 100%. Taking into account the assumptions that have been made already, we can estimate in advance of the coding and the testing phases how much that cost will be for a required level of reliability.

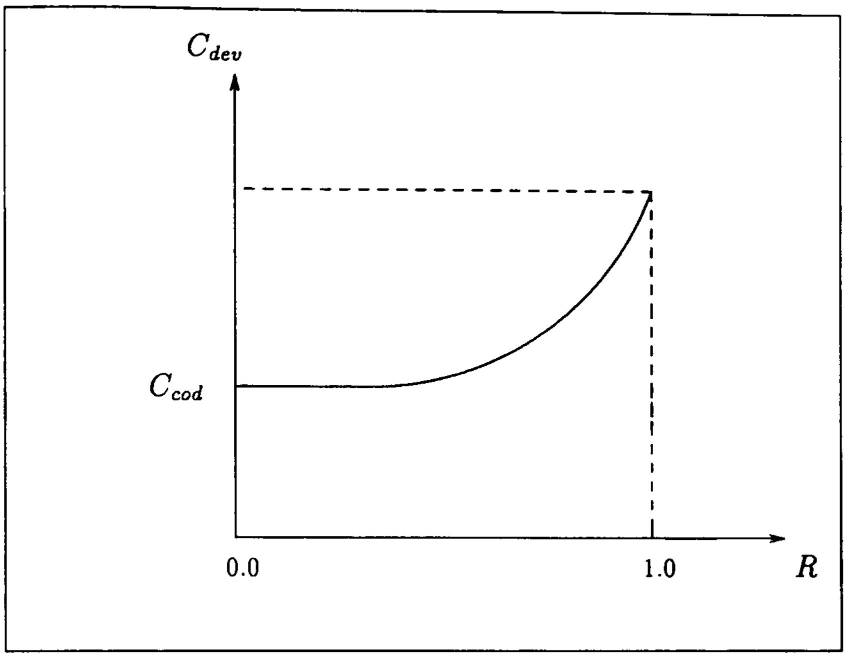

Figure 3.4: Cost of development versus reliability

- One might question in the established formula, why there is a finite cost C_{dev} to achieve reliability $R = 1.0$, (that is, 100% reliability), when a common feeling says that this value might be infinite.

  The reason is that, in this work reliability is linked to the number of faults that remain in the module, which is assumed to be finite. So one can estimate this number of faults and the effort required (man-months) to fix all these faults.

  The crucial matter, which is not addressed in this work at all, is how to apply software engineering practices, if any, such that the estimated effort E_{tes} be effectively able to find and fix all of the N sought faults, and certify that this has been achieved. If this matter is not fully dealt with during the testing phase, the cost may indeed turn out to be infinite!

Chapter 4

Overall System Reliability

4.1 Basic Concepts

To avoid the problems of complexity which the design of a single monolithic software system creates, it is usual to divide software into separate components called modules, which are subsequently integrated to satisfy problem requirements [55]. Modular software systems consist of a set of modules which carry out a range of different tasks. Among these modules there exists a pre-defined structure of "who-calls-who", which is known from a detailed requirement specification. This work considers a particular form of module interaction, where after a module has completed its execution, the control of the system is passed to another module, on either a deterministic or stochastic basis (as exemplified in figure 4.1).

4.1.1 Stochastic process

The execution of a system with modular structure can be regarded as a stochastic process, because its processing occurs by stages (corresponding to the modules), over a period of time; processing follows a sequence of stages $S_0, S_1, \ldots, S_n$.

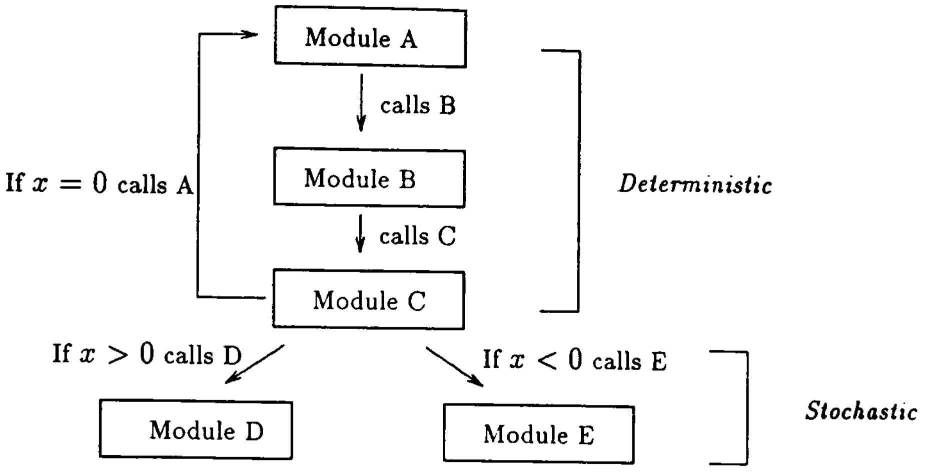

Figure 4.1: Processing in a modular structure

where one can predict the likehood of this behaviour, since the combined probability $P(S_0, S_1, \ldots, S_n)$ for any specific sequence is known.

As is well-known, the combined probability for a generic sequence can be expressed by:

$$P(S_0, S_1, \ldots, S_n) = P(S_0)P(S_1|S_0)P(S_2|S_0, S_1)\ldots P(S_n|S_0, \ldots, S_{n-1})$$

That is, the behaviour of any stage is conditioned by the outcome of all previous stages.

4.1.2 Markov chain

A *Markov Chain* is a stochastic process where the transitions between states do not depend on past history, nor on the current time, but only on the current state. So the probability of a given module being invoked, in a system with modular structure, is a function of the module currently being executed and the given module only. Then, the probability of a sequence $S_0, S_1, \cdots, S_n$ in a Markov Chain can be expressed as:

$$P(S_0, S_1, \ldots, S_n) = P(S_0)P(S_1|S_0)P(S_2|S_1)\ldots P(S_n|S_{n-1})$$

Any modular structure that follows this particular form of interconnection among the modules (which is known as a *Markov property*) is called a ***first order Markov chain.*** The modular systems considered in this work are assumed to have this Markov property.

It should be emphasized that the assumption of a Markov process is a good representation of the actual control exchange process in many applications, and is frequently used in software engineering practice [10], with the states of the Markov process representing software modules. An example can be seen in [35], where it is assumed that the transitions between modules follow a Markov property.

A Markov chain can be regarded as a 3-tuple (S, P, S_0), where S is a finite set of states, S_0 is the initial state ($S_0 \in S$) and P is a probabilities transition relation that indicates the probability of moving from one state to another. P is usually expressed as a matrix, where the states are the indices and the transition probabilities are the elements of this matrix. Thus, the probability that the next transition from state i will be to state j will be the matrix entry in the ith row and the jth column, denoted here by P_{ij}. This matrix is called the *transition matrix* of the Markov chain. In section 4.2.1, for the context of this work, this matrix and its elements are defined.

Using the transition matrix P we can determine the probability of transition from state i to state j in a sequence of n steps, by taking powers of the transition matrix. For example: for $n = 2$, we have that the probability of transition from state i to state j in exactly two steps as being the element P_{ij} of P^2.

Another concept utlized in a Markov chain is that of *absorbing states.* Once an absorbing state is entered it is never vacated. Hence, if, for example, F is an absorbing state, then P_{Fj} is zero for all j.

4.2 Constructing the Transition Matrix

To apply Markov analysis to a modular system during the early stage of the design phase, we must firstly obtain the transition matrix for the Markov chain that expresses the behaviour of the system in terms of the probabilities of transition among states (as characterized in the previous section).

In the context of the work developed in this chapter, the first and main task in constructing the transition matrix is to obtain a behavioural view of the system based on its requirement specification. The outcome of this view is a hierarchical decomposition of the system into macro-processes, which can be associated with software modules, and the probabilities of transition among processes can be assigned.

These activities, which are desired to be performed during the early stage of the design phase of software development, are summarized in figure 4.2 (the meaning of each term used in figure 4.2 will be defined subsequently).

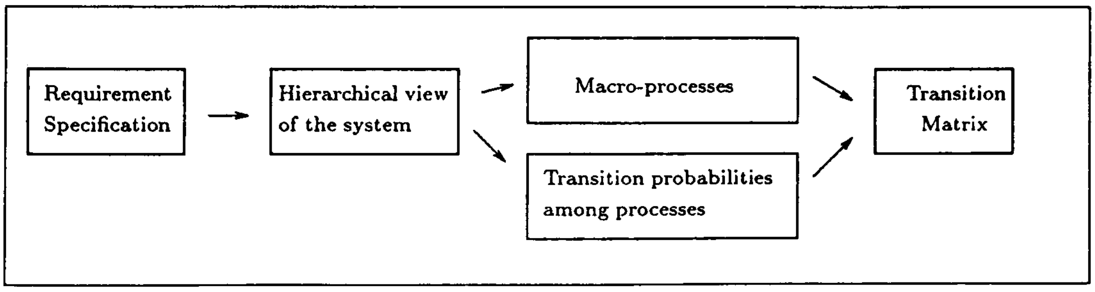
Requirement Specification
Hierarchical view of the system
Macro-processes
Transition probabilities among processes
Transition Matrix

Figure 4.2: Overview of the procedures to obtain the transition matrix

It can be observed that there exist many software engineering techniques, employing a varied degree of formality in their approaches, which can guide the project manager in obtaining a hierarchical view of the system from a requirement specification, as can be seen, for example, in [11, 15, 17, 22, 55, 68, 81]. Then, using what is noted in [86] as being "a creative design step and not an algorithm", the transition matrix can be constructed.

At this point, we must stress that in this work we do not attempt to assess available methods or propose new methods for constructing a transition matrix. Indeed, we simply assume that the transition matrix exists with the properties that are defined in detail in the next section. The section 4.2.2 provides more details of how this transition matrix may be constructed.

4.2.1 Transition matrix

When a software system has a modular structure it is apparent that the overall level of system reliability that will be experienced by the user depends on the sequence of modules to be executed and, naturally, on the reliability of each individual module [10, 40]. The reliability of any software system also depends, of course, on the profile of use, that is, the dynamic characteristics of a typical execution of the system in a particular user environment—a system operating in two distinct environments will exhibit different levels of reliability, depending on the utilization of the modules in each of the environments.

A transition matrix can be defined that expresses the pattern of interaction between the modules; this matrix can be used to represent (some aspects of) the behaviour of the modular structure. This matrix underlies the relationship between the overall system reliability and the reliability of each module.

The following definitions are employed in defining the transition matrix used throughout this work:

- **M_i** represents a generic module i and forms a "state S_i" of a system with n modules ($n = 4$ in the example to follow);

- **R_i** is the reliability of module M_i. There are two ways of dealing with software reliability [8]:

  ⋆ Failure rate over time

  The reliability is the probability that a module M_i operates according to specifications for a given period of time before a failure;

  ⋆ Failure rate per demand for service

  The reliability is the probability that module M_i will operate according to its specification when called and will transfer control correctly when finished.

  The latter approach is utilized in this thesis. As can be observed in the next chapter, we do not work with "time" in our formulation;

- **P_{ij}** is the probability that the transition between modules M_i and M_j will be taken, given that control is at module M_i and execution is completed according to its specification. The values P_{ij} have to be obtained from the requirement specification of the system ($0 \leq P_{ij} \leq 1$), as, for example, suggested in [86] and discussed in the next section;

- **R_iP_{ij}** thus represents the probability that the execution of module M_i completes according to its specification and control of the system is then transferred to module M_j;

- **M_1** is the start module, that is, S_1 is the initial state of the system;

- **F** is an absorbing (terminal) state that is reached when a module produces a result not conforming to its specification, that is, when a failure of that module occurs. This state is reached from module M_i with the probability $P_{iF} = (1 - R_i)$. This is a state of the model. The actual operating software does not necessarily reach a "failed" state that can be recognized as such.

- **T** is an absorbing (terminal) state which is reached when the system of software modules completes its overall task successfully. More precisely, a module M_i will make a transition to state T, with probability $R_i P_{iT}$, if the execution of M_i completes according to its specification and M_i should not then make a transition to any other module M_j. So $\sum_{j=1}^{n} P_{ij} + P_{iT} = 1$;

- $\mathbf{R_{req}}$ is the overall reliability of the system that the user needs to achieve. The value for this reliability is known in advance of the design stage;

- $\mathbf{R_{est}}$ is the reliability of the system obtained from the transition matrix using Markov analysis. It is the probability of reaching the terminal state **T** from the initial state $\mathbf{M_1}$. This represents the probability that the system completes its execution without failing.

As an example, the transition matrix shown in figure 4.3 describes a modular structure having four modules.

$$\begin{array}{c|cccccc|}
 & M_1 & M_2 & M_3 & M_4 & F & T \\
\hline
M_1 & 0 & R_1P_{12} & R_1P_{13} & R_1P_{14} & (1-R_1) & R_1P_{1T} \\
M_2 & R_2P_{21} & 0 & R_2P_{23} & R_2P_{24} & (1-R_2) & R_2P_{2T} \\
M_3 & R_3P_{31} & R_3P_{32} & 0 & R_3P_{34} & (1-R_3) & R_3P_{3T} \\
M_4 & R_4P_{41} & R_4P_{42} & R_4P_{43} & 0 & (1-R_4) & R_4P_{4T} \\
F & 0 & 0 & 0 & 0 & 1 & 0 \\
T & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

Figure 4.3: Example of a matrix using four modules

4.2.2 Procedures to obtain the transition matrix

To provide a very brief introduction for this topic, we summarize next how the transition matrix can be obtained. The elements in figure 4.2 are outlined, following the definitions given in [55], as well as the research developed in [47, 86], which, in our opinion, can satisfactorily serve as the starting point for constructing the transition matrix.

- *Software Requirement Specification*

  A document produced in the phase that precedes the design phase in the life-cycle of software development, which is a direct result of a software requirement analysis[1].

  Following the suggestion made in [55, page 199], among other information, the software requirement specification needs to hold: a *detailed functional description* (which should enable subsequent identification of the processes that will be implemented as software modules), a *behavioural description* (system states, events and actions) and *validation criteria* (containing characteristics of performance and constraints, such as an overall desired reliability for the system).

- *Hierarchical View of the System*

  A decomposition of the system functions, identified in the software requirement specifications, in a model which contains what are called in [47] "the process states"; these are, in fact, the computational functions to be performed, and also those data or events which produce control information, reports or displays, that influence how the system moves from one process to another.

[1]Software requirement analysis is an earlier phase in the development life-cycle, where the basic characteristics of the software to be developed are captured from the user.

As stated at the beginning of section 4.1, we focus our work on a particular type of modular system. We make the fundamental assumption that this hierarchical functional view must yield a modular structure, such that module interaction behaves as decribed in section 4.1, that is, either in a deterministic or stochastic basis, according to Markovian property.

- *Macro-Processes* (or *Process States*)

  One "process state" (or a group of them) should correspond to a likely software module which will be accurately defined later in the development life-cycle, during the design phase. Each such process state can then be regarded as a "state" in the Markov chain.

- *Transition Probabilities Among States*

  The last step to complete the Markov chain is to assign the probabilities of transition among states.

  In this task, we must observe that a module can have three possible behaviours (the exact meaning of the elements introduced below was discussed in the previous section):

  i. A module i passes control to a module j successfully, that is, according to its specification, with probability P_{ij};

  ii. A module i completes its task successfully and goes on to a terminal state T with probability P_{iT}, without passing control to another module;

  iii. A module i produces a result not in accordance with its specification, with probability P_{iF}, and "goes" to a terminal (absorbing) state F called "failure". The real problem with software is that a module i does not behave according to its specification but nevertheless may transfer control to some other module j. This situation would also mean "to go" to state F.

Then, the software developer has to be able to assign the values P_{ij} and P_{iT}, for all modules identified from the hiearchical view. This task is very likely to lead to a misinterpretation of the system behaviour, because there may be a high degree of subjectivity involved.

The way these probabilities are assigned can be classified in three different ways [86]:

i. The informed approach

   Used when information about the actual sequences in which the modules are called is known in advance. This may be the case when a prototype has been used during the procedure of producing the software requirement specification or when a prior version of the system is available.

ii. The intended approach

   Used when the probabilities are assigned based on some hypotheses as to how the modules will be called.

iii. The uninformed approach

   Used when no information is available about the system behaviour. In this case, the usual technique is to assess the probability for all transitions as being equal.

As discussed in [86], the informed approach is the best, followed by the intended approach. As a last resort the uninformed approach is used, which can produce "anomalous results in the light of knowledge or intuition".

4.2.3 Formalizing the problem

An iterative process can be used to refine estimates of $\underline{R} = R_1, \cdots, R_n$ so that the Markov analysis would yield a result $R_{est} \geq R_{req}$ with R_{est} as close to R_{req} as possible (since this will keep software development costs down). It is well known that to achieve a higher figure for system reliability than R_{req} would entail spending more time and cost during development; aiming at a minimum acceptable reliability R_{est} indicates that we seek to keep the development cost of the software to a minimum.

The values obtained for each of the R_i from this iterative process constitute the proposed reliability allocation for the modules M_i.

This problem may be stated in the following form.

Find a value for $\underline{R}$ such that R_{est} is minimized

subject to

(i) $0 < R_i < 1$, where $i = 1, \cdots, n$

(ii) $R_{est}(\underline{R}) - R_{req} \geq 0$

Any set of values $< R_1, \cdots, R_n >$ that satisfies the conditions above would be an acceptable set. The problem is to find an acceptable set which allows a compromise between the reliabilities R_i and other contraints of the system; a case in point would be cost. The result obtained here addresses this problem.

In section 4.3 a formula is derived for calculating the overall reliability R_{est} from the transition matrix and $\underline{R}$. The allocation of values R_i is treated in chapter 5.

4.3 Determination of the Reliability of a System

The transition matrix defined in section 4.2.1 describes a finite Markov process with two absorbing states T and F, and a set of n transient states $S_1, \ldots, S_n$. The matrix can be depicted in the following form:

$$P = \begin{array}{c|cc} & S & A \\ \hline S & Q & H \\ A & 0 & I \end{array}$$

Here A represents the absorbing states and S represents the transient states. The matrix Q contains the probabilities of transitions between the transient states. The block matrix H contains the transition probabilities from the transient states to the absorbing states.

The transition probabilities matrix that represents the transformation of the system after k steps is given by forming powers of the single step matrix P, that is, P^k [77]. This k-step transition probability matrix P^k has the following form

$$P^k = \begin{bmatrix} Q^k & H' \\ 0 & I \end{bmatrix}$$

The Q^k_{ij} entry of the matrix Q^k denotes the probability of arriving in transient state S_j after exactly k steps starting from transient state S_i [77]. The block matrix H' is of no use in our formulation and therefore will not be analysed.

Hence the probability of arriving in transient state S_j after (exactly 0, or exactly 1, or ..., or exactly k) steps, starting the system from transient state S_i, is given by W_{ij} where

$$W = Q^0 + Q^1 + Q^2 + \cdots + Q^k$$

$$W = I + Q + Q^2 + \cdots + Q^k = \sum_{i=0}^{k} Q^i$$

It is shown in [18] that if $Q^k \rightarrow 0$, when $k \rightarrow \infty$ (which is the case here since Q is a matrix of probabilities), then

$$I + Q + Q^2 + \cdots + Q^k \approx (I - Q)^{-1}$$

Hence the limiting value of W as k increases is very close to the matrix inverse $(I - Q)^{-1}$; this is called the fundamental matrix of the Markov chain [77].

The matrix $(I - Q)^{-1}$, which we will now refer to as W, enables us to calculate the transition probabilities we need. The probability of the system reaching state j after some number of steps, starting in state i, is W_{ij}.

Now we can calculate the probability of reaching state T, after starting the system in state S_1 (corresponding to module M_1).

Given that

- W_{1i} is the probability of reaching state i from state 1 (after an unspecified number of steps) and tells us whether a state has been reached at some stage, as analysed in [77, page 312]. For a non-absorbing state control may have been passed on.

- R_iP_{iT} is the probability of reaching state T from state i in one step;

Let P_{S_i} be the probability that starting from state 1 the system reaches state S_i after an arbitrary number x of steps and then in one further step reaches T directly from S_i (fig.4.4). As the probability R_iP_{iT} does not depend on W_{1i}, then

$$P_{S_i} = W_{1i}R_iP_{iT}$$

The estimated overall reliability of a system R_{est} will then be the probability that starting in state 1 the system enters the absorbing state T, from any state S_i (fig.4.5). Then

$$R_{est} = P_{S_1} + P_{S_2} + \cdots + P_{S_n} = \sum_{i=1}^{n} P_{S_i}$$

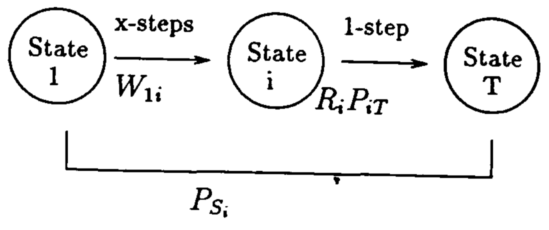

Figure 4.4: Probability of reaching state T from state 1

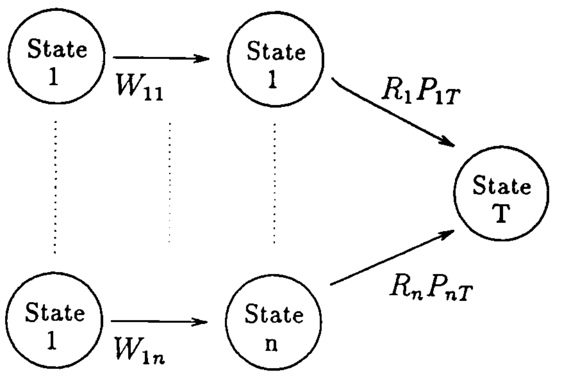

Figure 4.5: Overall probability of reaching state T from state 1

$$R_{est} = \sum_{i=1}^{n} W_{1i}R_iP_{iT} \tag{4.1}$$

Formula (4.1) allows us to determine the overall reliability R_{est} of a system from the reliability of each module R_i, and the transition probabilities P_{ij} and P_{iT}. In chapter 5 this formula is utilized to find the values of R_i corresponding to R_{est}, P_{ij} and P_{iT}, which are known in advance, where a cost constraint is introduced. Thus, we claim, we will be able to analyse the feasibility of the trade-off between a cost against a required reliability, during the early stage of the design phase.

Formula (4.1) is a generalization of that given in [10], where here there are no restrictions on the number of states that can reach state T. In [69] this formula is briefly cited.

4.3.1 Example of utilization

An example of the utilization of formula (4.1) is shown below.

For a system with three modules, the general transition probabilities matrix P is

$$P = \begin{bmatrix} 0 & R_1P_{12} & R_1P_{13} & (1-R_1) & R_1P_{1T} \\ R_2P_{21} & 0 & R_2P_{23} & (1-R_2) & R_2P_{2T} \\ R_3P_{31} & R_3P_{32} & 0 & (1-R_3) & R_3P_{3T} \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The transition probabilities matrix Q within the transient states is

$$Q = \begin{bmatrix} 0 & R_1P_{12} & R_1P_{13} \\ R_2P_{21} & 0 & R_2P_{23} \\ R_3P_{31} & R_3P_{32} & 0 \end{bmatrix}$$

so that

$$I - Q = \begin{bmatrix} 1 & -R_1P_{12} & -R_1P_{13} \\ -R_2P_{21} & 1 & -R_2P_{23} \\ -R_3P_{31} & -R_3P_{32} & 1 \end{bmatrix}$$

The inversion of a 3×3 matrix L is given by

$$L = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}$$

$$L^{-1} = \frac{1}{|L|} \begin{bmatrix} b_2c_3 - b_3c_2 & a_3c_2 - a_2c_3 & a_2b_3 - a_3b_2 \\ b_3c_1 - b_1c_3 & a_1c_3 - a_3c_1 & a_3b_1 - a_1b_3 \\ b_1c_2 - b_2c_1 & a_2c_1 - a_1c_2 & a_1b_2 - a_2b_1 \end{bmatrix}$$

Hence

$$(I - Q)^{-1} = W = \frac{1}{|I - Q|} \times$$

$$\begin{bmatrix} 1 - R_2P_{23}R_3P_{32} & R_1P_{13}R_3P_{32} + R_1P_{12} & R_1P_{12}R_2P_{23} + R_1P_{13} \\ R_2P_{23}R_3P_{31} + R_2P_{21} & 1 - R_1P_{13}R_3P_{31} & R_1P_{13}R_2P_{21} + R_2P_{23} \\ R_2P_{21}R_3P_{32} + R_3P_{31} & R_1P_{12}R_3P_{31} + R_3P_{32} & 1 - R_1P_{12}R_2P_{21} \end{bmatrix}$$

Using formula (4.1) we have

$$\begin{aligned} R_{est} &= \frac{1}{|I-Q|} \cdot \{(1 - R_2P_{23}R_3P_{32})(R_1P_{1T}) + \\ &\quad (R_1P_{13}R_3P_{32} + R_1P_{12})(R_2P_{2T}) + \\ &\quad (R_1P_{12}R_2P_{23} + R_1P_{13})(R_3P_{3T})\} \end{aligned}$$

As an example of an application of this formula, consider a system with the simple transition structure shown in fig. 4.6.

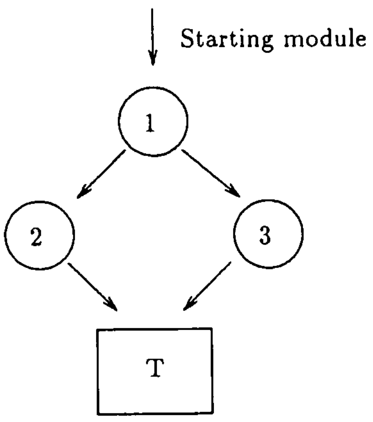

Figure 4.6: System with 3 modules

Given

$P_{12} = 0.6$; $P_{13} = 0.4$; $P_{1T} = 0$; $R_1 = 0.9$;

$P_{21} = 0$; $P_{23} = 0$; $P_{2T} = 1$; $R_2 = 0.99$;

$P_{31} = 0$; $P_{32} = 0$; $P_{3T} = 1$; $R_3 = 0.9$;

Then $R_{est} = 0.8586$, that is, the probability that the system produces a final result in according with its specification will be 85.86%.

4.4 Allocation of the Reliability

As an illustration of the application of the formula (4.1), we will ouline a method for allocating the values R_i so that if the modules M_i attain reliability R_i then the desired overall software system reliability R_{req} will be achieved. There are no other constraints. In the following chapter this problem is re-examined, by focusing on how to find the reliabilities R_i so that the development cost is a minimum.

This problem can be summarized as follows.

We seek values of $< R_1, \ldots, R_n >$ which will give a value of R_{est} close or equal to R_{req}, but we can only use values of R_i with $0 < R_i < 1$, and such that we obtain $R_{est} \geq R_{req}$.

We know:

a. the required reliability R_{req}, which is given in advance;

b. the transition probabilities P_{ij} and P_{iT}, which are obtained from the requirement specification;

To accomplish this task we have the following formulae, which are described in section 4.3:

$$R_{est} = \sum_{i=1}^{n} W_{1i} R_i P_{iT}$$

$$\text{where } W = X^{-1} \text{ with } X_{ij} = \begin{cases} 1 & i = j \\ -R_i P_{ij} & i \neq j \end{cases}$$

We can summarize this problem description as follows:

Find values of $\underline{R}$ such that R_{est} is minimized, subject to

$R_{est}(\underline{R}) - R_{req} \geq 0;\ 0 < R_i < 1$, where $i = 1, \ldots, n$

The resulting solution of this minimization problem[2] will be the allocation of the reliabilities $R_1, \ldots, R_n$.

It should be noted that if it was known in advance that the required reliability could always be achieved exactly, then a more direct method could be used to solve the problem. An appropriate root-finding technique, such as Newton's method, could be employed to find values of $\underline{R}$ which satisfy

$$f(\underline{R}) \equiv R_{est}(\underline{R}) - R_{req} = 0$$

However, the problem of constraining the permitted values for R_i would complicate this approach. In general, we cannot expect a value of R_{est} equal to R_{req} to be attainable and hence the minimization approach suggested here has a wider application.

[2]A complete solution for this problem, with examples, can be seen in [9]. This solution makes use (roughly) of the same framework discussed in chapter 5 (this being the reason why it is not discussed in this chapter) where the NAG routine E04VDF is used for the minimization procedure.

Chapter 5

Scenarios for Cost and Reliability

5.1 Problem Description

In this chapter we describe how some scenarios for estimating the overall cost of development associated with an overall required reliability can be constructed. These scenarios enable us to know, during the design phase, before any line of code has been coded or tested, the minimum overall cost of development of a modular software system. This cost is considered here to be the cost of the coding plus the testing phase, taking into account different levels of overall required reliability. As a direct outcome of these scenarios, the estimated cost and targeted reliability for each individual module are obtained.

The task above is carried out by employing the results obtained in chapter 3–the development cost of a module linked to its reliability; and chapter 4–overall estimated system reliability based on the reliability of each module.

To accomplish this task we develop in this chapter a minimization method that enables us to allocate values R_i (the level of reliability that should be aimed at for each module), such that if the modules M_i attain reliabilities R_i then the estimated overall software system reliability R_{est} will at least be equal to the overall required reliability R_{req}. Our minimization method produces a set of values R_i such that the (presumed) minimum overall cost of development will be achieved, taking into account the reliability constraints.

By knowing the predicted values R_i, $(C_{dev})_i$, R_{req} and the overall cost of development C_{tot} for a number of scenarios, a project manager could evaluate these different scenarios for cost and reliability before allocating the resources for the coding and testing phases.

This minimization problem can be summarized as follows:

Find a value for $\underline{R}$ such that $C_{tot} = \sum_{i=1}^{n} (C_{dev})_i$ is minimized, subject to

- $R_{est}(\underline{R}) \geq R_{req}$

  This constraint is to ensure that the set of reliability levels R_i allocated to each module will yield an overall estimated reliability that is at least equal to the required reliability.

  $R_{est} = \sum_{i=1}^{n} W_{1i} R_i P_{iT}$ (equation 4.1)

  where $W = X^{-1}$ with $X_{ij} = \begin{cases} 1 & i = j \\ -R_i P_{ij} & i \neq j \end{cases}$

  $i = 1, \cdots, n$ (number of modules in the modular system under estimation)

- $0 < N_i \leq F_i$

  As the number N_i of faults to be removed during the testing phase is linked to R_i, as defined in equation (3.3), then this constraint must be established to avoid allocating a value for R_i that produces a meaningless figure for N_i. This erroneous situation might occur, such as $N < 0$, depending on the values of R_i and β_i that are used in equation (3.3). Hence, the values for R_i should be established so as to avert that inconsistency.

- $0 < R_i < 1$

  Although the maximum theoretical value for reliability is 1.0, that is, a module reliability equal to 100%, this value is probably never achieved in practical software systems. Thus, the minimization method has to take into consideration this constraint, so that the maximum resulting values for each R_i be set less than 1.0.

The resulting solution of the minimization problem described above will be an allocation of the reliabilities $R_1, \cdots, R_n$. Consequently, the individual cost $(C_{dev})_i$ of each module can be estimated, as well as the overall cost C_{tot}, as discussed subsequently.

5.2 Framework of the Minimization

To deal with the problem stated above the following points are considered:

a. It is *not* of concern that the minimization method yields the "optimum" solution (performance, response time, precision or whatever the term means) for the problem described. It is intended to employ a simple and easy-to-handle method that produces a sound result, demonstrating the feasibility of constructing the scenarios discussed earlier. So, any method that yields a result taking into account the inputs and constraints required may be used[1].

b. The method must be simple to use and easily understood by a project manager.

[1]In [5, 63] can be found other methods, which are employed in other contexts, that deal with the reliability optimization problem for a modular software system. These methods deal with the application of optimization to determine the optimal redundancy level of fault-tolerant software systems, in order to maximize overall software reliability.

c. The intrinsic features of the method, that is, the internal details of the algorithm embodied in the minimization program, are not a matter of direct concern here. The algorithm will be treated as a black-box.

Allowing for the points described above, the published NAG routine E04UCF [52] (which is able to handle the specified inputs and constraints, to produce the desired outcome) is employed for the minimization. The routine is used as depicted in figure 5.1 and commented upon subsequently.

Briefly summarized[2], it can be said that the routine E04UCF is designed to minimize[3] an objective function subject to constraints, which may include bounds on the variables, and linear and non-linear constraints. In the context of this work:

- the objective function is the function $C_{tot}(\underline{R})$ that we want to minimize.
- bounds on the variables represent the acceptable range of variation for the values R_i.
- non-linear constraints are the constraints on $R_{est}(\underline{R})$ and N_i discussed in section 5.1.

In addition, the NAG routine F04AAF is also utilized, which solves a system of equations with multiple right-hand sides and thereby allows us to calculate W_{1i}, as analysed subsequently.

[2]As noted previously, the algorithm of minimization is treated in this work as a black-box. Anyone interested in details about the algorithm used by E04UCF should consult the reference [52], as well as the reference [20], which is said in [52] to provide a detailed discussion of the features of the method of E04UCF.

[3]It has to be stressed that the routine E04UCF finds a local minimum only, that is, the final result produced by E04UCF is *not* guaranteed to be the global minimum.

```
Begin Program
Supply   Reliability required $R_{req}$;
Supply   Upper bounds on reliabilities $R_i$, as constraints for routine E04UCF;
Supply   Data required for $C_{dev}$;                                  *see Note 1*
Supply   Transition matrix (values $P_{ij}$ and $P_{iT}$);
Supply   Control parameters for E04UCF;                                *see Note 2*
Begin E04UCF                                                   *minimization routine*
   Count-of-iterations= 0;
   Repeat
      Generate   $R_1, \cdots, R_n$;                                   *see Note 3*
      Call       Routine to evaluate $W_{1i}$ and constraints;         *see Note 4*
      Call       Routine to calculate $C_{tot}$;                       *see Note 5*
      If    $R_1, \cdots, R_n$ constitute an optimal solution          *see Note 6*
         then    $R_1, \cdots, R_n$ are the results needed;
                 Return-code= 0;
                 Exit     E04UCF;
      Endif;
      If    E04UCF considers it is pointless to continue               *see Note 7*
         then    Return-code$\neq$ 0
                 Exit     E04UCF;
      Endif;
      Increment  Count-of-iterations;
   Until    Limit on maximum number of iterations has been reached;
            Return-code= 4;                                            *see Note 8*
End E04UCF;
If  Return-code= 0
   then  the values $R_1, \cdots, R_n$ produced by E04UCF are the results required;
         calculate $C_{tot}$;
   else  examine the return-code generated;                            *see Note 9*
Endif;
End Program.
```

Figure 5.1: Framework of the minimization

Regarding figure 5.1, notes are used in relation to points that are worth discussing in order to make clear some of the procedures performed. These notes are explained below.

Note 1

The following data are required to estimate the cost $(C_{dev})_i$ of development for each module, and thus the overall cost C_{tot}. For each module considered, the following data must be supplied, as discussed in chapter 3.

- P_{dev}

  The cost of development per unit time to be applied for all modules, which must be consistent with the unit of effort that is being used. If the effort of development is expressed, for example, in man-months, then P_{dev} must be expressed in pounds per man-month.

- $(E_{cod})_i$

  The effort of coding, which is estimated as shown in chapter 2. Again, the unit utilized for this effort must be consistent with the unit used for P_{dev}.

- β_i

  The probability that a residual fault will produce a failure in the module, where $0 < \beta_i \ll 1$. As discussed in section 3.2.2, further study should address how to obtain a good estimate of β_i. It is assumed here that there exists a β_i for each module under estimation.

- F_i

  The number of faults assumed to be present in the module after the end of the coding phase (section 3.2.3). It must be emphasized that F_i is presumed to depend on the module size, which, in turn, is estimated as discussed in chapter 2.

- δ_i and γ_i

  As characterized in chapter 3, these two parameters represent, respectively, the ratio between the effort of removing the Nth and the first fault, and the relationship between the effort of coding and the maximum effort of testing.

Note 2

The routine E04UCF requires some control parameters, such as initial crude estimates for $R_1, \cdots, R_n$, accuracy required for the solution, maximum number of iterations that should be performed when finding a solution, and which first derivatives (gradients) are supplied. It must be emphasized that the initial estimates for $R_1, \cdots, R_n$ have a significant effect on the outcome of the minimization, since the function $C_{tot}(\underline{R})$ appears to have several local minima.

Note 3

This is the fundamental procedure performed by E04UCF. The routine generates a new set of values $R_1, \cdots, R_n$ derived from the values used in the previous iterations to find a point that is feasible (complies with the defined bounds and constraints). The nonlinear constraints will not generally be satisfied until an "optimal" point is reached (see note 6).

Initial estimates must be supplied, because the routine requires a starting point for each R_i. However, there is no precise guidance as to how to choose this initial estimate. As stated in [52], these initial estimates must be "an initial estimate of the solution".

By experiment it was found that taking initial estimates for each R_i in the interval between R_{req} and the upper bound on the acceptable module reliability limit produced satisfactory results. Then, the initial estimates, in the minimization method developed, are established based on the upper bounds for the reliability of each module and the level of overall required reliability. The formula representing this situation is as follows.

$$sp_i = R_{req} + (bu_i - R_{req}) \times adj_i$$

- sp_i is the starting point for reliability of the module i;
- bu_i is the upper bound for the reliability of the module i. As stated on page 88, the bounds on the variables R_i must be supplied. The lower bound for R_i is zero. Although the maximum theoretical value for reliability is 1.0, it is not achieved in practical software systems. Thus, the upper bound must be set less than 1.0. In the following examples, the upper bounds bu_i are set to 0.99.
- adj_i is the factor of adjustment for choosing a suitable value between R_{req} and bu_i as the starting point. This value was determined by trial and error (the values utilized are shown in each example).

Note 4

A routine was developed to calculate $\{W_{1i}\}$ and the constraints considered (R_{est} and N_i).

To calculate $\{W_{1i}\}$, the NAG routine F04AAF is utilized. This routine, as stated earlier, solves a system of equations with multiple right-hand sides and thereby calculates W_{1i}.

As analysed in chapter 4, $\{W_{1i}\}$ is the first row of an inverse matrix with the following characteristics:

$$W = X^{-1} \text{ with } X_{ij} = \begin{cases} 1 & i = j \\ -R_i P_{ij} & i \neq j \end{cases}$$

To find the first row of X^{-1}, that is, $\{W_{1i}\}$, we have

$$X^T (X^{-1})^T = I$$

and so solving

$$X^T \underline{r} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

will give us $\underline{r}^T$ as the required answer $\{W_{1i}\}$. Putting the problem in this form means we can take advantage of standard routines, as provided by the NAG routine F04AAF. This routine can be used to solve the equation $A\underline{x} = \underline{b}$, where, in this case, $A = X^T$, $\underline{x} = \underline{r}$ and $\underline{b} = (1, \ldots, 0)^T$.

The relation $\underline{r}^T = [W_{11}, \cdots, W_{1n}]$ is obtained using the following line of reasoning:

$$X^{-1} = \begin{bmatrix} X_{11} & X_{12} & \ldots & X_{1n} \\ X_{21} & X_{22} & \ldots & X_{2n} \\ \vdots & & & \\ X_{n1} & X_{n2} & \ldots & X_{nn} \end{bmatrix}$$

$$(X^{-1})^T = (X^T)^{-1} = \begin{bmatrix} X_{11} & X_{21} & \ldots & X_{n1} \\ X_{12} & X_{22} & \ldots & X_{n2} \\ \vdots & & & \\ X_{1n} & X_{2n} & \ldots & X_{nn} \end{bmatrix}$$

$$X^T \underline{r} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Then

$$\underline{r} = (X^T)^{-1} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\underline{r} = \begin{bmatrix} X_{11} & X_{21} & \dots & X_{n1} \\ X_{12} & X_{22} & \dots & X_{n2} \\ \vdots & & & \\ X_{1n} & X_{2n} & \dots & X_{nn} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\underline{r} = \begin{bmatrix} X_{11} \\ X_{12} \\ \vdots \\ X_{1n} \end{bmatrix}$$

And

$$\underline{r}^T = \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1n} \end{bmatrix}$$

To use E04UCF, as many first partial derivatives as possible should be provided for constraints functions and objective function. Unspecified derivatives are approximated by finite differences. Thus, in addition to the evaluation of N_i and R_{est}, the first partial derivatives of the function N_i with respect to R_i are also required. The first partial derivatives of the function R_{est} are not provided, due to the complexity of its formulation.

Note 5

A routine was provided which calculates the objective function $C_{tot} = \sum_{i=1}^{n} (C_{dev})_i$, using the data mentioned in Note 1.

Note 6

According to [52], an optimal solution is found when:

a. The partial derivatives of the function $C_{tot}(\underline{R})$ with respect to R_i are sufficiently small, considering the accuracy required[4]; and

b. The residuals[5] of constraints are sufficiently small, again considering the accuracy required; and

c. The values for $R_1, \ldots, R_n$ do not change significantly between iterations.

As analysed in [52], "there are several optional parameters in E04UCF which define choices in the behaviour of the routine. In order to reduce the number of formal parameters of E04UCF these optional parameters have associated default values that are appropriate for most problems. Therefore the user need only specify those optional parameters whose values are to be different from their values". The optional parameters *function precison* and *feasibility tolerance* provide, respectively, the accuracy required and the tolerance for the residuals.

[4]Differences between two consecutives calculations for the objective function.

[5]Differences between the value provided to and that established by E04UCF.

Note 7

In this case the routine E04UCF terminates and a return code is generated to indicate the likely cause of this abnormal exit (see Note 9). Some situations can arise that prevent E04UCF from progressing, such as no feasible point can be found for the nonlinear constraints, the accuracy required cannot be achieved (because it is too small), or the upper bounds limits do not permit the location of a feasible solution.

Note 8

If the program terminates with a return code equal to 4, the limit on the maximum number of iterations has been reached without any feasible values for $R_1, \cdots, R_n$ being found. If it is decided that the routine needs to perform more iterations to find a solution, then the value of the limit on the number of iterations should be set higher and the program re-run.

Note 9

Depending on the return code generated, the values yielded by E04UCF may still be considered valid results for $R_1, \cdots, R_n$. The user should examine the return code [52] and the messages produced and, if necessary, change the parameters required and re-run the program.

5.3 Example

To illustrate the utilization of the method developed, an example will be presented using mainly hypothetical (but realistic) data, which serves to clarify how a project manager can obtain the scenarios for cost and reliability.

The reason for using mainly hypothetical data is that, as discussed in chapters 3 and 4, there are some parameters introduced in this work for which there are no suitable data available in the literature. Incidentally, it is pretty clear that the utilization of the data from real software development would be a better way to evaluate our approach. However, to collect the data that would be needed was not feasible during the development of this research. So, the example shown below contains a mixture of real data, where these are available, and hypothetical data for the remaining parameters.

In section 6.3 the sensitivity of cost of development in relation to each parameter is analysed, and the influence that a bad estimate may have on C_{tot} is discussed.

5.3.1 Definition

Consider a hypothetical project, where what is needed is to construct scenarios for the overall cost of development depending on the level of the overall required reliability.

During the design phase the structure for the software system, and how control is passed between modules, is obtained. These are assumed to have the behavioural features characterized in chapter 4; this behaviour and structure for the example are depicted in figure 5.2. The hypothetical example utilized in this chapter is an adaptation of the example shown in [55, page 222], which characterizes a software that enables a homeowner to configure the security system in his house when it is installed, monitors all sensors connected to the security system and interacts with the homeowner through a key pad and function conatined the the system control panel.

The six modules involved are:

- M_1 - Interact with the user; M_2 - Configure system; M_3 - Activate/deactivate system; M_4 - Monitor sensors type 1; M_5 - Display messages and status; M_6 - Monitor sensors type 2.

We would like to clarify that the concept of module utilized throughout this work is to be seen as a functional unit which is identified during the architectural design. The "module" should be, as close as feasible to obtain in this early stage of software development, the actual software module that will be implemented during the coding phase.

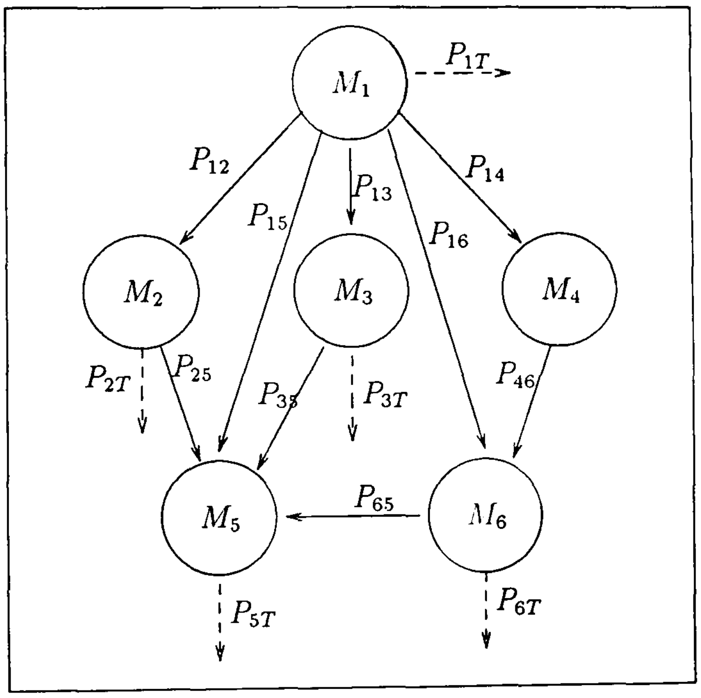

Figure 5.2: Modular software system used in the example

The transition matrix corresponding to the modular system is shown in figure 5.3, where T represents the terminal state.

The following information is also proposed for the system under consideration:

- Module sizes

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & T \\
1 & 0.00 & 0.25 & 0.30 & 0.20 & 0.10 & 0.05 & 0.10 \\
2 & 0.00 & 0.00 & 0.00 & 0.00 & 0.90 & 0.00 & 0.10 \\
3 & 0.00 & 0.00 & 0.00 & 0.00 & 0.30 & 0.00 & 0.70 \\
4 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\
5 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\
6 & 0.00 & 0.00 & 0.00 & 0.00 & 0.10 & 0.00 & 0.90
\end{array}
$$

Figure 5.3: Transition matrix for the example

It is hypothesized that the following module sizes (in lines of code) have been estimated for each module:

$M_1 = 4000$; $M_2 = 1500$; $M_3 = 4000$; $M_4 = 1500$; $M_5 = 1500$; $M_6 = 3500$.

- Effort of coding

Shown in [31] are some relationships between the effort of coding (man-months) and software size S (thousands of lines of code) for software developed in an Algol-like language, where all projects are system utilities, such as job scheduling and tape management. The effort of coding in relation to software size S in KLOC (thousand of lines of code) is indicated there[6] to be

$$E_{cod} = 1.7 \cdot S^{0.82} \tag{5.1}$$

[6]The formula shown in [31] enables us to estimate the overall effort expended in a system utilities project; this effort is said to be $E = 4.27 \cdot S^{0.82}$. A percentage break-down of effort by phase is presented, which permits us to say, very roughly, that $E_{cod} \approx 0.4E$. Then $E_{cod} \approx 1.7 \cdot S^{0.82}$.

Hypothesizing that estimation for our example can be based on formula (5.1), then the effort of coding E_{cod} for each module is estimated as:

$(E_{cod})_1 = 5.3$; $(E_{cod})_2 = 2.4$; $(E_{cod})_3 = 5.3$; $(E_{cod})_4 = 2.4$; $(E_{cod})_5 = 2.4$; $(E_{cod})_6 = 4.7$.

- Number of faults present in each module

  Allowing for the module sizes indicated above and the assumption of an Algol-like programming language, the following values for the number of faults expected to be present in each module after the coding phase are estimated using formula (3.4):

  $F_1 = 115$; $F_2 = 38$; $F_3 = 115$; $F_4 = 38$; $F_5 = 38$; $F_6 = 99$.

- Cost of development

  It is hypothesized that P_{dev} =£2,000.00 per man-month, which indicates that a junior programming team could be employed for the coding and testing phases.

- β

  As there is no suitable data available that enable us to obtain directly β, we have to assume a value for this parameter. It is hypothesized that the probability that a fault produces a failure is $\beta = 0.005$ for all six modules. To verify, very crudely, that this value for β is not disparate, the following line of argument is utilized.

  ⋆ It has been suggested [6, 68] that a standard expression for software reliability is given by $R = e^{\eta \cdot t}$, where η is the failure rate in any time interval and t is the length of the time interval in which the reliability is estimated.

- ⋆ Let us consider $t = 1$ run, that is, the time interval corresponds to 1 run of the software module. Then, there are $\eta = -\ln R$ failures per run.
- ⋆ Suppose a high value for R, say, $R = 0.96$. Then, $\eta = 0.04$ failures per run, which, following the example of estimation of β shown on page 52, produces $\beta = 0.008$.

Thus, it can be said that a hypothesis of $\beta = 0.005$ seems to be reasonable.

- δ and γ

These are parameters for which there are no precise data available. Therefore, they need to be hypothesized. Although δ clearly varies with the number of faults to be removed, we will assume here that δ is constant for the purpose of this example. As will be seen in chapter 6, C_{dev} is not very sensitive to δ. Consequently, using a constant value for δ in this hypothetical example should not produce significant distortions in the results produced for C_{dev}.

It is hypothesized that for every module $\delta = 5.0$ and $\gamma = 1.0$, since these appear to be reasonable values (see footnotes on pages 59 and 60).

5.3.2 Some examples of scenarios

To produce some examples of scenarios, a program written in Fortran 77 was developed so as to implement the minimization method proposed. This program follows the steps and requirements depicted in figure 5.3. In all scenarios the basic input data were the same and equal to the data discussed in section 5.3.1. These data are summarized in Table 5.1.

Using the input data shown in Table 5.1, some scenarios for the overall cost of development are built up, where different values for the overall required reliability

Module	E_{cod}	F
1	5.3	115
2	2.4	38
3	5.3	115
4	2.4	38
5	2.4	38
6	4.7	99
For all modules: P_{dev} =£2,000.00; Upper bound (reliabilities):0.99; $\beta = 0.005$; $\delta = 5.00$; $\gamma = 1.00$		

Table 5.1: Input data employed in the example

R_{req} are employed in each case. These results are summarized in Table 5.2. A detailed view of each scenario is depicted in Tables 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8.

In all scenarios the following information is yielded for each module:

⋆ R is the estimated reliability that is achieved if N faults are removed.

⋆ N is the estimated number of faults that have to be removed in order to achieve the reliability R.

⋆ C_{dev} is the cost of development.

⋆ E_{dev} is the estimated effort of development, which is obtained using $\frac{C_{dev}}{P_{dev}}$.

The values estimated and required for the overall system reliability are also presented in all scenarios, as well as the values employed as a starting-point for

Scenario	R_{req}	C_{tot} (£)	Table
1	0.75	58689.19	5.3
2	0.80	63610.65	5.4
3	0.85	68686.47	5.5
4	0.90	74626.28	5.6
5	0.95	81624.97	5.7
6	1.00	90000.00	5.8

Table 5.2: Cost estimate for each scenario

the reliabilities (see Note 3 on page 91). The scenarios and results obtained are commented upon in sequence.

Module	E_{dev}	N	R	C_{dev} (£)
1	9.5078	103	0.943	19015.65
2	2.9229	21	0.921	5845.73
3	6.1370	57	0.748	12274.03
4	2.4697	13	0.884	4939.39
5	2.9229	21	0.921	5845.73
6	5.3843	48	0.778	10768.65
	29.3446		C_{tot} $\longrightarrow$	58689.19

$R_{est} = 0.755$

Starting point for the reliability of each module: 0.894

Factor of adjustment for this starting point: 0.6

Table 5.3: Scenario for cost of development with $R_{req} = 0.75$

Module	E_{dev}	N	R	C_{dev} (£)
1	9.6859	105	0.952	19371.84
2	2.5420	15	0.894	5083.92
3	8.3971	90	0.884	16794.11
4	2.5420	15	0.892	5083.92
5	3.0952	23	0.928	6190.50
6	5.5432	51	0.789	11086.37
	31.8053		$C_{tot} \longrightarrow$	63610.65

$R_{est} = 0.80$

Starting point for the reliability of each module: 0.912

Factor of adjustment for this starting point: 0.592

Table 5.4: Scenario for cost of development with $R_{req} = 0.80$

Module	E_{dev}	N	R	C_{dev} (£)
1	10.0470	109	0.974	20094.08
2	2.4059	9	0.867	4811.75
3	10.2299	111	0.985	20459.85
4	2.4059	9	0.867	4811.75
5	3.5975	28	0.955	7194.94
6	5.6570	53	0.797	11314.09
	34.3432		$C_{tot} \longrightarrow$	68686.47

$R_{est} = 0.85$

Starting point for the reliability of each module: 0.945

Factor of adjustment for this starting point: 0.68

Table 5.5: Scenario for cost of development with $R_{req} = 0.85$

Module	E_{dev}	N	R	C_{dev} (£)
1	10.2299	111	0.982	20459.85
2	3.8207	30	0.963	7641.37
3	9.8657	107	0.964	19731.36
4	2.7039	18	0.907	5407.82
5	3.9361	31	0.968	7872.26
6	6.7568	69	0.861	13513.62
	37.3131		C_{tot} $\longrightarrow$	74626.28

$R_{est} = 0.90$

Starting point for the reliability of each module: 0.913

Factor of adjustment for this starting point: 0.15

Table 5.6: Scenario for cost of development wih $R_{req} = 0.90$

Module	E_{dev}	N	R	C_{dev} (£)
1	10.3219	112	0.989	20643.84
2	3.9361	31	0.969	7872.26
3	10.1383	110	0.979	20276.60
4	3.5975	28	0.954	7194.94
5	4.1737	33	0.980	8347.49
6	8.6449	91	0.964	17289.86
	40.8125		C_{tot} $\longrightarrow$	81624.97

$R_{est} = 0.951$

Starting point for the reliability of each module: 0.954

Factor of adjustment for this starting point: 0.1055

Table 5.7: Scenario for cost of development with $R_{req} = 0.95$

Module	E_{dev}	N	R	C_{dev} (£)
1	10.6	115	1.0	21200.00
2	4.8	38	1.0	9600.00
3	10.6	115	1.0	21200.00
4	4.8	38	1.0	9600.00
5	4.8	38	1.0	9600.00
6	9.4	99	1.0	18800.00
	45.0		C_{tot} $\longrightarrow$	90000.00

$R_{est} = 1.00$

Starting point for the reliability of each module: 1.00

Factor of adjustment for this starting point: 0.0

Table 5.8: Scenario for cost of development with $R_{req} = 1.0$

5.3.3 Comments on the results obtained

Some supplementary comments and observations are made below about the example scenarios, and also on the general behaviour of C_{dev}.

- In addition to the scenarios 1 to 5 summarized in Table 5.2 (scenario 6 is just a special scenario), a minimum (C_{min}) and maximum (C_{max}) cost of development are also estimated.

  As can be seen from equation (3.13), we can regard the minimum cost of development as being the cost of coding. So, for m modules

  $$C_{min} = \sum_{i=1}^{m} P_{dev} E_{cod}$$

  Consequently, for the example system $C_{min} = 2000(5.3+2.4+5.3+2.4+2.4+4.7) = £45{,}000.00$.

  To obtain C_{max}, which would happen if $R_{req} = 1.0$, we relaxed the constraint of upper bound for R_i, such that $R_i \leq 1$. As shown in Table 5.8, the outcome is that each module also has to have reliability equal to 1.0.

- For the two scenarios shown in Tables 5.3 and 5.7, we note that the value of R_{est} obtained is slightly bigger than R_{req} . This is because it is not always possible to find values of R_i satisfying the constraints such that the resulting R_{est} is exactly equal to R_{req}–thus the value of C_{dev} may be slightly higher than R_{req} requires.

- Modules with the same characteristics (software size, β, δ, etc.) can obviously have different development cost, as can be seen in the results obtained for the modules 1 and 3, and modules 2, 4 and 5. Clearly this is because the cost depends on the reliability required for each module, as established by the minimization procedure. In figure 5.4 the curve of C_{dev} plotted against R is depicted for the three different sizes of module in the example. With this curve, a project manager would be able to estimate the value of C_{dev}, depending on the required level of reliability for the software.

- In figure 5.4 we see that the cost of development, for example, for the 4000-lines-of-code modules, is in the range £10,600.00—£21,200.00[7]. The lower end of the range corresponds to the situation in which the software is just coded (and not tested at all); the upper end represents the theoretical situation for reliability equal to 100%.

  It is worth emphasizing that the horizontal section of the curves shown in figure 5.4 indicates that below a certain level of required reliability, the estimated development cost is practically constant (and equals C_{cod}). Consider module 1; having spent C_{cod} on coding the software, we obtain $R \approx 0.6$. If our requirement is $R < 0.6$ for this module, we still have to spend C_{cod}, and obtain $R_{est} = 0.6$.

  The level of required reliability mentioned above, can be estimated using in equation (3.3) $N = 1$. By so doing, we are estimating the reliability that would be achieved after just one fault having been removed. This value is estimated as follows.

[7]As $\gamma = 1.0$, the maximum $E_{tes} = E_{cod}$. Therefore, the maximum $E_{dev} = 2 \cdot E_{cod}$.

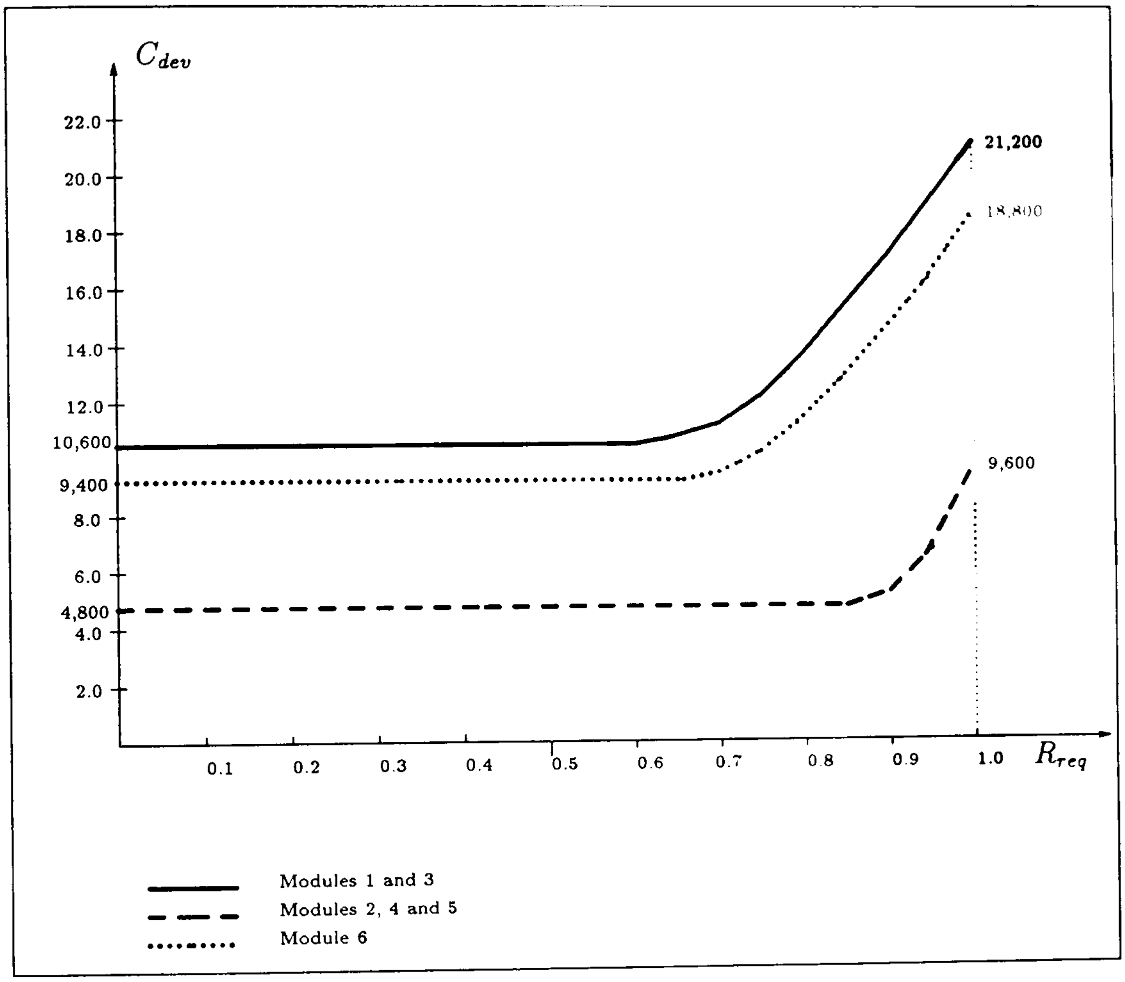

Figure 5.4: Plots of C_{dev} against R for individual modules

– Modules with 4000 lines of code

$$1 = 115 - \frac{\ln R}{\ln(1 - 0.005)} \qquad R = 0.564$$

– Modules with 1500 lines of code

$$1 = 38 - \frac{\ln R}{\ln(1 - 0.005)} \qquad R = 0.830$$

– Module with 3500 lines of code

$$1 = 99 - \frac{\ln R}{\ln(1 - 0.005)} \qquad R = 0.611$$

Hence, when $R > 0.564$ (4000-lines-of-code module), $R > 0.83$ (1500-lines-of-code module) and $R > 0.611$ (3500-lines-of-code module), each fault removed results in a significant increase in the cost C_{dev}. Below these thresholds C_{dev} is nearly constant.

- Considering the entire modular system, some further observations can be made:

  ⋆ The relationship between C_{dev} and the required reliability R_{req} is similar to that for an individual module, where the explanation for the flat area in the curve is the same as already discussed above.

  ⋆ It can be observed in figure 5.5 (as discussed for figure 5.4) that for $0.5 < R < 0.6$ there is a gradual increase of the exponential curve. This can be attributed to the fact that for the modules with 4000 lines of code, which exert a leading influence in C_{dev}, the reliability achieves a value between 0.5 and 0.6 when just one fault is removed, as analysed previously.

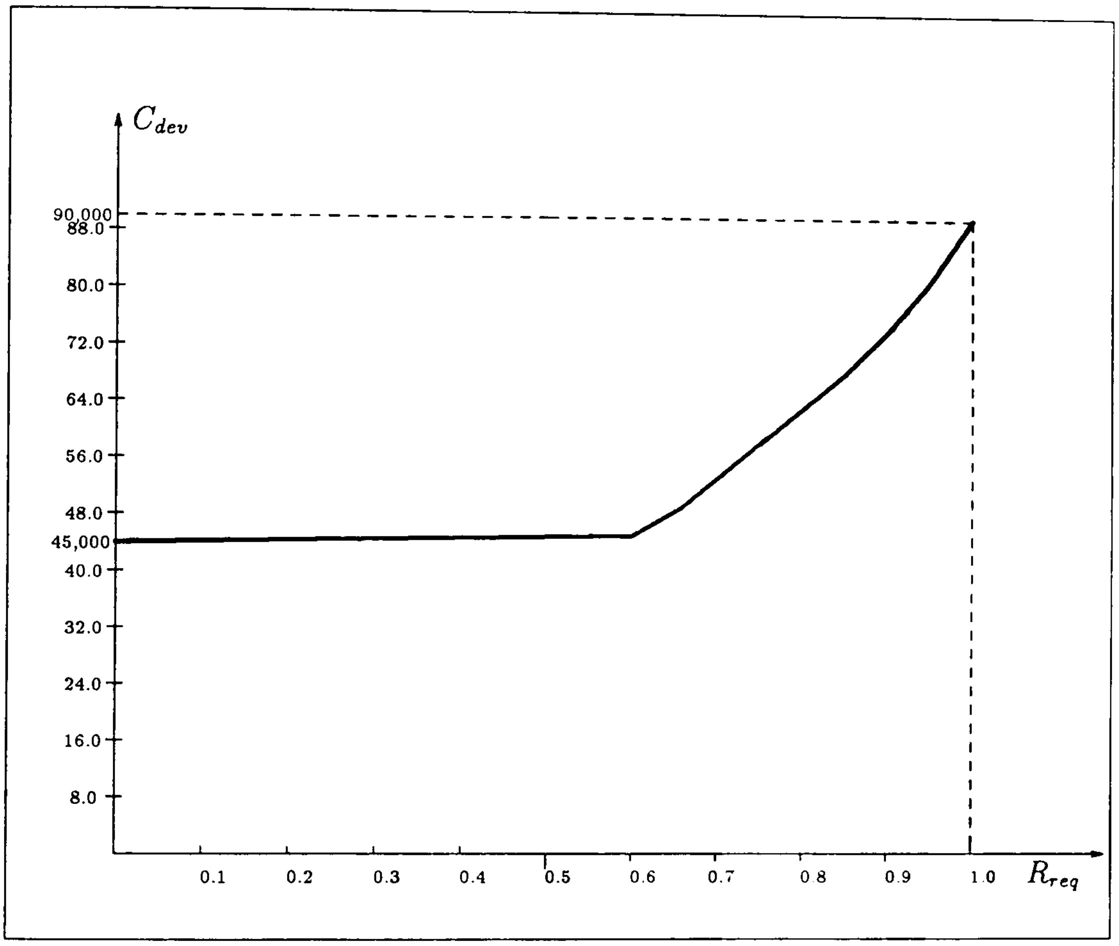

Figure 5.5: Plot of C_{dev} against R for the whole system

⋆ Another conclusion that may be drawn from the example shown is that the increase in the cost of each module does not follow the same pattern of the whole system. In other words, an increase of, say, 10% in C_{dev} does not necessarily imply, as can be seen, that a general increase of 10% for each module ensues.

⋆ The graph also allows us to perform inverse interpolation—given an upper limit on the budget, we can estimate the reliability that can be achieved.

5.3.4 Considerations

Despite the fact that we are using hypothetical (but as far as possible realistic) values for some parameters, which makes a precise comparison impossible, we present a very rough analysis of the results obtained here in comparison with other data available in the literature, in which, it should be stressed, a target reliability is not clearly cited. This comparison indicates that the results produced here, using the developed formulas, yield a reasonable outcome. A module with 4000 lines of code (modules 1 and 3 in the previous example) is used for this comparison.

Two different sources are used:

- Source 1$\rightarrow$ Using Boehm's formulas ([7]).

  Despite the fact that there are some criticisms of the results that Boehm's formula can produce in different environments, we make this comparison to have a first impression of our results in comparison with a well-known method.

  Considering Boehm's basic model, "which is suitable for most of the small to medium size projects, where the problem to be solved is sometimes unique", and supposing that our module with 4000 lines of code is classified in this sort of model, we have the following formula in [7] to estimate the total effort E

required (design+code+test) to produce software.

$E = 3.0 \times S^{1.12}$, where S is the module size in thousands of line of code.

Hence

$E = 3.0 \times 4^{1.12} = 14.2$ man-month.

Boehm recommends *for his method* that the effort spent in the coding and testing phase $E_{cod} + E_{v\&v}$ should be aproximately 60% of the total effort estimated above. So

$E_{cod} + E_{v\&v} = 14.2 \times 0.6 = 8.52$ man-month.

Thus the total cost of coding, testing and debugging a module of 4000 lines of code, where no required level of reliability has been stated, using Boehm's formula would be

$C_{dev} = 8.52 \times 2000 =$£17,040.00.

- Source 2→ Using data from [31].

  Total effort E of development (design+code+test) has the following formula in [31], which is based on real data gleaned in several different environments.

  $E = 4.27 \times 4^{0.82} \approx 13.3$ man-months.

  The percentage of effort spent in the coding and testing phases represents aproximately 80% of the value calculated above, again according to data shown in [31]. Thus $E_{cod} + E_{v\&v} = 13.3 \times 0.8 \approx 10.64$ man-month.

  Thus the total cost of coding and testing, using the data acquired in [31] would be $C_{dev} \approx 10.64 \times 2000 =$£21,280.00.

Hypothesizing that the module with 4000 lines of code has a high required reliability, say, higher that 90%, it can be seen in figure 5.4 (and in Tables 5.3 to

5.7) that the cost of development for this module is estimated here as being around £20,000.00.

So, we have the following estimates:

- From Boehm's method - £17,040.00
- From [31] - £21,280.00
- From our model - $\approx$ £20,000.00

Therefore, it may be said that the result produced here as expected has yielded a reasonably consistent estimate when compared with these two sources [8], however, it should again be emphasized, that a required level of reliability was used in the estimate.

It is claimed that the representation proposed here for C_{dev} is a reasonable estimate for the cost of development, and this estimate can be acquired during the design phase. The relationship between cost and reliability is modelled realistically and produces outcomes which are in line with other models. Furthermore, we are also incorporating the required level of reliability in the method, which, to our knowledge, has not been done elsewhere.

[8]Our result is even consistent with Boehm's method, which uses a completely different dataset in its formulation.

Chapter 6

Sensitivity Analyses

6.1 Introduction

This chapter presents an analysis of the sensitivity of the formula for the cost of development to variations in the parameters employed in that formula. Some expressions for these sensitivities are obtained, and a comparison of the sensitivity of the parameters is made.

Firstly, some features which are thought to be useful in judging the goodness of a model are briefly discussed. As shown in [12, pages 275-276], there are some *subjective* criteria that can be used for this purpose. According to [12] these criteria are:

- Objectivity

  a) Are the final estimates based on measurements and data that are obtained algorithmically?

  b) Do the estimates depend on subjective factors that can vary significantly with different estimators (for instance, the project managers) ?

The answer to question a) is (essentially) *yes*, because all parameters used for establishing the cost of development can be based on historical data and are supposed to have been obtained through regression analysis, which involves an algorithmic routine.

To question b) the answer is therefore *no*; as noted above, none of the parameters utilized are based on judgements or guesses by an estimator. However, as can be observed in Chapter 5, in this thesis we have "guessed" some parameters (guided by data that are available in the literature, when possible). The method can, in principle, be very objective—but, as stated previously, a lack of opportunity to obtain realistic data means that some parameters in our examples have, of necessity, been assessed subjectively.

Nevertheless, it can justifiably be claimed that our method should be characterized as *reasonably objective.*

- Ease of Use

  a) Is the data needed for the model easy to obtain?

  b) Is too much data needed?

  c) Is the information needed available early in the life cycle?

In response to question a), it cannot be realistically claimed that the data required are very easy to acquire. Instead, it can be argued that they are "feasible" to collect, which may involve considerable effort and some overheads during the design, coding and testing phases of previous system development.

As can be seen, there are several parameters involved in the method. To obtain some of these parameters a great deal of data will be handled. So, the answer to question b) is *yes*, in the sense that we would prefer to have a method which required rather less data to be processed.

As to question c), the answer is *yes*; the data are expected to be available during the early stages of design phase where the developed method is thought to be applied.

Thus, it cannot be claimed that this method is easy to use, in view of the effort required to acquire data on previous developments. However, the actual procedures are then straightforward and easily automated.

- Transportability

  Is the model so dependent on local data that it cannot be used in a different environment?

  The straightforward answer is *yes*, because the method should be calibrated for a specific environment, using local data. This fact is a common characteristic of software estimation methods and has the benefit of enabling a more accurate estimate (it may provide closer estimates based on a local reality).

  The method is generic, and can be applied in a range of environments. However, it remains to be seen, whether the method based on data for a given category of development, for example, business applications, can be applied in other environments in the same category with few, or no, adjustments. If we infer from the experience of other models (a case in point is Boehm's), it might be suggested that this kind of extrapolation is not very likely.

- Sensitivity

  Does a small change in one or more input parameter lead to a relatively large change in the model estimate?

  The issue of sensitivity is the subject of the next section.

Examining the issues of *objectivity*, *easy to use* and *transportability* we have some positive points which may suggest that it is a workable model (with respect to its "goodness").

6.2 Expression for Sensitivity

In mathematical algorithms a very small change in the intial data may sometimes cause extreme variations in the final outcome. A system that exhibits such characteristics is said to be ill-conditioned.

An analysis of *sensitivity* is an attempt to identify combinations of data values, within permissible limits, that can cause particularly significant variations in the results (as defined in [55]).

It is well-known that a partial derivative (gradient) of a function in relation to a single variable gives a clear indication of the effect that changes in that variable would have on the value of the function. Thus, the partial derivative of a function can enable us to assess the sensitivity of the function to changes in its parameters.

It is worth recalling (see [26]) that the idea involved in using partial derivatives is that we hold all of the independent variables in a function constant, except one, at some value of interest. Therefore, the function then becomes a function of the single remaining independent variable. We may then differentiate the function as if it were a function of that one variable.

To develop an expression that represents the concept of sensitivity outlined above, the following line of reasoning is used, where an example serves to illustrate the explanation.

Given a function $u = f(x, y)$, then the magnitude of sensitivity to x (or y) is represented by the relationship between the proportional variation of x (or y), and the proportional variation of u. This notion can be represented by

$$\left|\frac{\Delta u}{u}\right| = \Theta_x \cdot \left|\frac{\Delta x}{x}\right| \tag{6.1}$$

$$\left|\frac{\Delta u}{u}\right| = \Theta_y \cdot \left|\frac{\Delta y}{y}\right|$$

where,

- Δu means either a positive variation (an increment) or negative variation (a decrement) of the value of the function u.
- Δx, Δy mean a positive or negative variation of x and y, respectively.
- $\left|\frac{\Delta u}{u}\right|$ is the magnitude of the proportional variation of u.
- $\left|\frac{\Delta x}{x}\right|$, $\left|\frac{\Delta y}{y}\right|$ are the magnitudes of the proportional variations of x and y, respectively.
- Θ_x and Θ_y represent relationships between changes in x and y, and their consequent effect on u.

The value of Θ_x (the same reasoning is applied to Θ_y) can be interpreted as follows:

- $0 < \Theta_x < 1$

  Then $\left|\frac{\Delta u}{u}\right|$ will be less than $\left|\frac{\Delta x}{x}\right|$, which means that variations in x will affect u in a lesser proportion, that is, u will change more "slowly" than x.

- $\Theta_x = 1$

  Then $\left|\frac{\Delta u}{u}\right| = \left|\frac{\Delta x}{x}\right|$, meaning that variations in x will affect u in exactly the same proportion.

- $\Theta_x > 1$

  Then $\left|\frac{\Delta u}{u}\right| > \left|\frac{\Delta x}{x}\right|$, meaning that variations in x will affect u in a greater proportion, that is, u will change more "rapidly" than x.

- $\Theta_x = 0$

  In this case u is not affected at all by variations in x.

It can therefore be concluded that Θ_x provides the required sensitivity function. Hence, for instance, if the value of function Θ_x (for some set of values of the parameters x and y) is less than the value of function Θ_y (for the same values of x and y), this fact enables us to conclude that u is more sensitive to changes in y than in x.

Now, rearranging the equation (6.1),

$$\Theta_x = \left|\frac{x}{u}\right| \cdot \left|\frac{\Delta u}{\Delta x}\right|$$

It is well-known that when the variation Δx tends to 0, then the term $\left|\frac{\Delta u}{\Delta x}\right|$ tends to the partial derivative $\left|\frac{\partial u}{\partial x}\right|$. That is,

$$\lim_{\Delta x \to 0} \left|\frac{\Delta u}{\Delta x}\right| = \left|\frac{\partial u}{\partial x}\right|$$

Then, the expression for sensitivity is given by

$$\Theta_x = \left|\frac{x}{u} \cdot \frac{\partial u}{\partial x}\right| \tag{6.2}$$

Equation (6.2) will be utilized in the following sections to analyse the sensitivity of the parameters involved in the estimated cost of development C_{dev}.

6.3 Sensitivity of the Cost of Development

To clarify the sensitivity aspect of formula (3.13) for the parameters involved, we calculate the partial derivative of C_{dev}. For each parameter a brief discussion is given on the influence that changes in its value would exert on C_{dev}, and a comparison of sensitivity among parameters is presented. The module with 4000 lines of code is used for illustration, considering the values employed in the example of Chapter 5.

From formula (3.13) (repeated below) we see P_{dev}, E_{cod}, γ, N and s need to be estimated. As analysed in Chapter 3, the parameters N and s are estimated based on the parameters β, F and δ. So, we select the following parameters for our sensitivity analysis:

- P_{dev} (page 44); E_{cod} (page 45); γ (page 60); β (page 49); F (page 53); and δ (page 59).

$C_{dev} = P_{dev} E_{cod} \left(1 + \gamma \frac{e^{Ns}-1}{e^{Fs}-1}\right)$ (equation 3.13)

6.3.1 Sensitivity due to P_{dev} and E_{cod}

Let the sensitivities due to P_{dev} and E_{cod} be denoted, respectively, by $\Theta_{P_{dev}}$ and $\Theta_{E_{cod}}$. Using equation (6.2) the following results are obtained.

- $\Theta_{P_{dev}}$

  As can be seen from equation (3.13), C_{dev} varies linearly and in direct proportion to P_{dev}, which is confirmed below.

$$\Theta_{P_{dev}} = \left| \frac{P_{dev}}{C_{dev}} \cdot \frac{\partial C_{dev}}{\partial P_{dev}} \right|$$

$$\frac{\partial C_{dev}}{\partial P_{dev}} = E_{cod} \left(1 + \gamma \frac{e^{Ns}-1}{e^{Fs}-1}\right)$$

So,

$$\Theta_{P_{dev}} = \left| \frac{P_{dev}}{C_{dev}} \cdot E_{cod} \left(1 + \gamma \frac{e^{Ns}-1}{e^{Fs}-1}\right) \right|$$

Substituting the expression for C_{dev} obtained from equation (3.13), it is immediately found that

$$\Theta_{P_{dev}} = 1$$

- $\Theta_{E_{cod}}$

$$\frac{\partial C_{dev}}{\partial E_{cod}} = P_{dev}\left(1 + \gamma\frac{e^{N_s}-1}{e^{F_s}-1}\right)$$

Then, similarly, we have

$$\Theta_{E_{cod}} = 1$$

Thus, it can be concluded that variations in P_{dev} and E_{cod} produce the same proportional variation in C_{dev}. So, C_{dev} has the same sensitivity for those two parameters.

6.3.2 Sensitivity due to γ

Let Θ_γ denote the sensitivity of C_{dev} due to γ. From equation (6.2) we have that,

$$\Theta_\gamma = \left|\frac{\gamma}{C_{dev}} \cdot \frac{\partial C_{dev}}{\partial \gamma}\right|$$

$$\Theta_\gamma = \frac{\gamma}{C_{dev}} \cdot P_{dev} \cdot E_{cod} \cdot \frac{e^{N_s}-1}{e^{F_s}-1}$$

$$\Theta_\gamma = \frac{\gamma(e^{N_s}-1)}{(e^{F_s}-1) + \gamma(e^{N_s}-1)}$$

$$\Theta_\gamma = \frac{\gamma}{\gamma + \frac{e^{F_s}-1}{e^{N_s}-1}} = \frac{1}{1 + \frac{e^{F_s}-1}{\gamma(e^{N_s}-1)}} \tag{6.3}$$

γ	C_{dev}	Θ_γ
0.50	13945.46	0.193
0.75	15618.19	0.303
1.00	17290.91	0.387
2.00	23981.83	0.586
2.25	25654.56	0.617

Table 6.1: Example of C_{dev} for different γ

In Table 6.1 are several examples which illustrate the behaviour of C_{dev} as γ varies. These examples consider the same values utilized in the example of Chapter 5, for a module with 4000 lines of code, with a required reliability $R_{req} = 0.9$.

To obtain the results presented in Table 6.1 we need to set a reasonable range for γ. It can be found in the footnote on page 60 that a typical range for γ can be $0.4 < \gamma < 2.5$. In the example analysed in Chapter 5 we have used $\gamma = 1.0$. So, from this range, we select five values for γ, which, in our opinion, may serve to illustrate the influence of γ on C_{dev}. In Table 6.1 we then have:

- Value for $\gamma = 0.5$, which is 50% of the value utilized in the example of Chapter 5, and greater than 0.4.
- Value for $\gamma = 0.75$, which is a medium point between the value above for γ and that utilized in Chapter 5;
- Value for $\gamma = 1.0$, as in the example of Chapter 5.
- Value for $\gamma = 2.0$, which is twice the value in that example, and less than 2.5.
- Value for $\gamma = 2.25$, which is slightly smaller than the maximum value (that is, 2.5).

From Table 6.1 it can be seen that Θ_γ is always less than 1.0. Thus, it can be concluded that C_{dev} appears to be less sensitive to γ than to P_{dev} and E_{cod}. A comparison with other parameters is presented subsequently.

If γ changes, C_{dev} varies considerably, as expected, but not in the same proportion to γ, e.g., a 2-fold increase in γ results, in this example, only in a 1.39-fold increase in C_{dev}.

6.3.3 Sensitivity due to β

The parameter β is used in formula (3.3). As can be seen in that formula (3.3), β affects N, which, in turn, also affects s (formula (3.9)). And both exert influence on C_{dev}.

The sensitivity of C_{dev} with respect to β, from equation (6.2), is

$$\Theta_\beta = \left| \frac{\beta}{C_{dev}} \cdot \frac{\partial C_{dev}}{\partial \beta} \right| \tag{6.4}$$

The result for $\frac{\partial C_{dev}}{\partial \beta}$ is a rather complicated expression. An expression for $\frac{\partial C_{dev}}{\partial \beta}$ was obtained using the software MATHEMATICA [87]. When used in equation (6.4) and employing the same data as the example in Chapter 5, we have the results shown in Table 6.2.

To obtain the results shown in Table 6.2 we needed to select a suitable range of values for β, in order to illustrate the influence of β on C_{dev}.

Suppose that for the 4000-lines-of-code module, a reliability of 0.90 is required. We know that $N = F - \frac{\ln R}{\ln(1-\beta)}$ (formula 3.3).

Since N cannot be a negative number, and assuming that N is always different from zero, we have $\frac{\ln R}{\ln(1-\beta)} < F$. Rearranging this expression we have that $\beta > 1 - \sqrt[F]{R}$.

For the values utilized in the example of Chapter 5, and using the expression

β	N	C_{dev}	Θ_β
0.002	62	12814.76	0.475
0.0025	72	14073.62	0.413
0.005	93	17290.91	0.205
0.01	104	19193.32	0.098
0.05	112	20643.84	0.018

Table 6.2: Examples of C_{dev} for different β

above, we have $\beta > 0.001$. So, the range for β may be set[1] as $1.0 \geq \beta > 0.001$. We select to use five representative values of β (Table 6.2):

- $\beta = 0.005$ is employed in the example of Chapter 5;
- $\beta = 0.002$ is just greater than the minimum, i.e., 0.001;
- $\beta = 0.01$ and $\beta = 0.0025$ represent, respectively, twice and half of the value utilized in the example of Chapter 5. These values will be utilized for comparison among parameters.
- $\beta = 0.05$ constitutes an artificially high value for β. As discussed in Chapter 3, this value of β means that for 100 executions of the module, on average 5 failures will occur;

As can be seen in Tables 6.1 and 6.2, $\Theta_\gamma > \Theta_\beta$, when we utilize the same variation for both parameters.

If β decreases, the number of faults to be removed to achieve a required reliability decreases gradually. So, if it is the case, for smaller values of β, the flat area in figure

[1]Recall that the value of β represents a probability, therefore $1.0 \geq \beta \geq 0$.

5.4 would be larger than that using $\beta = 0.005$, because with just 62 out of 115 faults removed the reliability would be 0.90. Thus, considering smaller values for β, the cost of development would be cheaper for a larger values of required reliability (because less faults will need to be removed in order to achieve R).

As $R = 1.0$ corresponds to the maximum cost, we then have (for smaller values of β) that the exponential curve for C_{dev} plotted against R_{req} would rise up sharply, for $0.90 < R_{req} < 1.0$.

By contrast, as β increases, the probability that a remaining fault will produce a failure increases, and there is a more gradual effect of changes in β on N, and consequently on C_{dev}.

Summing up: the value of β affects the number of faults to be fixed in a marked way, which, in turn, influences C_{dev} as well, but *not* in the same proportion.

6.3.4 Sensitivity due to F

This parameter has a direct effect on N, as can be seen in formula (3.3). Here, the sensitivity of C_{dev} with respect to F is given by

$$\Theta_F = \left| \frac{F}{C_{dev}} \cdot \frac{\partial C_{dev}}{\partial F} \right| \tag{6.5}$$

The software **MATHEMATICA** was again consulted to find $\frac{\partial C_{dev}}{\partial F}$.

Based on the examples shown in [38], we consider a range from half the estimated F using formula (3.4) to double the latter. So, we have in Table 6.3 five scenarios which illustrate the behaviour of C_{dev} as F varies. They are:

- $F = 57$ represents approximately 50% of the value utilized in the example of Chapter 5.

- $F = 76$ represents approximately 2/3 of the value utilized in the example.

- $F = 115$ is utilized in Chapter 5.
- $F = 172$ is 50% higher than the value utilized in Chapter 5.
- $F = 230$ is twice the value utilized in Chapter 5.

F	C_{dev}	Θ_F
57	13846.26	0.509
76	15464.03	0.322
115	17290.91	0.206
172	18543.59	0.133
230	19198.74	0.098

Table 6.3: Example of C_{dev} for different F

In Tables 6.2 and 6.3, it is seen that C_{dev} is more sensitive to F than to β.

6.3.5 Sensitivity due to δ

In formula (3.13) it can be seen that C_{dev} varies inversely to the parameter s. That is, if s increases the expression $(e^{Fs} - 1)$ will increase more rapidly than $(e^{Ns} - 1)$, which means that C_{dev} will decrease. As s varies in the same direction as δ (formula (3.9)), then it can be concluded that C_{dev} varies inversely to δ. The sensitivity of C_{dev} with respect to δ is

$$\Theta_\delta = \left| \frac{\delta}{C_{dev}} \cdot \frac{\partial C_{dev}}{\partial \delta} \right| \tag{6.6}$$

To verify the above conclusion, the software **MATHEMATICA** was again employed to find an expression for $\frac{\partial C_{dev}}{\partial \delta}$.

In the footnote on page 59 it was noted that a typical range for δ is $1 < \delta \leq 10$. In the example of Chapter 5 we use $\delta = 5.0$. Based on this range for δ, we choose here five values to illustrate the typical influence of δ on C_{dev} (see Table 6.4).

- $\delta = 1.5$ is just greater than the minimum (that is, 1.0).
- $\delta = 2.5$, half of the value used in the example of Chapter 5.
- $\delta = 5.0$, just as in that example.
- $\delta = 7.5$ is a medium point between the value utilized in Chapter 5 and the maximum value (that is, 10.0).
- $\delta = 10.0$, double the value utilized in that example.

δ	C_{dev}	Θ_δ
1.5	18736.43	0.061
2.5	18141.23	0.064
5.0	17290.91	0.069
7.5	16791.99	0.070
10.0	16444.47	0.071

Table 6.4: Example of C_{dev} for different δ

Comparing the value of Θ_δ to the previously analysed parameters, it may be concluded that δ affects C_{dev} the least.

Therefore, a large error in the estimate of δ would result in considerably *less* error in the estimate of C_{dev}, in the opposite direction. Thus, it can be said that C_{dev} is not markedly sensitive to δ.

6.3.6 Summary of the sensitivity analyses

The sensitivity of C_{dev} with respect to the parameters P_{dev}, E_{cod}, γ, β, F and δ, has been examined. Some typical values of these sensitivity factors were obtained, and some quantitative analysis has been done. By doing so, we are able to gain an indication of how much influence an inaccurate estimate for one of these parameters might exert on C_{dev}.

Tables 6.5, 6.6 and 6.7 display not only numerical results, but reflect a classification for level of influence for the parameters analysed. This classification is represented in each table by four levels. Each row in the table corresponds to a level. The levels are:

- Level 1: minimal influence ($0 < \Theta \leq 0.1$).
- Level 2: moderate influence ($0.1 < \Theta \leq 0.5$).
- Level 3: steady influence ($0.5 < \Theta \leq 1.0$).
- Level 4: large influence ($1.0 < \Theta$).

Table 6.5 shows the minimum values found for the sensitivities; Table 6.6 shows the sensitivity for the values used in the example of Chapter 5; and Table 6.7 shows the maximum sensitivities obtained.

Analysing Tables 6.5, 6.6 and 6.7, it may be suggested that C_{dev} is not ill-conditioned and is not excessively sensitive to any of the parameters involved.

	P_{dev}	E_{cod}	δ	β	F	γ
1			0.061	0.018	0.098	
2						0.193
3	1.0	1.0				
4						

Table 6.5: Summary of the sensitivity analysis: "minimum" values

	P_{dev}	E_{cod}	δ	β	F	γ
1			0.069			
2				0.205	0.206	0.387
3	1.0	1.0				
4						

Table 6.6: Summary of the sensitivity analysis: "median" values

	P_{dev}	E_{cod}	δ	β	F	γ
1			0.071			
2				0.475	0.509	0.617
3	1.0	1.0				
4						

Table 6.7: Summary of the sensitivity analysis: "maximum" values

It can also be said that,

- There is no parameter classified as exerting "large influence" on C_{dev}.
- An inaccurate estimate for δ does not affect C_{dev} very much. This conclusion, as discussed previously, enabled us to use a constant value of δ throughout the examples shown in Chapter 5.
- Variations in P_{dev} and E_{cod} produce the same (linear) proportional variation in C_{dev}. Thus, inaccurate estimates for these parameters will affect C_{dev} directly in the same proportion.
- The remaining parameters exert an influence, but *not* dangerously high, on C_{dev}.
- Based on Tables 6.5, 6.6 and 6.7, it may be suggested that γ, β and F exert (*roughly*) the same influence on C_{dev}, and δ exerts the least influence.

A more rigorous and complete sensitivity analysis may be possible if the parameters β, γ and δ can be expressed as functions of even more basic parameters (to be figured out) that exist in a software development process (a case in point might be software size). In such a case, we may be able to learn more about the interaction among the parameters and the behaviour of C_{dev} as these parameters vary. Thus, we may be able to establish, for example, what would be the sensitivity of C_{dev} due to software size.

It also has to be said that a combination of changes in different parameters may result in changes in C_{dev}, which is not easy to predict. Therefore, the classification indicated in Tables 6.5, 6.6 and 6.7 should be used just as a preliminary guideline to the sensitivity of C_{dev} with respect to the parameters utilized.

6.4 Sensitivity of the Overall Reliability

To construct scenarios between cost and reliability, as analysed in Chapter 5, equation (4.1), which estimates the overall system reliability, is employed. This formula, which is repeated below, contains the coefficients P_{ij} (see page 70) and P_{iT} of the transition matrix (see page 71) which must be predicted.

$R_{est} = \sum_{i=1}^{n} W_{1i} R_i P_{iT}$ (equation 4.1)

where $W = X^{-1}$ with $X_{ij} = \begin{cases} 1 & i = j \\ -R_i P_{ij} & i \neq j \end{cases}$

$i = 1, \cdots, n$ (number of modules in the modular system under estimation).

$\sum_{j=1}^{m}(P_{ij} + P_{iT}) = 1.$

The two parameters sets of P_{ij} and P_{iT} may be analysed to determine how the overall estimated system reliability R_{est} is affected by changes in their values. In this way the sensitivity of R_{est} with respect to changes in P_{ij} and P_{iT} might be assessed.

However, as can be seen, the general expression for R_{est} depends on the number of modules under consideration and how they are linked. Thus, it seems to be infeasible to define the sensitivities to P_{ij} and P_{iT} without knowing a closed form for the expression of R_{est}. Furthermore, we should bear in mind that changes in P_{iT} result in changes in P_{ij} as well. Moreover, any change in P_{ij} or P_{iT} results in changes to other coefficients in the same row and column, and, consequently, in the entire matrix. As analysed in Chapter 4, $\sum_{j=1}^{n} P_{ij} + P_{iT} = 1$.

Hence, we consider it unworkable to develop expressions for the sensitivity of the overall system reliability with respect to the transition matrix.

Chapter 7

Conclusion

The main issue dealt with in this work is how to estimate the cost of developing a modular software system, during the early design phase of software development, taking into account a required level of reliability for the system.

By examining some relevant software cost estimation models, which are applied during the design phase, we confirmed that these models do not usually treat reliability requirements as a cost driver. Even when they do, no explicit figure for the required reliability is utilised in their formulation. The treatment of a reliability requirement is invariably broad rather than precise, yielding an outcome which can only roughly indicate the influence of reliability in software cost estimation, and which certainly does not constitute any step towards supporting an effective trade-off between cost against reliablity during the design stage. Software size, which can be argued to be the foremost factor in relation to the software cost, is also discussed.

Thus, it may be concluded that current software cost estimation models available in the literature, either on their own or through software sizing models, do not provide any special approach in dealing with a trade-off between cost against reliability before the coding phase.

As is outlined in Chapter 2, a real discussion on the trade-off between cost against reliability can be established during the testing phase by means of software release policies. However, the input data for these policies can only be obtained during the testing phase, which constitutes a clear impediment for their utilization in the earlier phases of the life-cycle of software development, such as at the design phase.

Therefore, it seems to be worth considering whether software release policies, might be adjusted, or their concepts utilized, so that we may apply them, combined with software estimation models, to obtain the desired trade-off during the design phase. The model developed in this work is a step in this direction.

The line of argument developed here is that the reliability of a software module is closely linked to the effort spent during the testing phase, meaning that a higher level of desired reliability requires more testing effort and, consequently, will cost more. On this basis, a straightforward decomposition technique is used to estimate the cost of development, based on the number of faults which will have to be found and fixed to achieve a required reliability, using data obtained from the requirement specification and historical data. The model proposed was developed and investigated solely on the basis of hypothetical data, guided (where available) by published data values. Figure 7.1 represents the basic chain of relationships that is utilized to link cost and reliability (although only a few of the parameters actually appear in figure 7.1).

It is well known that a high reliability requirement means that a software system will need more time and cost for its development. For this reason it is assumed that the minimum acceptable value for the overall reliability of the software is known in advance. On this basis, this thesis elaborates a proposal for allocating reliability levels to individual modules of a software system; a formula was obtained that allows us to calculate the overall system reliability using Markov analysis.

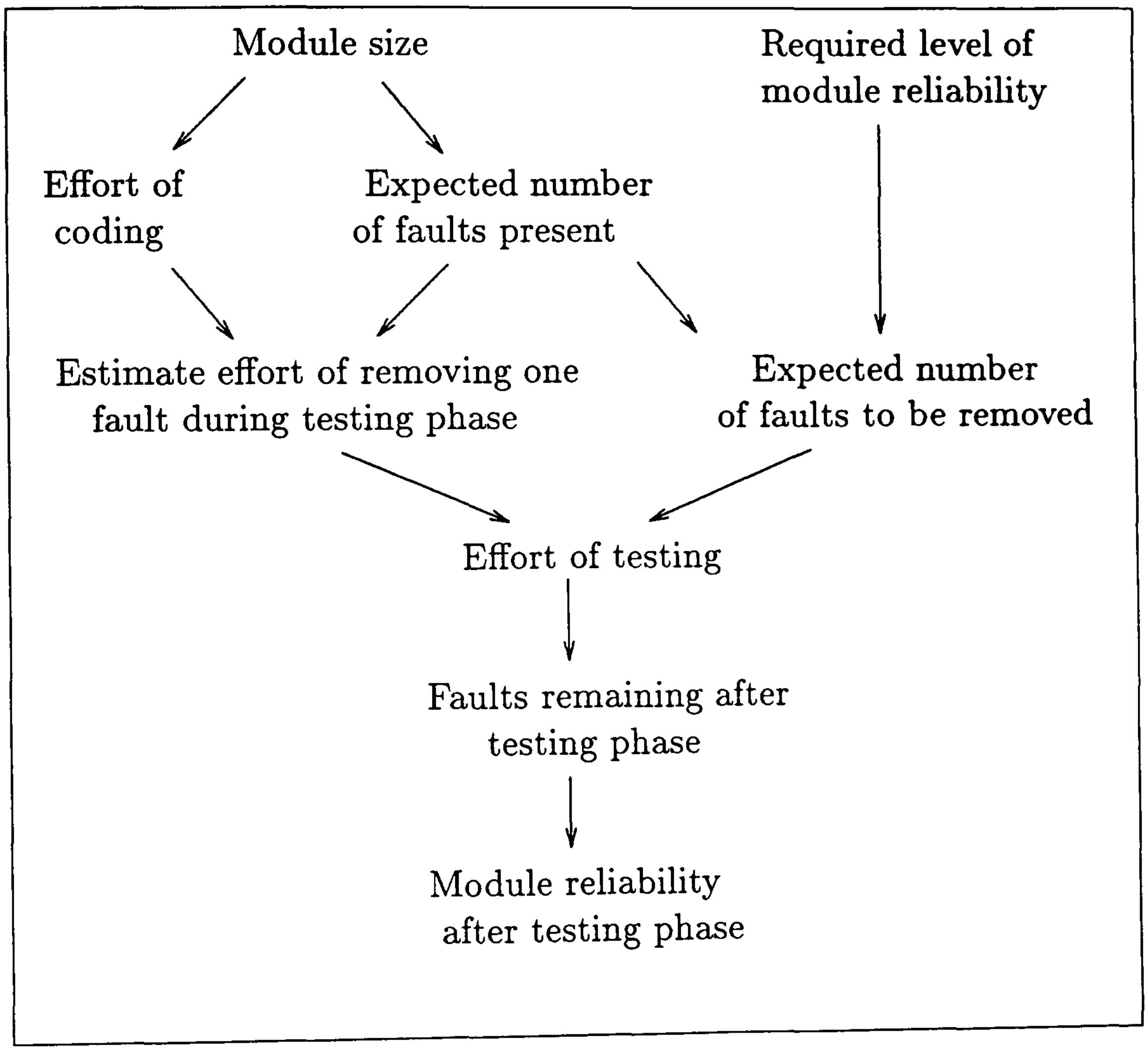
Module size
Required level of module reliability
Effort of coding
Expected number of faults present
Estimate effort of removing one fault during testing phase
Expected number of faults to be removed
Effort of testing
Faults remaining after testing phase
Module reliability after testing phase

Figure 7.1: Basic relationships that link cost and reliability

From that formula, and using a standard minimization approach, a reliability level for each module can be selected to ensure that the overall system cost is minimal.

A point that it is worth highlighting is that if reuse is employed in the software development, we can see this fact as a bonus. In this situation it may be suggested that the number of faults to be removed would be smaller (than normal software development without reuse), consequently, the software would cost less.

In line with recent published papers [33, 36], which assert that project managers wish to utilize simple methods for estimating development cost, this work has proposed an uncomplicated method to estimate the cost of coding and testing a software system. This estimate is based on data available at the design phase, i.e., before beginning the coding phase, and, as the main contribution of our work, taking into account a required level of system reliability.

Some formulas were derived that enable a project manager to obtain the required estimates in a relatively straighforward way. The results obtained show a reasonable behaviour for the method proposed. Despite being still very preliminary, the outcome of this work allows us to say that the method proposed here is clearly a step forward in formulating a trade-off between cost and reliability during the design phase.

Necessary sensitivity analysis has been carried out in order to evaluate the behaviour of the proposed formula for the development cost, with respect to variations in the several parameters involved. By means of this analysis, we have been able to verify that this formula is not ill-conditioned and the sensitivities found are numerically within acceptable limits, in all our examples.

From this research a number of open problems arise for further investigation. We have used several parameters in our formulae, and some of these parameters we have assumed to be known. In reality, that may not be *exactly* the case. An important research item would then be to develop appropriate expressions for those parameters, in terms of even *more basic* parameters present in the software development process. Furthermore, our model needs to be evaluated and, if possible, refined, based on long-term data collected in a real software development environment.

So, the following themes are suggested for future research:

- Analysis of the parameters introduced in C_{dev}

  The parameters β (page 49), δ (page 59) and γ (page 60) utilized in C_{dev} need to be studied in order to find a suitable expression for them (again, in terms of more basic parameters, for example, software size). For the parameters P_{dev}, E_{cod}, F and S, if there exist data of previous projects (using similar characteristics of development in the installation under consideration), we may use the methods already available in the literature for estimating them, as discussed earlier.

  It can be argued that these parameters can be estimated using data from projects with similar characteristics of development. Thus, to enable a project manager to take full advantage of the method developed in this work, the following points should be considered:

  ⋆ a) How can "similar characteristics of development" be characterized?

  ⋆ b) How can data best be gleaned to obtain a sound estimate of each parameter?

  ⋆ c) Which analysis techniques should be used to yield the best estimate for each parameter?

  ⋆ d) Can a software measurement and analysis procedure be established so that these parameters may be applicable to a different environment by an ordinary project manager?

Finding the answers to the above questions will definitely provide very challenging research work[1].

- Contructing a transition matrix

  As presented in Chapter 4, it is required to assess available (or propose new) methods for constructing a transition matrix; such a matrix is based on a hierarchical view of a system, and obtained using the software engineering techniques employed during the design phase of development.

  Such work could refine the methods for constructing the transition matrix, in the early stage of design phase, such that the estimated probabilities of transition between modules can as accurately as possible be assigned (some preliminary ideas are presented in Chapter 4).

- Validation of the method developed

  There should be a validation procedure of the method developed here. The trade-off model, as discussed, was developed using hypothetical data in its procedures. Some topics could be dealt with in this validation:

  - ⋆ Analyse how the proposed trade-off model shapes up in a real software development environment, that is, compare the results yielded by the proposed model with those of the real one.
  - ⋆ Reassessment of the several assumptions made throughout this work, which may enable one to verify that the assumptions work properly or need to be adjusted.

[1][12, 16, 67] provide a thorough treatment of software metrics, containing an extensive bibliography in this subject, which may serve as a starting-point for this research.

In relation to this validation procedure and the proposed model, there is a point that might be emphasized. This is the fact that at the end of the day the results we want involve (i) allocating resources among different parts of a given project and (ii) estimating the additional resources needed to achieve reliability results above a given base line. That is, they involve relations among things rather than the things themselves. The point about this is that even if the assumptions made about the things are occasionally inaccurate, those assumptions apply everywhere and are probably inaccurate to a similar extent everywhere, so the relations could well be accurate.

Even if the numbers are slightly imprecise, the relationships need not be, and could well be better than the educated guesses people use at present (though this would have to be checked). So while it would be preferable if absolute quantities could be measured in some way, we can still have enough information to make good managerial decisions even if the estimates are slightly inaccurate, provided they are applied consistently.

The above points remain to be seen during the procedure of validation.

- Another view of the cost of development

  Consider figure 5.4 (the graph of development cost against reliability). This is a plot of equation 3.13 for three sets of values of parameters.

  If a whole set of curves was produced for various sets of values (specifically values that appear to make sense in practice) then it may be possible to use a curve fitting algorithm as a way of finding a possibly simpler characterisation of the essential behaviour of the curves. This issue may be dealt with as follows.

  Suppose that, after examining the cost versus reliability curves like those in figure 5.4, an executive decision is made that they are of the form

$$C_{dev} = a + b \cdot e^{c \cdot R} \tag{7.1}$$

where,

- C_{dev} is the cost of development; a, b and c are constants; and R is the required reliability.

Then, to find out the curve in any particular case, the three constants a, b and c need to be determined. That is, it is sufficient to know the values of C_{dev} at three different points R (e.g., the start-up cost if the reliability is equal to zero, which gives us $a + b$).

These values may be derived from partial observations of the real system development. So, the suggested work cited above could be complemented with this and so may enable one to obtain a simpler expression for cost of development using data from real software development.

With the suggested points for development, we have an interesting and applied research field in this subject, it can be claimed.

Considering the method developed in this thesis, we would claim a project manager could deal with various scenarios of estimated total cost and overall required reliability before allocating the resources for the coding and testing phases, which could lead to better management of the software project as a whole.

Bibliography

[1] Albrecht, A.J. and Gaffney, J. Software function, source lines of code, and development effort prediction: a software science validation *IEEE Transactions on Software Engineering*, SE-9(6):639-648, November 1983

[2] Ashrafi, N. and Zahedi, F. Software reliability allocation based on structure, utility, price and cost *IEEE Transactions on Software Engineering*, 17(4):345-356, April 1991

[3] Bailey, J.W. and Basili, V.R. A meta model for software development resource expenditures *Proceedings of the Fifth International Conference on Software Engineering*, pp 107-116, 9-12 March, 1981, San Diego, California

[4] Banker, R.D and Kemerer, C.F Scale economies in new software, *IEEE Transactions on Software Engineering*, 15(10):1199-1204, 1989

[5] Berman, O. and Ashrafi, N. Optimization models for reliability of modular software systems *IEEE Transactions on Software Engineering*, 19(11):1119-1123, November 1993

[6] Bittanti, S., Bolzern, P. and Scattolini, R. An introduction to software reliability modelling *Lectures Notes in Computer Science*, N.341:43-67, October 1988

[7] Boehm, B.W. **Software engineering economics** Prentice Hall, 1981

[8] Brown, D. **A method for obtaining software reliability measures during development** *IEEE Transactions on Reliability*, R-36(5):573-580, December 1987

[9] Burnett, R. and Anderson, T. **Reliability allocation for a system with modular structure** *Proceedings of the VIII Brazilian Symposium of Software Engineering*, pp 37-48, 25-29 October, 1994, Curitiba, Brazil

[10] Cheung, R. **A user-oriented software reliability model** *IEEE Transactions on Software Engineering*, SE-6(2):118-125, March 1980

[11] Cohen, B. **The specification of complex systems** Addison-Wesley 1986

[12] Conte, S.D., Dunsmore, H.E. and Shen, V.Y. **Software engineering metrics and models** The Benjamin/Cumming Publishing Company, 1986

[13] Corbett, M. and Kirakowski, J. **An analogical approach to cost estimation** *Proceedings of European Software Cost Modelling Meeting 1992*, (without page numbering), 27-29 May, 1992, Munich

[14] Cox, D.R. and Miller, H.D. **Theory of stochastic processes** Methuen, 1965

[15] DeMarco, T. **Structured analysis and system specification** Prentice-Hall 1979

[16] Fenton, N. **Software metrics: a rigorous approach** Chapman Hall, 1991

[17] Fickas, S. and Nagaraju, P. **Critiquing software specification** *IEEE Software*, 5(6):37-47, November 1988

[18] Fox, L. **An introduction to numerical linear algebra** Monographs on Numerical Analysis, Oxford Science Publications, 1964

[19] Gaffney, J.E. **Estimating the number of faults in code** *IEEE Transactions on Software Engineering*, SE-10(4):459-464, July 1984

[20] Gill, P.E., Murray, W. and Wright, M.H. **Practical optimization**, Academic Press, 1981

[21] Goel, A. and Okumoto, K. **Time-dependent error-detection rate model for software reliability and other performance measures** *IEEE Transactions on Reliability*, R-28(3):207-211, August 1979

[22] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A. and Trakhtenbrot, M. **STATEMATE: A working environment for the development of complex reactive systems** *IEEE Transactions on Software Engineering*, 16(4):403-414, April 1990

[23] Heemstra, F.J. and Kusters, R. **Software cost estimation and control: lessons learned** *Proceedings of the European Software Cost Modelling* 1992, (without page numbering), 27-29 May, 1992, Munich

[24] Hihn, J. and Habib-agahi, H. **Cost estimation of software intensive projects: a survey of current practices** *Proceedings of 13th International Conference on Software Engineering*, pp 276-287, 13-17 May, 1991, Austin, Texas, USA

[25] Jeffery, D.R, Low, G.C. and Barnes, M. **A comparison of function point counting techniques** *IEEE Transactions on Software Engineering*, 19(5):529-532, May 1993

[26] Jeffrey, A. **Mathematics for engineers and scientists**, Chapman-Hall, 1992

[27] Keremer, C.F. **An empirical validation of software cost estimation models** *Communication of ACM*, 30(5):416-429, May 1987

[28] Keremer, C.F. and Porter, B.S. **Improving the reliability of function point measurement: an empirical study** *IEEE Transaction on Software Engineering*, 18(11):1011-1024, November 1992

[29] Keremer, C.F. **Reliability of function points measurement: a field experiment** *Communications of ACM*, 36(2):85-97, February 1993

[30] Khoshgoftaar, T.M., Bhattacharyya, B.B. and Richardson, G.D. **Predicting software errors, during development, using nonlinear regression models: a comparative study** *IEEE Transactions on Reliability*, 41(3):390-395, September 1992

[31] Kitchenham, B. and Taylor, N.R. **Software project development cost estimation** *Journal of Systems and Software*, 5:267-278, May 1985

[32] Kitchenham, B. **Software development models** in *Software Reliability Handbook (ed. Paul Rook)*, Centre for Software Reliability, Elsevier Applied Science, 1990

[33] Kitchenham, B. **Empirical studies of assumptions that underlie software cost-estimation models** *Information and Software Technology*, 34(4):211-218, April 1992

[34] Kubat, P. and Koch, H. **Optimal release time of computer software** *IEEE Transactions on Software Engineering*, SE-9(3):323-327, May 1983

[35] Kubat, P. Assessing reliability of modular software *Operation Research Letters*, 8(1):35-41, February 1989

[36] Lederer, A.L. and Prasad, J. Nine management guidelines for better cost estimating *Communications of ACM*, 35(2):51-59, February 1992

[37] Leung, Yiu-Wing. Optimum software release time with a given budget *Journal of System and Software*, 17:233-242, 1992

[38] Lipow, M. Number of faults per line of code *IEEE Transactions on Software Engineering*, SE-8(4):437-439, July 1982

[39] Lipow, M. Comments on "Estimating the number of faults in code" and two corrections to published data *IEEE Transactions on Software Engineering*, SE-12(4):584-585, April 1986

[40] Littlewood, B. Software reliability model for modular program structure *IEEE Transactions on Reliability*, R-28(3):241-246, August 1979

[41] Littlewood, B. Forecasting software reliability *Lecture Notes in Computing Science*, No.341, Software Reliability Modelling and Identification (ed. Bittanti, S.), pp 141-209, Springer-Verlag, 1987

[42] Littlewood, B. Modelling growth in software reliability in *Software Reliability Handbook (ed. Rook, P.)*, pp 137-154, Elsevier Science Publishers Ltd, 1990

[43] Londeix, B. Cost estimation for software development Addison-Wesley Publishing Company, 1987

[44] Maghsoodloo, S., Brown, D. and Deason,W.H. **A cost model for determining the optimal number of software test cases** *IEEE Transactions on Software Engineering* 15(2):218-221, February 1989

[45] Masuda, Y., Myiawaki, N., Sumita, U. and Yokoyama, S. **A statistical approach for determining release time of software system with modular structure** *IEEE Transactions on Reliability*, 38(3):365-372, August 1989

[46] Matson, J.E., Barret, B.E. and Mellichamp, J.M. **Software development cost estimation using function points** *IEEE Transactions on Software Engineering*, 20(4):275-287, April 1994

[47] McCabe, T.J., John Jr, F.C., Adams, K.A. and Sturgill, A.M. **Structured real-time analysis and design** *Proceedings of COMPSAC85*, pp 40-52, 9-11 October, 1985, Chicago, Illinois, USA

[48] Mills, H.D. and Dyson, P.B. **Using metrics to quantify development** *IEEE Software*, 7(2):14-16, March 1990

[49] Mukhopadhyay, T. and Kekre, S. **Software error models for early estimation of process control applications** *IEEE Transactions on Software Engineering*, 18(10):915-924, October 1992

[50] Musa, J., Iannino, A. and Okumoto, K. **Software reliability: measurement, prediction, application** McGraw-Hill, 1987

[51] Musa, J. and Ackerman, A.F. **Quantifying software validation: when to stop testing?** *IEEE Software*, 6(3):19-27, May 1989

[52] The Numerical Algorithms Group Limited **NAG Fortran Library, MARK 14** (1st Edition), Oxford, 1990

[53] Okumoto, K. and Goel, A.L. Optimum release time for software systems based on reliability and cost criteria *The Journal of Systems and Software* 1:315-318, January 1980

[54] Ottenstein, L. Quantitative estimates of debugging requirements *IEEE Transactions on Software Engineering*, SE-5(5):504-514, September 1979

[55] Pressman, R.S. Software engineering: a practitioner's approach McGraw Hill, 1992

[56] Pucci, G. On the modelling and testing of recovery block structures *IEEE Transactions on Software Enginering*, SE-18(2):159-167, February 1992

[57] Putnam, L.H. A general empirical solution to the macro software sizing and estimating problem *IEEE Transactions on Software Engineering* SE-4:345-361, July 1978

[58] Putnam, L.H. and Fitzsimmons, A. Estimating software costs, *Datamation*, September 1979 (Part I), 25(10):188-198; October 1979 (Part II), 25(11):171-178; November 1979 (Part III), 25(12):137-140, 1979

[59] Ratcliff, B. and Rollo, A.L. Adapting function point analysis to Jackson system development *Software Engineering Journal*, 15(1):79-84, January 1990

[60] Reifer, D.J. Asset-R: a function point sizing tool for sientific and real-time systems *Journal of System Software*, Vol.11, pp 159-171, 1990

[61] Riet, R.W.N. The mermaid project *Proceedings of European Software Cost Modelling 1992*, (without page numbering), 27-29 May, 1992, Munich

[62] Rook, P. **Tutorial:software sizing techniques** Seminar IN: *UK Regional Meetings of European COCOMO User's Group*, (without page numbering), 22nd January 1993, Milton Keynes, England

[63] Sarper, H. **No special schemes are needed for solving software reliability optimization models** *IEEE Transactions on Software Engineering*, 21(8):701-702, August 1995

[64] Schreiber, B.M. and Zwegers, A.J.R. **Software cost data collection** *Proceedings of European Software Cost Modelling Meeting 1993*, (without page numbering), 22-24 March, 1993, Bristol

[65] SEI **Software size measurement, with applications to source statement counting** (draft for review) Size Subgroup of the Software Metrics Definition Working Group *Software Engineering Institute (SEI), Carnegie Mellon* University USA, August 1991

[66] Shepperd, M. and Ince, D. **Derivation and validation of software metrics** Oxford Science Publications, 1993

[67] Shepperd, M. **Foundations of software measurement** Prentice Hall, 1995

[68] Shooman, M.L. **Software engineering** McGraw Hill, 1983

[69] Siegrist, K. **Reliability of systems with Markov transfer of control** *IEEE Transactions on Software Engineering*, SE-14(7):1049-1053, July 1988

[70] Siegrist, K. **Reliability of systems with Markov transfer of control, II** *IEEE Transactions on Software Engineering*, SE-14(10):1478-1480, October 1988

[71] Stetter, F. **Comments on "Number of faults per line of code"** *IEEE Transactions on Software Engineering*, SE-12(12):1145-1145, December 1986

[72] Stutzke, R.D. **Size estimation: helping the neophyte** in *Proceedings of European Cost Modelling Meeting 1992*, (without page numbering), 27-29 May, 1992, Munich

[73] Symons, C.R. **Function point analysis: difficulties and improvements** *IEEE Transactions on Software Engineering*, 14(1):2-11, January 1988

[74] Symons, C.R. **Software sizing and estimation:MK II FPA** Wiley Series in Software Engineering Practice, John Wiley and Sons, Inc., New York 1991

[75] Takahashi, M. and Kamayachi, Y. **An empirical study of a model for program error prediction** *IEEE Transactions on Software Engineering*, 15(1):82-86, January 1989

[76] Trachtenberg, M. **The linear software reliability model and uniform testing** *IEEE Transactions on Reliability*, R-34(1):8-16, April 1985

[77] Trivedi, K.S. **Probability and statistics with reliability, queuing and computer science applications** Prentice Hall 1982

[78] Verner, J. and Tate, G. **A software size model** *IEEE Transaction on Software Engineering*, 18(4):265-278 April 1992

[79] Yamada, S and Osaki, S. **Cost-reliability optimal release policies for software systems** *IEEE Transactions on Reliability*, R-34(5):422-424, December 1985

[80] Yamada, S. and Osaki, S. Optimal software release policies with simultaneous cost and reliability requirements *European Journal of Operational Research*, 31:46-51, 1987

[81] Yourdon, E.N. Modern structural analysis Prentice-Hall 1990

[82] Yun, W. and Bai, D.S. Optimum software release policy with random life cycle *IEEE Transactions on Reliability*, 39(2):167-170, June 1990

[83] Xie, M. On the determination of optimum software release time *Proceedings 1991 International Symposium on Software Reliability Engineering*, pp 218-224, 17-18 May, 1991, Austin, Texas

[84] Wallace, D. R. and Fujii, R. U. Software verification and validation: an overview, *IEEE Software*, 6(3):11-17, May 1989

[85] Whittaker, J. Markov chain techniques for software testing and reliability analysis *PhD Thesis*, The University of Tennessee, Knoxville, May 1992

[86] Whittaker, J. and Poore, J. Markov analysis of software specifications *ACM Transactions on Software Engineering and Methodology*, 2(1):93-106, January 1993

[87] Wolfram, S. MATHEMATICA – A system for doing mathematics by computer, Addison-Wesley Publishing Company, 1988