3516

738773

2054072

TR 3516

**Value of Information
Based
Design of Control Software**

Jan Gerhard Norstrøm

**Value of Information
Based
Design of Control Software**

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus, prof. ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,

op maandag 22 mei 2000 te 10.30 uur

door

Jan Gerhard NORSTRØM

Master of Science in the faculty of Mathematics and Physics
The Norwegian University of Science and Technology

geboren te Tønsberg, Norway

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. R.M. Cooke

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter

Prof. dr. R.M. Cooke, Technische Universiteit Delft, promotor

Dr. T.J. Bedford, Technische Universiteit Delft, toegevoegd promotor

Prof. dr. F.M. Dekking, Technische Universiteit Delft

Prof. dr. ir. J. van Katwijk, Technische Universiteit Delft

Prof. dr. ir. J. Hellendoorn, Technische Universiteit Delft

Prof. dr. S. French, Manchester Business School, The University of Manchester

Prof. P. Hokstad, NTNU/SINTEF Industrial Management, Trondheim

Published and distributed by:

Delft University Press

P.O. Box 98

2600 MG Delft

The Netherlands

Telephone: +31 15 2783254

Telefax: +31 15 2781661

E-mail: DUP@Library.TUdelft.NL

ISBN 90-407-2062-2

Copyright 2000 by J.G. Norstrøm.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilised in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the publisher: Delft University Press.

Cover design by Marian Seiergren

Printed in The Netherlands

Acknowledgements

I am grateful to my supervisors Prof. Roger M. Cooke and Dr. Timothy J. Bedford for giving me the opportunity to carry out my Ph.D. research thesis at the Delft University of Technology. I could never have dreamed of more enthusiastic and challenging supervisors than Roger and Tim. Our round table discussions taught me the techniques of mathematical research. I would also like to emphasise the supportive friendship that I share with them both.

There have been many people who have helped me during my time here. I would like to thank Prof. Simon French for advising me on literature about optimal stopping problems. I am grateful to Floor Koornneef for many fruitful discussions on software quality assurance. To Etienne de Klerk and Erling D. Andersen for sharing some ideas on numerical integration and to Valery Kritchallo for discussing and fixing computer problems. A special thanks to Marian Seiergren for designing the cover illustration of this thesis.

On a personal level I would like to highlight the international atmosphere that I was a part of in my daily life. It was special and I am grateful to all whom I share such a friendship. I would also like to thank my squash partners Petteri Nieminen, Edward Melger, Etienne de Klerk and Knut S. Lidsheim for all the competitive, but enjoyable matches that we played.

Finally my Father Anders G. Norstrøm, my mother Tove Norstrøm and my sister Anne Margrethe who supported me unwaveringly during these years and taught me to never give up. Then, last but not least, my son Jon whom I missed during this period. To them I dedicate this work.

Contents

CHAPTER 1	1
INTRODUCTION	1
CHAPTER 2	5
SOFTWARE RELIABILITY MODELS	5
2.1 SOFTWARE ERRORS AND SOFTWARE RELIABILITY	6
APPLICATION-SOFTWARE EXAMPLE: A PERSONNEL DATABASE.....	6
CONTROL-SOFTWARE EXAMPLE: SATELLITE SOFTWARE.....	7
2.1.1 Software Reliability Concepts	9
2.1.2 Software Failure Visualisation	10
2.1.3 Software Error Mechanisms.....	12
Software code generation errors	12
Specification errors	12
Complex errors.....	13
2.2 SOFTWARE QUALITY ASSURANCE AND SOFTWARE RELIABILITY	13
2.2.1 Software quality metrics	13
2.2.2 Software Quality Assurance Systems.....	14
2.2.3 Management Philosophies.....	16
2.2.4 The Software Safety Lifecycle.....	16
2.3 QUASI-FORMAL RELIABILITY TECHNIQUES	18
2.3.1 Languages.....	18
2.3.2 Development of Software - Structured & Modular Programming	18
2.3.3 Fault tolerance	19
2.3.4 FMEA & FMECA's of software-based systems	21
2.3.5 Formal Technical Reviews	22
2.3.6 Software Sneak Analysis.....	22
2.3.7 Software Testing.....	23
2.3.8 Reporting of Problems.....	24
2.4 SOFTWARE RELIABILITY PREDICTION MODELS	24
TIME-DOMAIN MODELS	25
FAULT SEEDING AND DATA DOMAIN MODELS.....	26
MARKOV MODELS.....	27
AXIOMATIC MODELS	28
2.4.1 The Jelinsky and Moranda (JM)-model.....	29
2.4.2 The Schick - Wolverton model.....	31
2.4.3 The Geometrical De-Eutrophication Process	31
2.4.4 A Flexible Modelling Approach	31
2.4.5 The Bayesian JM-model	33
2.4.6 The Littlewood (L)-model - Faults with different occurrence rates.....	36
2.4.7 The Littlewood and Verrall (LV)-model	38
2.4.8 Keiller and Littlewood (KL)-model	39
2.4.9 Weibull Order Statistics.....	39
2.4.10 Non-homogenous Poisson process models.....	40
The exponential intensity model	41
The logarithmic expectation model	42
The Duane (D) Model	42
2.4.11 The Nelson model - The Basic Data Domain model.....	43
2.4.12 A Bayesian Data Domain Model.....	43
2.4.13 The Hypergeometric Data Domain Model.....	44

2.4.14 Markov-models	45
Reliability of a software system	46
Availability and maintainability of a software system	47
2.4.15 Reliability and Debugging Data	48
2.5 DEVELOPMENT PHASES AND RELIABILITY MODELS	49
2.6 CONCLUSIONS	50
2.6.1 Reliability Concepts	51
2.6.2 Software Reliability Practise	52
2.6.3 Reliability Models – What do they Measure	52
2.6.4 Future Control Software Reliability Modelling	54
2.7 REFERENCES	55
CHAPTER 3.....	65
DECISION THEORETIC BACKGROUND.....	65
3.1 SOME PROPERTIES OF EVENTS AND OBSERVATIONS.....	67
3.2 SAVAGE'S DECISION THEORY	74
3.2.1 Quantitative Probability	77
3.2.2 Utility	78
3.2.3 Observation and The Value of Information	79
3.3 DECISION TREES AND INFLUENCE DIAGRAMS	81
3.4 REFERENCES	83
CHAPTER 4.....	85
VALUE OF INFORMATION AND SCHEDULING OF TESTS	85
Set-up cost	85
Running cost of performing a test:	86
The problem of designing control system	86
4.1 THE CHOICE TO DO AN OBSERVATION	87
4.1.1 Formalism	92
4.1.2 Graphical Representation - Choice Tree	100
4.2 SEQUENTIAL TESTING.....	102
4.2.1 Parallels to Savage	112
4.2.2 The Optimal Sequence And Its Computation Complexity	112
4.2.3 Sequential Sampling and Conditional Independence	114
4.2.4 Exchangeability and Additive-Cost	115
4.3 GREEDY ALGORITHMS AND THE OPTIMAL TEST ORDER	126
Greedy - Counter Example	134
CONSTRUCTION OF GREEDY EXAMPLE.....	136
NOTATION	147
4.4 REFERENCES	149
CHAPTER 5.....	151
OPTIMAL INSPECTION OF FAULT-TREES	151
5.1 INTRODUCTION	151
5.2 USEFUL APPROXIMATIONS, PROBABILITIES AND ERROR-BOUNDS	154
5.3 VALUE OF INFORMATION BASED INSPECTIONS STRATEGY.....	177
5.3.1 Inspections Sequenced by a Greedy Algorithm	180
Simplifying gives	183
5.3.2 Test-Schedule Sorting	185
5.4 WANG'S ALGORITHM AND FUSSELL-VESELY IMPORTANCE APPROACHES.....	188

5.4.1 <i>Traditional Inspection Strategy</i>	188
5.4.2 <i>Wang's Algorithm</i>	189
5.5 CONCLUSION	191
5.6 REFERENCES.....	192
CHAPTER 6.....	195
SATELLITE CONTROL-SOFTWARE.....	195
6.1 DETAILED CONTROL SOFTWARE DESCRIPTION	196
6.1.1 <i>Required Control-Software Decisions</i>	198
6.2 FORMAL DECISION PROBLEM FORMULATION.....	199
6.2.1 <i>Making observations</i>	202
ONE BREAK WIRE INSPECTION	202
6.2.2 <i>The Set of Available Acts</i>	203
6.2.3 <i>Consequence Assessment & Utility Definition</i>	204
6.2.4 <i>Parameter Specification</i>	204
6.2.5 <i>The Value of The Break Wire Test</i>	205
6.2.6 <i>The Optimal Control-Flow</i>	207
6.2.7 <i>Working Out the Probabilities</i>	208
6.3 REFERENCES.....	211
INDEX	213
SAMENVATTING	215
CURRICULUM VITAE	219

Chapter 1

Introduction

Sneak path analysis was my first contact with software reliability. It is used to find design errors in software and hardware. The method seeks to find all paths between critical functions and their initiating commands. When all paths have been identified they are compared with the system requirement specifications. Each mismatch is investigated to see how the consequences affect the system with respect to safety, reliability and system performance. The method does not allow one to question the underlying architecture or specifications. It would, for example, be impractical to use the method to evaluate various design solutions. The method does not enable one to question whether the flowchart itself is well designed. Indeed no standard software design method contains tools allowing the designer to decide on an optimal ordering of sensor inspection. When I started my PhD work at TU Delft, it was decided to work on the area of designing control software by using techniques from decision theory.

First the software reliability literature, summarised in Chapter 2, was searched to see whether it contained something on evaluation of different design options in the context of sensor monitoring and ordering. Since reliability and safety activities are often organised under software quality assurance in industry this was a natural place to start. I found that the software reliability models mostly considered the software as a black box. The literature concentrates on the concepts of decreasing failure rate or decreasing failure probability where failure data is obtained from the debugging process or via life experience. The software performance increases as faults are removed after detection. The black box simplification implies that none of the methods question the software architecture. Therefore these models can not explain how control software should make observations or read sensors. The models can however be used to decide the release time for a “new” software program. Chapter 2 does not go further than the existing literature and was written to understand software reliability engineering. It concludes that the issue of control-software architecture have not been studied in the context of software reliability.

To provide the mathematical basis for further research, Savage’s decision theory was studied as background material. His profound theory formalises the notion of the rational decision-maker. This is briefly summarised in Chapter 3. During this work two areas of interest were selected. The first one relates to the use of influence diagrams and decision trees. The second describes how sensors should obtain probabilistic information. It was found that an influence diagram is a good way of modelling symmetric decision trees. The graphical representation was however impractical and insufficient to model the problem of architecture optimisation. The reason is that it is not possible to model the impact of a change in the ordering of

tests/observations without redrawing the influence diagram and the decision tree. This research gave ideas and a solid background for further research in Chapter 4. The other objective was to understand how sensors should be designed to obtain probabilistic information. Self conditional expectations were studied to consider the probability of an event A given the outcome of a random variable T , $p(A | T = t)$, to find out if there are ways in which a sensor can be designed so that the expectation of the conditional probability changes. This work is original and some of the results here can be taken as arguments to say that sensors should be designed to obtain probabilistic information on a finer partition of events. Indeed, probabilistic considerations of control software would involve this kind of conditional probability.

In Chapter 4 the main goal is to find a decision theoretic way to represent the choice the software has when it chooses between "to do" an observation and "not to do" an observation. New notation was introduced to show clearly the possible choices in scheduling tests. A simple test-schedule example is " $A \rightarrow B$ " which means "Do A and then choose whether to do B ". For optimisation purposes it was important that the model enabled a rational selection of tests based on expected utility. When the concept of a test-schedule had been defined it was possible to formulate many new results. The theorems and their proofs can not be found elsewhere in literature and are original. It was shown that the test-sequence is important i.e. $A \rightarrow B$ gives different expected utility than $B \rightarrow A$. Hence the expected utility depends on the decision problem, the test-schedule and the test-schedule order. From the initial work on influence diagrams and decision trees in Chapter 3, a new and compact graphical representation Choice-Tree was created to show the choice between "to do" and "not to do" a test. Our theory is related to De Groot's theory of optimal stopping. The link is, however, not obvious right away. The theory on optimal stopping problems uses statistical assumptions on conditionally independence of random variables, the test results, given a prior distribution. It also assumes that each test has the same performance cost. These are strong mathematical assumptions. Given both assumptions it is proved that our problem simplifies to optimal stopping problems studied in the literature.

The confirmation that the expected utility depends on the test order in Chapter 4 required research on heuristic methods to find an optimal test-order for a test-schedule. In Chapter 5 we apply the theory established in Chapter 4 in a different setting namely to the problem finding fault detection algorithms. The Fussel-Vesely importance measure is often recommended to rank components for inspection priority in fault detection algorithms. Fussel-Vesely importance was studied together with other heuristic concepts like "Greedy" which is described in Chapter 4. The main objective was to find out how well these heuristic strategies work. Our findings show that these strategies give bad performance in the expected number of tests required to find the failure cause. The performance is up to 50% worse than the best achieved test order obtained via exhaustive search. Hence, it can be argued that it is wrong to recommend the use of Fussel-Vesely importance measure for fault detection. Since the results differed so much we tried to study why the heuristic approaches like Fussel-Vesely and "Greedy" performed so badly. It was however impossible to find a simple explanation and it was easier to research a new method. Hence, a test-schedule sort algorithm was developed to reduce the expected number of inspections required to find the fault. In the examples that were studied, test-schedule sort quickly improved the test order so that the expected number of inspections was near "optimum".

Chapter 6 returns to the problem where our research started. We consider a small system that involves satellite control-software and show how software architecture can be enhanced by using decision theory. Since the model developed in Chapter 4 was successfully used in Chapter 5 we needed to demonstrate that decision theory could have practical application in the design of control software architecture. This was the original idea when the research on decision theory started. In practical software engineering and in particular, in the design of control software the use of voting logic is a popular way to take decisions under uncertainty. For example, satellite control software may decide to deploy an antenna boom if 3 out of 5 break wires test broken. However, this heuristic is not the optimal way of deciding under uncertainty, and can produce very suboptimal software designs. After using decision theory on a simplified problem where one test is made we found the optimal time to read a sensor. Given various sensor readings the optimal deployment time changed. This information can be directly used when the control-software architecture is designed and expressed in a flowchart.

Chapter 2

Software Reliability Models

This chapter¹ provides an introduction to basic software reliability models. It is our goal to cover the literature that introduces reliability models and to understand their main assumptions. The first models arrived around 1970. The later models are small variations on the earlier models. Therefore we stress earlier models more than later models. The reader should also be aware that this is a young science and consensus and consistency in notation and concepts are still evolving.

In many large organisations software reliability is ordered under activities related to quality assurance. Software quality assurance is an “umbrella” that also covers software reliability. Quality assurance reflects the development process. It is the process of monitoring the development and coding of software to improve both the software-product and efficiency of making it. Quality assurance is monitored via characteristics of the product. Reliability can be one such characteristic. For example “reliable software” can be one of the goals for a quality assurance policy. Therefore to give a better overall understanding of the field we will briefly consider quality assurance before we consider software reliability models in detail.

Software reliability concepts are covered in Section 2.1, software quality assurance and software reliability in Section 2.2 and quasi-formal reliability techniques in Section 2.3. These topics are briefly covered while software reliability prediction models are covered critically in mathematical detail in Section 2.4. Section 2.5 summarises some thoughts around the use of software reliability models. Section 2.6 concludes with a small discussion.

Many papers covering the topic of software reliability have been published since 1970. In the seventies data, from debugging of software was collected by e.g. Moranda [45]. The various attempts to model these data formed the origin of software reliability. The reliability book by Villemeur [3] contains 125 different references. The Centre for Software Reliability at City University in London maintains a WWW-page (<http://www.csr.city.ac.uk/>) that references important publications. Recent books that covers the current state of the art in software reliability is the handbook of Lyu [1] and Musa’s [23] book on software engineering.

The importance of software reliability has been recognised by the European Commission, and in 1994 it was decided to sponsor the ENCRESS project. ENCRESS (European Network of Clubs for Reliability and Safety of Software) has

---

¹ I am grateful to Floor Koornneef for the discussions we shared about this Chapter. Floor Koornneef is member of the European Workshop on Industrial Computer Systems (EWICS).

grown to be a special interest group for everyone who shares a professional interest in the safety and reliability of software-intensive systems. Now, ENCRESS has been succeeded by ISA-EUNET that stands for Intensive Software Systems for Safety Applications, a high-tech European lean Network. ISA-EUNET is for example involved as a co-sponsor of the SafeComp 2000 conference. Annually there is a SafeComp conference and it serves the purpose of reviewing state of the art experiences and new trends in computer safety, reliability and security.

2.1 Software Errors and Software Reliability

Software is a word that covers a wide variety of products and systems. It is meaningless in itself since it could cover anything from operating systems to avionics in an aeroplane. Even in word processors today most people create software code.

The technology today allows us redefine or solve many problems in terms of software. For example in a modern car mechanical machinery has been removed and replaced with a microchip that contains some lines of a software code. Now we will give two examples based on the author's experience that uses software in two different ways. The examples illustrate the problem of software faults in the context of defining failures.

APPLICATION-SOFTWARE EXAMPLE: A PERSONNEL DATABASE

Think of an army-personnel database with a central data unit that stores all information concerning personnel in a database. The database contains all kinds of information about the personnel. There are usually lots of users logged into the database, and they have different objectives. Some users are only interested in information concerning personnel that have got a certain rank while others retrieve information about personnel stationed at a certain military station and do not want or should not have access to other personnel information. In order to meet such various user-needs, several applications are programmed, all with different user requirements in mind.

The programming is usually carried out so that one programmer is responsible for programming one particular user application according to some requirements of what the user needs to have in order to do his work.

Programming errors and faults in the code occur. The system has failure tolerance since faults in the code do not cause the entire system to go off-line during normal working hours. This means that the fault only affects people that are using one faulty application. While the fault is being fixed and after re-initialisation, the user is often able to apply the program to do some other work without entering the area of the application that caused problems.

Fault detection is done in the way that the programmers can interface with the user to find the input set that caused the failure. When the fault is discovered the programmer can start fixing it in a copy of the program running in a simulated environment. Thus, the entire system is not off-line while the fault is being fixed. After having fixed the fault the programmer tests that the main functions are working. The code is compiled and during the night copied onto the real environment where the user can continue his work in a new version of the program the next day.

Some questions can be raised here. What is software failure in the environment described above? Is it obvious for the army (in this case), that a fault in the software code is a failure? ...Probably not.

One could argue that the organisation has created an environment that is "*failure tolerant*" in the sense that it can cope with failures of the kind mentioned above. However, imagine that we change the impact of failure above so that it affected the user(s) so they could not perform their work for days and fail to do their job or lose critical data. A response of the organisation could conclude that this was a particular severe failure and a nasty fault. What we actually have seen is an example of how useful it can be to classify failures based on their severity. Some failures can be tolerated if they can be handled without problems. Note that when a system is able to handle failures it is often considered to be failure tolerant.

■

CONTROL-SOFTWARE EXAMPLE: SATELLITE SOFTWARE

Before launch antennas, solar-arrays and booms carrying instruments are folded to fit the spacecraft inside the cargo-bay of the launcher. To ensure that such equipment remains folded during the launch it is bolted to the main body of the spacecraft. When the spacecraft is separated from the launcher, it is particularly vital that the antennas are deployed to establish ground contact. Regardless of the mechanism that force the antennas to open (a spinning spacecraft can use centrifugal force), the bolts holding the antennas must be cut-off to release the antennas. This is frequently solved by the use of pyrotechnic bolts. Pyrotechnic bolts are cut-off by firing an explosive device from inside the bolt.

The process of deploying the antenna is performed by control software. For this system to operate, it needs to detect when the space-craft is separated from the launcher (free flying) to initiate safe deployment of the antenna. It needs to observe when a pyrotechnic device has been activated so it can time the firing of other pyrotechnic devices correctly (Note that devices are usually fired one by one to protect spacecraft structure from the shock impulse). A similar system is described in Dore & Norstrom [28].

To speak of such a system only in terms of software reliability makes no sense since the hardware alone could cause the software to be unable to operate as expected. For instance, if the software is **only** programmed to read the sensors that should detect separation (see Figure 1) the program is never going to finalise successfully if the sensors fail to transmit that separation has taken place.

Figure 1. Simplified control-flow diagram.

What happens in a situation where the program does not receive the correct "separation detected"-signal? Does it lead to infinite software loop failure or is it a hardware failure? It may also be considered to be both. This example is very simplified, however it illuminates the difficulties with the concepts of software failure and software reliability in a control software system.

This raises some other questions. At what level should the reliability for this system be evaluated, what is the system under review/control and what is the mission of the system? Should the system be evaluated at subsystem level containing both software and hardware, or should it be evaluated as one hardware system and one software system? One could even go further and ask if it makes sense to speak about software faults without seeing the software as a part of the subsystem containing both hardware and software. In other words without seeing the software in the context of the system mission and its environment. Intuitively one should look at a system from an overall point of view and consider its mission. It is however easy to forget or lose the overview and this seems especially difficult when the system is large and complex. These aspects are frequently not thought of when theorists try to fit software reliability data to a software reliability model without considering application context (see Section 2.4 for software reliability models). It is my impression that the software reliability models and the data being used do not sufficiently cover the application context.

■

The two examples above illustrate that the usage of software in different environments greatly impacts on what we think of a software fault in terms of failure. For example in the database example one would be willing to accept faults in the software code as long as the failure severity is at an adequate level. However, even the smallest fault could be considered not acceptable in the control-software example. Lalli & Malec [2] attempt to solve this problem. For example they divide in-house and vendor acquired software into different categories: Embedded software, applications software etc. However these definitions do not take account of the importance of failure tolerance. For example embedded software is software

embedded in physical products such as radar controllers, avionics and Xerox-machines. It is easy to imagine that software embedded in an aeroplane has completely different acceptance levels with respect to failures and software faults compared to a Xerox-machine. For example the user of a Xerox machine can be willing to accept certain paper jam-failures regardless being caused by software, hardware or the user. An aeroplane passenger would for instance not accept that his plane crashes for whatever reason.

2.1.1 Software Reliability Concepts

If we think about the two examples in the introduction to Section 2.1 it is clear that there is no simple definition of software failure or software reliability. It is not sufficient only to look at typing errors in the code. A search in literature confirms this and software failures and faults have a much broader concept. Villemeur [3] lists several of the concepts that exists or that have been used. However, in order to have a consistent and sufficient framework we will use the concepts that Musa [23] uses:

- *Failure*: The departure of system behaviour in execution from user requirements or needs.

A software failure is a user-oriented concept. Note that the system must operate for a failure to occur. Note that a software failure can also be caused by hardware failure. If the sensors fail in the control software example in Figure 1 it could cause infinite software loop failure. The fault that causes this could be missing code to protect against infinite loop failure. Note that the problem of hardware and software failure interaction forms an area of study frequently called hardware/software dependability. Lalli & Malec [2] discuss literature that describes hardware and software failures in the telecommunications field attempting to clarify the differences and to develop a combined reliability approach.

- *Fault*: The defect in the program that when executed under partial conditions cause a failure.

A fault is a developer oriented concept since it by the developer can be regarded as the cause for failures. For a specific fault different sets of conditions can cause various failures. Hence, a fault can cause more than one failure. Note that a fault can also be missing instructions in the code. Faults can be detected without having a failure for example during a code review. There exist several fault-detection methods that can be applied (see Section 2.3). Note that a fault does not necessarily lead to a failure. It depends on the circumstances that the software operates under. Assume for example that it is required that infinite loop failure should not be allowed in the control software example above. Then the control software in Figure 1 has the fault that code is missing to protect against infinite loop if the sensors fail. This fault **only** causes failure if the program is executing when the sensors are failed in a condition such that the infinite loop is possible. In other words faults may or may not lead to failure depending on the system mission and the environment. Hence, it can be faults in the code that never causes a failure.

- *Error*: An incorrect or missing action by a person or persons that causes a fault in a program.

It follows that software errors are caused by human “mistakes” during software development. Error mechanisms and causes are discussed in Section 2.1.3.

- *Software reliability*: The ability of the software to operate without a failure.

A measure of software reliability is the probability that the software will operate without a failure for a given period of time and under stated conditions.

In software quality assurance similar definitions of failure and fault are also applied by Fenton et al [4]. We will return to quality assurance in Section 2.2.

It may not be technically or economically feasible to eliminate all software faults. See for example Davis and Giloth [82] that consider software in the telecommunications area. If one can not eliminate all faults, it would imply that some faults are likely to cause failures. On the other hand, a user may accept some faults in the program if the failure probability is low and if the failure severity is acceptable. It is up to the requirements to specify to what extent failures are accepted. Just as important as reliability is severity classification since failures normally differ in their impact on the operations and the consequences for an organisation varies. Musa [23] uses the following severity classification system:

Class 1: Basic service interruption.

Class 2: Basic service degradation.

Class 3: Inconvenience, correction not deferrable.

Class 4: Minor tolerable effects, correction deferrable.

If we apply the severity classification to the personnel database example in Section 2.1 above one might define the failure that was detected to be of class 3 since correction was not deferrable. Non correctable destruction of data in the database can for example be considered to be of class 1 while correctable degradation of the database could be class 2. An unclear user interface could be graded as a class 4 failure. It is the authors impression that when software reliability models as in Section 2.4 are discussed in literature they are evaluated after how well they can model data sets while the severity of the failures in the data sets are offered sparse consideration. This is important since some organisations would for example only like to know the probability of a Class 1 failure. Severity classification is also difficult. Consider the example in Figure 1 and assume that there is a fault in the control software that can cause a separation detection failure when triggered. What is a basic service interruption for the control software? The failure can be considered a basic service interruption if deploy antenna is not triggered if separated. However, if the operation is assessed at system level and it is apparent that there are other implementations elsewhere in the design that influence the separation, the severity of the failure would depend on those implementations. For example there could be a function that triggers a separation mechanism that alters separation parameters. Thus even if the software fails for certain data inputs, the data can be changed over time by other procedures so that "deploy antenna" is triggered. If this occurs, then we rather speak of a degraded service since it can take more time before separation is detected.

2.1.2 Software Failure Visualisation

Littlewood [8] uses the simple visualisation in Figure 2 to describe software failure.

Figure 2. Mapping of the program execution with failure input/output regions I_f , O_f .

The program P can be interpreted as a function $P : I \rightarrow O$ that maps the input domain I to the output domain O . In this model a failure of the program is experienced only if an input in the subset I_f is given and the program creates the faulty output $P : I_f \rightarrow O_f$. Note that the model only concerns the software program P . The software program P is treated as a "black box" since behaviour only is evaluated from its inputs and outputs. The model only interacts with other hardware or software via inputs and outputs. This is a simplification of reality where the program P may run in parallel with other software and hardware that can influence the performance of P . For example a program running in parallel can change the value of a parameter used by P while it is running. Problems caused by interaction with O_f by concurrent software or hardware is not handled.

The model gives an intuitive understanding of reliability growth when a fault is being fixed. Then it is assumed that the program P is changed such that the input set I_f becomes smaller i.e. some elements in I_f that created a faulty output in O_f is changed to a non-faulty input that gives a non-faulty output. Uncertain aspects in this model are:

- It can be difficult to specify the input region completely. For large software systems it is practically impossible to define the input region I . Hence, the output domain O is considered to be uncertain. It is unlikely that one is able to specify and design a program that does not fail when the input region is not defined.
- Different systems have different input regions. The input region of for example bank-transfer software is different compared to e.g. control software of a scientific satellite.
- The environment may also change over time so the input region changes and has a new failure domain I_f (the program does not consider the 'new' region correctly). This can happen when clever users find new tasks where they can apply existing software that was originally designed for another purpose. For example the Ariane 5 accident was caused by some code that originally were developed for the Ariane 4 rocket (see Inquiry Board Report [16]). The failure was caused by faster vertical acceleration of the Ariane 5 that was beyond the software design needed to

successfully launch the Ariane 4. Thus, the environment for the Ariane 4 software was changed when used in Ariane 5.

- There can be program-related uncertainties that are introduced when for instance a fault is removed. A fault is removed if the program is changed for input $i \in I_f$ such that the output o of O given i is not faulty. This process may however introduce new faults so that other values of I change to give an output in O_f .

Littlewood [8] argues that these uncertain aspects should enable statistical modelling of software reliability.

2.1.3 Software Error Mechanisms

We will briefly go through some error mechanisms that are known to create faults that can cause software failure if they are executed during system operation. These mechanisms are due to design errors created by humans during the development of the software. According to Musa [23] most of the causes can be grouped into one of the categories: Communication, knowledge, incomplete analysis and transcription. Communication errors arise between the various parties involved in a software design project. For example miscommunication between the customer and the designer could cause errors. The software project personnel could have a lack of experience and knowledge about the application area, the design methodology and the programming language that could cause errors. If the system is incompletely analysed so that the designer has not the full overview of possible conditions that can occur at a given point in the program an error can be created. Transcription errors are errors that can occur between the mind and paper or machine.

Several errors can cause a fault and several faults may arise from one error. When the error processes above cause software failure they are often different than the nature of mechanisms that leads to failures of for example mechanical components. Mechanical components can fail due to ageing or wearing while software does not age nor wear. Such differences between software and mechanical components are discussed by for example O'Connor [12]. Two processes that are known to cause errors are software code generation and software specification. They are briefly discussed below.

Software code generation errors

Code generation is the prime source of errors, since it is a creative process where the programmer can make mistakes covering logical errors and typographical errors. Code generation is also the last phase where faults can be detected before code reviews, testing and acceptance (afterwards the code is compiled). Note that the compiler also can introduce faults during compilation due to failures during compilation. The failures may not be flagged and are therefore hidden in the compiled program. Typical faults that can arise are: Typographical faults, compiler seeded faults, incorrect numerical values, omission of symbols, e.g. parentheses, inclusion of expressions which can become indeterminate such as division by zero, references to non existing variables or code.

Specification errors

According to O'Connor [12] typically more than half of the faults recorded during software development originate in the specification phase. Specifications can be wrong, wrongly perceived, incomplete or inconsistent. For example according to the

specifications an input variable x is required to be less than the value 100 while the user actually wanted the value 10. Faults in the code caused by miss-specification are often not detected by the programmer when he generates the software code since the nature of the fault is different than the typographical faults that are normally introduced at this stage. Note that this is the reason why specification errors often are considered to be more serious than software code generation errors.

Complex errors

As we grow better at removing specification errors and software code errors, the less obvious and more complex errors as design concept errors are revealed. Such errors can be sneak paths that are only seen if rare and special conditions trigger them e.g. see Dore & Norström [28]. These errors can manifest themselves as hidden loops, sneak paths and dead code (intended code that can never be executed) in the software program. To handle such errors during development would require another design process and other non standard error and fault detection methods as for example Sneak Analysis e.g. see Section 2.3.6.

2.2 Software Quality Assurance and Software Reliability

Software quality assurance is the activity that leads to a software quality product. A quality software product is characterised by how well it fits to the user requirements and expectations. A quality software product should also be reliable. It should be able to perform a required function up to a required reliability level. Software quality assurance is therefore often thought of as being “the most efficient way” to improve software reliability. Quality assurance is the process of monitoring the production, technology and personnel performance so predefined requirements are satisfied by the software. It also implies learning from the past so that it allows increased and more efficient production without loss of quality or with quality improvement in the software. In other words the quality of the product is measured and the information is used to improve the process of producing it. For an introduction to software quality assurance see Villemeur [3] and Lalli & Malec [2]. Fenton et al [4] explain software quality assurance and measurement as tools in management philosophies. Pressman [13] discusses software quality assurance as an element in software engineering. We will first give an introduction to how quality is measured in Section 2.2.1 before we describes what a software quality system should contain in Section 2.2.2.

2.2.1 Software quality metrics

Fenton et al [4] emphasise the importance of applying statistics to measure software quality. For example a quality characteristic for software is reliability (note that quality characteristic also is called quality factor or attribute). A different characteristic is freedom for faults in the source code. The quality characteristic is measured by a software metric. For example a software metric that measures the characteristic reliability is probability of no failure (see Section 2.1.1). Further information on metrics can be found in the book by Gilb [5] and the guidelines produced by the European Workshop on Industrial Computer systems in Redmill [6]. A software metric for freedom of faults in the source code is the number of faults pr 1000 lines delivered source code. An example given by Fenton et al [4] compare lines of code delivered per person year and faults per 1000 lines of code for a company in

USA and a company in Japan. In the US-company the lines of source code per person year of work was 7290 while in the Japanese company it was 12 447. The number of faults per 1000 lines of code was 4.44 for the US-company and 1.96 for the Japanese one. Clearly one is confident that the Japanese company did better. One problem with this comparison is the cause for the difference. As pointed out by Fenton et al [4] it could reflect cultural differences as well as different management quality assurance philosophies. However, the measurement gives one a common way to communicate and question the difference. In order to discuss software quality with respect to changes in management policy, new technology etc. one needs to measure the results objectively. This is emphasised both by Lalli & Malec [2] and by Fenton et al [4].

Some other typical software characteristics are:

- *Maintainability*: The effort required to locate and fix a fault in a program. Note that when measured in time it does not necessarily influence the system downtime as for hardware since maintenance can be performed on copies of running software.
- *Portability*: The effort required to transfer the program from one hardware or software system environment to another.
- *Testability*: The effort required to test a program to ensure that it executes an intended or required function.
- *Usability*: The effort required to learn, operate, prepare input, and interpret output of a program.
- *Flexibility*: The effort required to modify an operational program.

For a more extensive list see Lalli & Malec [2] and Pressman [13]. Gilb [5] gives general rules for how to specify characteristics. In a software project it is important to define or choose which characteristics that are important for the particular product that is being made. It is often not simple to find software metrics that quantifies the chosen characteristics. For example it is much more difficult to quantify flexibility than for example freedom of faults in the source code as discussed above. Often best engineering practice is used. Pressman [13] shows two different forms that can be applied to use software quality characteristics. Note that this is a difficult area since two different persons due to subjective opinion can apply the engineering practice differently and draw conflicting conclusions.

There exist formal methods to derive metric strategies. One such method is the goal question metric method in Fenton et al [4]. The method targets the reasons why a metric should be applied. Any metric extracted must serve some purpose. Details are beyond the scope here.

2.2.2 Software Quality Assurance Systems

Quality assurance is not something that is achieved by each employer in a company. It is a system that for example a company or an organisation has where each individual plays a precise role in the organisation to create a quality product. In other words it has to be implemented so that managerial structure, responsibilities, activities, capabilities, and resources are defined. The goal is to ensure software projects/products so that selected quality characteristic objectives are fulfilled. The quality management system will contain activities as:

- *Auditing of projects:* In an audit of a project one evaluate quality assurance activities to see whether they have been carried out correctly. For example if it was decided to use a tool to enforce programming standards, auditing staff should be able to verify that it has been used correctly.
- *The review of the Quality System:* A quality system needs continuous improvements. New technologies should be adopted and current approaches should be sharpened based on experience from past projects. For example if the programming language is changed from conventional to object oriented programming the quality manuals in the organisation must adopt the change.
- *Career development of staff:* It is important to develop quality assurance staff so they do not become technically out of date.
- *Negotiation of resources:* The organisation needs to have a financial plan to do quality assurance. This is negotiated with the management and/or the involved parties in a project.
- *The development of standards and procedures:* The standards detail how documents such as the requirement specification should be laid out on paper. The procedures detail how activities as testing or code reviews should be executed.
- *Reporting:* High-level management should be informed about efficiency of the current quality system and opportunities to improve.

The quality management system is described in the quality manual to the organisation. A quality manual contains (internal) standards, procedures, and guidelines and is influenced by external standards. Lists of external applicable software standards are given in Fenton et al [4] and Pressman [13]. An important international standard is for example IEC 61508 [14]. An internal *standard* could for example specify certain tests to be performed and how many successful trials are needed before the program is accepted. This is one area where software metric as in Section 2.2.1 can be used to set quantitative goals for the organisation. A *procedure* would for example detail how to perform the tests in the standards. Note that international standards might also contain procedures.

Creation of quality standards and procedures is an important task in a quality system. It is described by Lalli & Malec [2] as a continuous process where the project influences the quality standards and procedures as seen in Figure 3. Again the standards and procedures impacts quality measurements and characteristics.

Figure 3: Illustration of the process of updating and creating the quality standards.

In Section 2.3 we will briefly list some procedures that can be considered as quasi formal reliability techniques since they remove faults from the software.

2.2.3 Management Philosophies

An aspect of software quality assurance is motivation. The way management chooses to handle software reliability and quality assurance issues is also reflected by the people working with these issues. Total quality management is one management philosophy that was proposed for hardware manufacturing. More recently it has been modified to be useful for software quality assurance as well (see Fenton et al [4]). Total quality management (TQM) is an umbrella term for a philosophy that emphasises shared responsibility for quality. It focuses on customer needs and streamlines the process to eliminate errors and waste.

In the authors opinion there are a few pitfalls that are important to remark upon. Although management uses concepts of "zero errors" and "do it correctly the first time" to motivate its staff to perfect their effort to deliver a quality product, it does not mean that there is no need to measure the quality of the product. Probability of failure is an example of an important issue needed to be questioned for a software product. Often management can meet such questions by answering that they use concepts as "zero errors" and "we do it correctly the first time", so a reliability model is not needed. This shows lack of understanding and will to discuss efforts to improve technology and methods in area.

2.2.4 The Software Safety Lifecycle

It can be useful to base a quality management system on a product lifecycle. We will summarise the basic steps of safety lifecycle in IEC 61508 [14] which is applicable to programmable safety related systems. There are similar lifecycle summaries in Lalli & Malec [2], Villemeur [3] and Fenton et al [4]. Typical life-cycle phases are:

1. *System concept, definition and scope of hazard and risk analysis:* To develop an understanding of the equipment under control and its environment and to determine and define the control system. The equipment and the control system to be included in a hazard and risk analysis shall be specified.
2. *Risk analysis and Safety requirements allocation:* Hazard and risk analysis is used to identify hazardous event scenarios of the equipment under control. Overall safety requirements are established and safety functions contained in overall requirements are allocated to the safety related subsystems. To be specified are types of accident-initiating events and subsystems associated with hazards.
3. *Design and development:* The software safety design is based on the concepts and the allocated safety requirements for the particular subsystem. A subsystem-architecture is designed. The internal logic of each structure is created to perform the functions described in the functional specification. This involves division into modules so that one should be able to test each module independently.
4. *Coding Process:* In this phase the software source code is generated with the chosen or required coding language and tools. The code is reviewed to see if it conforms to the requirements and logic in the previous steps. It is debugged and compiled to detect syntax errors that must be removed so that the program becomes executable.
5. *Software module testing and integration testing:* Each of the modules developed during design and development phase is tested and integrated to verify the software system design and software architecture.

6. *Software operation and modification procedures*: To establish information and procedures concerning software system in order to ensure the functional safety of the system during operation and modification. This could involve impact analysis and revalidation of the system as consequence of a modification. It is up to the operation and modification procedures to specify needed actions.
7. *Software safety validation and verification*: The conformity of the integrated system with its functional specifications for performance, safety and reliability is verified. Safety tests are performed since they determine future expectations for the safety. Data input variation is tested as well as verification of main functions are taking place.
8. *Overall installation, validation and operation & maintenance*: The software is integrated into a system and it is required to consider the overall system design with respect to safety. The outputs from the previous lifecycle steps is being evaluated and integrated with outputs from other subsystems to an overall system safety "validation and verification".

A model that is often used to describe software development and integration testing to verify the software modules is the V-model described by IEC 61508 [14] standard. Steps 3, 4 and 5 in the lifecycle above summarises the V-model.

It is up to the quality system and quality manuals to precisely define lifecycles. Here we have shown a typical one that is at an overall and general level. Note that lifecycle models not only consider the software but also its environment and "housing" at an overall system level. In specialised areas the lifecycle can be specially fitted. For example projects in the space industry have a similar life cycle as the one described above, see ECSS-M-30A [15]. Each phase is normally ended with a review as one milestone for the project. The equivalence of the "Concept and definition phase" is "The mission/function activities phase". This phase is closed by the 'Preliminary Requirements Review' (PRR). The PRR identifies the feasibility status and leads to a functional configuration baseline. The review is the final "measurement" to see whether quality objectives have been met.

It can also be wise to have different software metrics for different lifecycle phases. For example error and fault counts during the coding process can be one way to evaluate the debugging effort while probability of failure is measuring how well the software works during operation (see Lalli & Malec [2]).

There is a general assumption that it is less expensive to achieve reliability by removing faults before the verification and validation phase (see Pressman [13]). Therefore there is an economic motivation for quality assurance in early design phases to remove faults before the software reaches verification and validation.

In the authors opinion the motivation to have a high quality level during the lifecycle steps before operation and maintenance is that one believe that it ensures high reliability in the software during the operational phase. This is worth a remark and one should be careful to blindly believe this assumption. One can easily imagine examples where a need is wrongly specified in the requirements and thus gives an unreliable product although the quality assurance goals for each step in the project were excellent. For example imagine that you are requested to deliver a world-wide-web service with a software/hardware system to support users of a certain product. A market survey shows that the software should be able to handle 10 users online simultaneously. If the survey was wrong and it should have been 100 users instead,

the service will likely not be able to handle all the user requests. Hence, the user considers the service to be unreliable. A real example is the Ariane 5.01 accident (see inquiry board report [16]). It is unlikely that the director of ESA (European Space Agency) would have officially supported the launch in the news right after the accident if the quality assurance reports were unsatisfactory. In fact the inquiry board recommended ESA to change its quality assurance review for certain types of software.

2.3 Quasi-formal Reliability Techniques

The human mind has difficulty in setting up perfect logical structures. This is demonstrated since few programs run "perfect" the first time they are executed. In this Section we will give an overview of some tools and methods that can improve the reliability of the software by finding faults in the code, by investigating failure mechanisms or searching for discrepancies between requirements and code implementation. But first we will consider some languages and development techniques that can have an impact on the ability to detect faults. However, note that it is not our intention to cover details of development techniques or formal design methods in software engineering. Pressman [13] details such topics in his software-engineering book.

2.3.1 Languages

The selection of computer language can have an impact on the reliability of software. According to O'Connor [12] past experience shows that modular and structured programming (see Section 2.3.2) is easier to implement in Pascal and Ada than in Basic since Basic uses Goto and GoSub commands. It is therefore believed that the choice of language is important. The choice of programming language as well as development techniques should therefore depend on the mission and the environment the software operates in. Newer languages also include object-oriented programming features (see Pressman [13]).

O'Connor [12] emphasises the difference between the programming language levels. It is more difficult to understand assembly language programs compared to high-level language programs by looking at the source code. Since it is more difficult to communicate assembler code to people that are not computer experts it is often considered to be more difficult to find faults in assembly level programs. Clearly, communication is important during a review and an "advanced" and difficult language can make a review more difficult and reduce the ability to find faults.

2.3.2 Development of Software - Structured & Modular Programming

'Programming style' is the expression used to cover the general approach to program design and coding. Structured and modular programming are aspects of style. Other aspects are use of remarks and tabulators in the source code to make it easier to understand. See Yourdon [17] and Shooman [18] for more about structured and modular programming. Pressman [13] gives a good introduction to applied development techniques in software engineering including structured and modular programming.

Structured programming is an approach that constrains the programmer to use certain clear, well-defined approaches to develop software code. It is believed that structured programming leads to fewer errors, and to clearer, more easily maintained software. For example:

Unstructured:

```
10: If A > B then goto line number 40
20: else goto line number 30
30: Execute B
40: Execute A
```

Structured:

```
If A > B Subroutine-A
else Subroutine-B.
```

Modular programming breaks the requirements down into separate, smaller program requirements, or modules that can be separately specified, written and tested. This makes things easy to understand which is important for error checks and corrections. It is hoped that although the approach is more time consuming due to the extra preparation required, this time will be repaid in the end since less time is needed for program writing and debugging. O'Connor [12] suggests as a rule of thumb that modules should not exceed 100 separate commands or lines of code in a high level language.

Since structured and modular programming do not allow the programmer full freedom to create the fastest possible code or the code that uses memory optimally one should expect the programs to be less efficient with respect to speed or memory requirements.

2.3.3 Fault tolerance

Programs should be written such that faults do not cause serious problems, or complete failure of the program. 'Robustness' is thus an aspect of fault tolerance. We will quote O'Connor's [12] description of how a program has fault tolerant behaviour:

A program should be able to find its way gracefully out of an error condition and indicate its error source. This can be achieved by programming internal tests, or checks of cycle time, with a reset and error indication if the set conditions are not met. Where safety is a factor, it is important that the program sets up safe conditions when an error occurs.

Note that O'Connor uses different terminology than we do. He uses for example error condition for what we refer to as fault condition. The IEC 61508 [14] describes fault tolerance as the ability of a functional unit to continue to perform its required function in the presence of faults or errors.

Various authors (see below) have attempted to achieve fault tolerance via redundancy in the software. There are several ways to implement redundancy in software. Two methods that applies different source code (differently programmed) implementations are:

- *N-version programming*: N programs, designed under different conditions and using different algorithms, are executed simultaneously. The voter compares the outputs and the common output is validated. Whenever outputs differ the majority output is validated. Two-version programming is often used. Two "independent" versions of the software are produced. The operator is alerted when or if the two outputs differ. The program is halted after notifying the operator. In reality it is in fact hard to have two independent versions of a software program since the persons developing and programming them might come from the same school or have read the same books during their studies so they would tend to use similar design solutions and approaches.
- *Recovery blocks*: N programs, designed under different conditions and using different algorithms, are executed consecutively. They are successively executed until a validation condition is met.

Note that the N -version programming differs from recovery blocks in how the programs are executed i.e. simultaneously or successively. It is hoped that N programs create "diversity" so that fault tolerance is established. In Anderson & Lee [19] fault tolerance techniques are described. Villemeur [3] discusses results from two studies by Kelly & Avizienis [20] and Anderson et al [21] that have been experimenting with the two approaches. It seems that the fault tolerance successfully eradicated the effect of the faults. Note that both experiments involved the use of inexpensive manpower since students were used to program the different versions. It could be very costly to make several different versions of a program for industry and one should question if there exists less expensive and more efficient ways to obtain fault tolerance.

In space, computers are vulnerable to single event upsets (bit flips) caused by radiation. Takano et al [22] discuss the use of redundant computer systems to reduce the number of software failures due to single event upsets. In the space environment two computers running the same software with the same input parameters can give different outputs due to single event upsets. Hence it is not necessary to have N different versions of a program to create "diversity". Note that space provides a computer environment unlike the one we are used to on earth. The computation and data transfer processes are vulnerable in space and thus introduces failures that must be handled. The fault tolerant space computer described by Takano et al [22] show how redundancy and voting is capable of handling single event upsets in the telemetry data transfer systems of a satellite.

An alternative way to introduce fault tolerance is to build in redundancy in the requirements. Figure 4 shows a real example of required control flow for satellite software (in fact Figure 4 shows the required control flow for a similar problem as described in the control software example of a satellite in Section 2.1).

Figure 4. Required control-flow (see Dore & Norstrom [28]).

After separation the 'Pyros' shall be fired. To improve fault tolerance there is implemented a RETRY path if the program has not finished correctly, that is if it did not fire the 'Pyros' given separation.

From the author's experience the difficulty with RETRY redundancy as shown in Figure 4 is due to lack of detail in the system requirement specifications in early design phases. Implementation would require some kind of monitoring of the 'Pyros' to see whether they are fired or not. For example the requirements should specify how the software decides that it has finished correctly.

2.3.4 FMEA & FMECA's of software-based systems

According to O'Connor [12] it is not practical to perform FMECA on software, since software 'components' do not fail (i.e. software failure is only attributed to design failures). Musa [23] agrees with O'Connor and for the same reasons does not recommend FMEA. If you knew what specific failure would occur, you would fix it during the coding phase. Note that FMEA and FMECA are closely related.

The author believes that information from an FMEA/FMECA of interfacing hardware equipment can be useful when software code is developed. For example consider the control software example in Figure 1. Assume that the specifications require the program not to cause a failure if the sensors are failed. An FMECA of the sensors that detects separation would give useful information about how the sensors fail. This information can again be used to introduce fault tolerance in the software. For example if it is known that the sensors transmit a failure signal if they have failed, it is

easy for the programmer to utilise this information to prevent software failure if the sensors fail. Musa [23] suggests similar applications of FMEA. However he points out that FMEA is an expensive analysis and one would apply it typically only on modular (system) level for critical operations. Used wisely it can be useful to help focus design review, code review and unit test efforts.

2.3.5 Formal Technical Reviews

During a software project there are reviews for example of specifications, systems and source code. A review is normally an activity to find problems that must be improved or removed. A review is an activity that often is an attribute of software quality assurance (see Section 2.2). Pressman [13] details the method. The objectives with a formal technical review are:

- To uncover errors in function, logic or implementation for any representation of the software.
- To verify that the software under review meets its requirements.
- To ensure that the software has been represented according to predefined standards.
- To achieve software that is developed in a uniform manner.
- To make projects more manageable.

Reviews are often carried out at various design steps to uncover errors early. A code review is the one that should uncover software faults e.g. $a > b$ instead of $a < b$. One way is to perform program code walkthroughs to cover main requirements or critical functions. A useful way of identifying critical functions is to list the system "DO's" and "DON'T's". Bishop [7] describes the DO's and the DON'T's as follows. In design for safety, the functions that the system must perform under specified conditions are called the systems DO's and what the system must not do under any condition are the systems DON'T's. The DO's and the DON'T's form a safety-related functional specification for the system. The system is here the part of the world that is being monitored or controlled. In this formalism a technical review should consider the system DO's and DON'T's. Depending on the criticality more formal methods like Software Sneak Analysis can and should be considered to analyse the system "DO's" and "DON'T's".

2.3.6 Software Sneak Analysis

In a software-controlled system it is difficult to error trace a failure due to the interaction between the hardware and the software. Hardware can fail due to degradation and the software may interpret the hardware wrongly. The hardware can have requirements that are wrongly specified, and when the system fails to operate as expected the failure is only seen in the software part.

Sneak analysis is briefly described in O'Connor's [12] as a method or as a set of methods to trace design errors. Sneak analysis is often considered to be similar to code walkthroughs, however it is a much more complex top down approach that goes to a specified detail level. The analysis starts by identifying a target (a required function) in the hardware from where to start backtracking a path to a source in the software (and vice versa). The goal is to check the path to see if there is inadvertent function activation or prevention of required activation in the system. A procedure

developed by European Space Agency to perform sneak path analysis is PSS-01-411 [26]. It was successfully applied on control software by Norstrøm [27] and the results are presented in Dore & Norstrøm [28]. The simplified diagram in Figure 4 shows the required control flow for parts of the control software that was analysed. One of the main findings was that the required RETRY (see Figure 4) was not correctly implemented in the source code due to a design error. Note that neither reviews nor testing were able to detect the fault before. The fault was hidden in the system and only a set of very special conditions revealed it during simulation. The conditions were uncovered by using path predicate analysis, which is one of the methods in sneak path analysis.

Other interesting references for sneak analysis are Campbell & Tuma [30], US Navy [31], Whetton [32] and Buratti & Pinkston [33]. These references treat sneak analysis in other areas ranging from chemical process industry to electric switching systems. Another related approach to detect *hidden failures* is given by Jalashgar [29].

2.3.7 Software Testing

Software testing is an important area that is by many software-engineering practitioners considered to be the most essential way to uncover software failures and faults. Tests are one of the main activities that are used for monitoring the software quality assurance process. We will separate between development testing and test for reliability growth and certification. Development testing is testing performed on for example software units before they are integrated. It also encompasses integration testing that is carried out when units are mounted together. Reliability growth test on the other hand has the purpose of “predicting software reliability growth”. It is typically carried out during the system test phase and often in connection with verification and validation (see Section 2.2.4). It uncovers and removes faults that create failures that are unacceptable during normal operation. Reliability growth test also gives information about when to release the software. Certification tests are the final tests to verify that the software is designed according to its requirements. After certification acceptance the software is released.

Pressman [13] discusses development and validation tests and explains *white box* and *black box* testing. Thunem [36] provides a literature survey covering software testing techniques and strategies. Musa [23] discusses software reliability engineering and the use of software reliability models. He uses concepts as reliability growth test and discusses the need for operational profiles of the software operations in order to predict the reliability based on test results. Chen et al [37] gives an example that shows the impact of test techniques on estimation of reliability when the basic Musa model (Musa model see Section 2.4.10) is used. This is essentially not new since Shick & Wolverton [43] discuss the need for operational profiles for reliability prediction already in 1978. However it shows that one still is debating the issue and there are conflicting opinions about test strategies and software reliability estimation. For example: According to Pressman [13] tests are carried out to detect failures and faults. When one test to reveal failures and faults one are interested in strategies that efficiently find as many failures and faults as possible. It is an iterative process where the designer tests the code whilst it is being developed/debugged. This makes it possible to correct errors and faults while the designer has the code fresh in his own memory. Code corrections are also less expensive in early development phases. To be tested are: Operations at extreme conditions (timing, input parameter values and rate of change), ranges of input sequences and fault tolerance (error recovery). It is

important to test the most critical input conditions and one should consider the system DO's and DONT's as described in Section 2.3.5. According to Musa [23] however one should not test input areas in the input domain of the program where one expects to find a large number of failures *during system test phase* if these areas do not represent intended operations of the system. If certain inputs are improbable one should not waste resources exhausting them and forget to test more frequently used functions of the software. He argues that an efficient test strategy reveals failures likely to cause problems during normal operation. It is not clear whether Musa's approach can be used for a safety critical system that can have a rather long list of DO's or DONT's that are all improbable or rare and are in the areas outside normal use.

Software testing is a field in itself and it is not our intention to cover the area. It is however important that the reader has in mind that test-strategies influence the reliability predictions when he reads about software reliability models in Section 2.4. This is unfortunate and creates debate and confusion in the field of software reliability modelling and testing. Musa [23] describes how to apply tests together with software reliability models for decision making problems such as the choice of release date for software.

2.3.8 Reporting of Problems

In a software quality assurance system it is important that the organisation have initiative to record software failures, faults and errors so that they can ensure that they are handled professionally when discovered. It must be decided to what extent failures and problems shall be recorded during for example the maintenance phase. If reliability models are to be used it is important that failure severity as well as execution times are monitored. If one can not report execution times then one should use "natural units" that indicate for example number of transactions before failure. Musa [23] discusses how one can convert data from natural units to program execution times. There are various systems for error reporting and various organisations have different approaches. Thayer et al [47] have a chapter on data collection where examples of forms for reporting problems are given. Note that manual reporting is not the only option and Thakur & Iyer [38] give an example of an automatic reporting system to uncover and locate network problems. Their experience shows that there are a limited number of faults that causes most of the failures that are reported automatically.

2.4 Software Reliability Prediction Models

Our intention here is to review various software reliability models in order to understand what they measure and what assumptions that have been made. As an introduction to the more detailed part where the models are explained in detail we intend first to categorise the models as suggested by Lalli & Malec [2], Ramamoorthy & Bastani [24] and Goel [25]. In this process we will reference literature in the area and summarise main assumptions. A recent review of existing approaches is given by Thunem [36] that covers software reliability models into the beginning of the nineties.

TIME-DOMAIN MODELS

Software reliability models in the time domain relate time to failure to the number of faults present in the software at the given time. Many time-domain models assume that the software failure rate is proportional to the number of faults in the software. They do not really consider that different kinds of faults contribute differently to the total failure rate or the system reliability. However, one of the advantages is that they measure the current software so that they are not based on past information. Bittanti [9] is the editor of lecture notes where several of the known "founders" of software reliability have contributed. It covers the field in this area until the late eighties. Musa [23] summarises the models used in software reliability engineering in late nineties. He discusses how difficult it was to validate the models in the eighties and seventies due to insufficient data and compares several of the models below. We first present a resume of our literature survey of time domain models and give a more complete description of the models latter in Sections 2.4.1 to 2.4.10.

Jelinsky & Moranda [39] developed a model known as the Jelinsky-Moranda model (JM-model) and it was presented at a conference for evaluation of statistical methods at Brown University in 1971. Shooman [40] also independently derived the model about the same time. Note these models are also known as a Binomial models and for example Angus et al [41] show the link to the Binomial distribution. The model can also be developed from the assumption that the inter-failure times $T_1, T_2, \dots$ are independent random variables with exponential probability density function (see Littlewood [8]). Since the exponential distribution is central in reliability theory we will use the exponential assumption later. The JM-model is generalised by Shick & Wolverton [42], [43]. They also assume that the error rate is proportional to the time spent to debug the software. Hence, the chance of discovering an error increases as time goes on. Moranda [44] assumes a geometrically decreasing intensity. Thayer et al [47] give another extension, where more than one error can be detected in a time interval with no corrections being made at the end of the interval. Thayer et al's [47] book also contains collections of data and discusses in detail model parameter estimation.

Mijamoto [49] gives a simple model for an on-line real time system which requires that the MTBSE (=Mean Time Between Software Errors) has to be more than about 30 days. The reliability model is $R(t) = e^{-t/MTBSE}$ where t is the fraction of hours the system is operational per day and MTBSE is a number found via experience from similar projects.

Littlewood & Verral [50], [51] developed the Littlewood-Verral model (LV-model) in 1973. It is introduced as a Bayesian reliability growth model with gamma prior distribution and an exponential sample distribution. Keiller et al [52] developed the Keiller-Littlewood model (KL-model), Littlewood [53] developed the Littlewood model (L-model) and in [54] the Bayesian JM-model. These models can all be considered to be reliability growth models with a gamma prior distribution and an exponential sample distribution. However, they implement the growth differently. Littlewood [8] gives a good overview of the models where he also compares performance to the traditional JM-model.

Littlewood [8] briefly describes the Weibull order statistics as a promising model for software reliability.

Musa [48] proposes a parametric debugging model called the Musa-model or the basic Musa model. It has similarities to the JM-model. The merit of the model is that it is the first software reliability model that develops theory to consider program execution time instead of calendar time. Later Musa & Okumoto [10] describe the Musa-model as an exponential intensity Poisson model.

The first software reliability Poisson process models starts to appear in 1975. They are based on the assumption that the failure intensity caused by faults is decreasing so they use a Poisson distribution with a decreasing intensity. Several authors including Schneidewind [59], Musa [48] and Goel & Okumoto [60] use exponential intensity models. The first logarithmic intensity model is proposed by Musa & Okumoto [61] in 1984. Musa [23] recommends the use of logarithmic and exponential intensity models since the models are capable to fit well to various kinds of software reliability growth data. Kaaniche et al [62] use the hyperexponential Poisson process model successfully to estimate reliability in three successive generations of a telecommunication switching system. The paper shows a data study for the three systems where data is collected from users during the operation and maintenance over a period of in total 5 years. From the modelling conclusions are drawn about differences of for example the development and functionality for the three systems. The contribution to the failure intensity of various software subsystems (components) was compared to the overall system failure intensity. It was shown that summation of individual component intensities gave an acceptable estimate of the system failure intensity. Later in 1996 Kaaniche & Kanoun [63] perform a similar study of a different telecommunication system. The preliminary data analysis attributed about 60% of the failure reports to software related problems.

Evaluation of various models has been difficult and parameter estimation can also be difficult see Littlewood & Verral [51]. Brocklehurst et al [64] discuss these problems and come up with a method to recalibrate the models we have discussed above. They show that re-calibration of software reliability models gives better fit to the data. It seems however that the method needs rather large data sets (they use rather large data sets in their examples). One of the basic assumptions for the re-calibration method to work is that the reliability model used predicts consistently wrong. One should conclude that the model for example consistently overestimates the failure probability. Brocklehurst et al [64] use u-plot techniques to verify this assumption.

FAULT SEEDING AND DATA DOMAIN MODELS

In principle the sets of all input data upon which the software program can operate is identifiable. The reliability of the software can be forecasted by running the program on a subset of input data. The failures are identified by for each input-data establish how well the output of the program fits the required output. Based on this information the reliability is forecasted. A typical measure for software reliability is given by the probability of no failures after k runs. The total number of faults in the program is also of interest. A resume of the literature is given below and some basic models are discussed in Sections 2.4.11 to 2.4.13.

The Nelson model dating back to 1973 is described by Thayer et al [47]. The development of the model was important since it gives an intuitive understanding of software reliability. The main assumption is that a software program is regarded as a function of the possible input data to the data that is the output of the program similar to the picture shown in Figure 2. The reliability can be computed as the number of successful runs divided by the total number of runs. Thayer et al [47] show how the

model via "heuristic" assumptions gives an exponential intensity model which is one of the main assumptions for time domain models. One of the major drawbacks of the model is that it would be intuitively wrong to use the model for strategic testing to efficiently remove errors since it is based on the operational profile (see Section 2.3.7). Ramamoorthy & Bastani [24] extend the Nelson model to get a model that also can be used for strategic testing. They have developed the model for reliability predictions in real-process systems where few failures are experienced during testing. Miller et al [65] use Bayesian modelling to estimate parameters in Nelson's model. They develop a model to predict software reliability that does not give zero failure probability when testing reveals no failure. However, the model seems to have a problem when it comes to parameter estimation of additional parameters used in the prior distribution.

Error-seeding is briefly discussed by Ramamoorthy & Bastani [24] and Goel [25]. Schick & Wolverton [66] provide a mathematical discussion of the topic by using hypergeometric distributions. They base their discussion on an unpublished paper written by Mills in 1970. Mills suggests to seed errors into the software and debug the software afterwards to estimate the number of faults in the software. The concept is to plant m errors in the software and to use testing techniques or other quality assurance- or development- techniques to see how many planted faults are detected. Ramamoorthy & Bastani [24] point out that the method is impractical due to the large number of program tests required to obtain a good prediction. Tohma et al [67], [68] have researched parameter estimation and reliability growth modelling by using the hypergeometric model. Huei Hong et al [69] have investigated optimal release-policies by considering cost of testing and cost of fixing faults in the test phase versus cost of fixing faults after the software is released. Avresky et al [70] describe a formal test method based on injecting faults to advanced systems containing fault tolerance. The system is represented by FTAM (fault-tolerance algorithms & mechanisms) which is a formalism of how to represent systems containing e.g. control-software. The idea is to inject failures to simulate how the system responds. The article shows how the injected faults are able to reveal design errors. Avresky et al [70] do not address reliability prediction however they show a practical way of using error seeding. Takano et al [22] use error seeding to verify the reliability of a fault tolerant computer system that have been built to operate in space. The data collection was extensive over a three year period of time and cover data of 655 single event upsets (bit flips) which the computer system should be able to withstand (see also Section 2.3.3).

MARKOV MODELS

Markov and Semi-Markov models often describe switching between different states in a software system. A simple model can for example have a state that describes software failure and another that describes software operation. They use the Markov property for stochastic processes which state that the probability of the outcome in the present state only depends on the previous state. We give a brief introduction in Section 2.4.14 where some of the basic models are shown. Note that the models can be changed and fitted to describe various systems. How to solve Markov models are an own study which we will not detail here however there exists mathematical techniques e.g. see Villemeur [3]. We have separated Markov and Semi-Markov models from the models above since they are capable of more detailed description of the system and its software states than the other models. They can also be used for studies similar to the models in the time domain, however they often contain more

parameters and are therefore considered more difficult and sometimes impractical for real application. However, Markov models are said to increase system understanding since they are capable to describe various states of a system in detail. Literature in the field is described by Thunem [36] however in the remainder of this Section we will briefly describe some historic papers.

The first papers covering this topic arrived around 1975. Littlewood [71], [72] describes a Semi-Markov process for a modular program structure. The Markov property ensures that the probability of calling a given module is a function of only the calling and the called modules. He shows that the failure process is asymptotically a Poisson process. Goel & Okumoto [73] describe a simple Markovian model that models a process where the repair of a fault is imperfect. The model can be used to predict time to a completely debugged system, time to a specified number of faults remaining and for “standard” reliability predictions. They use the model to analyse a real software project and conclude that the model gives good representation of the observed data. Costis et al [74] propose a Semi-Markov model that describes both hardware failures and software failures. This is accomplished by using two different failure states where one describes software failures and the other hardware failures. The model describes the repair process for the system after a software failure. Software repair is different than the repair following hardware failure. For example after hardware failure a component is replaced and the system is as good as new while after a software repair the system is changed (a different software program).

Thunem [36] describes an interesting concept developed by Whittaker in his dissertation. Whittaker suggested using Markov models to describe the operational profile of a program and to generate test sequences. The idea is to do this before generating the code.

It is the author’s impression that Markov models are applied less than the models in the time domain above. One of the reasons for this is lack of data since complex Markov models require more detailed system insight that also must be reflected the reliability data gathered. For the other models described in the previous sections above the system is an unknown black box and most data available is sampled for this purpose.

AXIOMATIC MODELS

A well-known axiomatic model based on the software science theory developed by Halstead [76], [77] is briefly described by Lalli & Malec [2]. The approach provides quantitative measures of program level, language level, algorithm purity, program clarity, effect of modularization and programming time. In particular the estimated number of faults in a program is given by $B = K \frac{V}{E_0}$ where

1. K proportionality factor
2. E_0 mean number of mental discriminations between errors made by programmer
3. V volume of algorithm implementation, $V = N \log_2(n)$ where
  - A. N program length
  - B. n size of vocabulary defined by language used. More specifically:
  - C. $N = N_1 + N_2$
  - D. $n = n_1 + n_2$ where

4. N_1 total number of occurrences of operators in a program
5. N_2 total number of occurrences of operands in a program
6. n_1 number of distinct operators appearing in a program
7. n_2 number of distinct operands appearing in a program

Lalli & Malec [2] reference an example where E_0 has been estimated to 3000 and the results of the method have been discussed. The correlation varies from 0.75 to 0.99 between prediction and the number of faults found in the software. They emphasise that the model allows one to quickly estimate the program reliability.

The model is based on experimental intuition and can not be proved from mathematical or probabilistic principles. However, good correlation alone is not sufficient as an evaluation of the model since one should always compare with alternative models or methods.

Note that the model is based on theory developed some 20 years ago for software engineering at that time. Since then, software development practise has changed while the model has remained unchanged so it is hard to have confidence in the model since it can be outdated. However, the model seems to have a status in software engineering since it is used as an example of a software reliability model by Pressman [13] in his book about software engineering while most other models discussed above are left out.

2.4.1 The Jelinsky and Moranda (JM)-model

Jelinsky & Moranda [39] and Shooman [40] developed this model independently about the same time. The JM-model assumes that the inter-failure times $T_1, T_2, \dots$ are independent random variables with exponential probability density function

$$p(t_i | \lambda_i) = \lambda_i e^{-\lambda_i t_i}, \quad (1)$$

where $t_i > 0$ and $\lambda_i = (N - i + 1)\varphi$. At the beginning of testing the model contains N faults. Removal of a fault occurs whenever a failure occurs. By removal of a fault the rate of occurrence of failures is reduced by φ . Thus φ can be taken to represent the size or the frequency of a fault.

We will develop the JM-model by using competing risks following Littlewood [8]. When the program contains N faults then it fails when the first fault causes a failure, hence the program can be considered as a series "system" where we imagine that a fault is equivalent to a component in standard reliability analysis. For $k = 1, 2, 3, \dots, N$ let fault $B_k(t)$ be defined such that

$$B_k(t) = \begin{cases} 0 & \text{if fault } k \text{ occurs at time } > t \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

When testing is started the program is expected to contain N faults. After the program has failed the fault that causes the failure is removed without introducing other faults. We use order statistics so that $B_{(1)}(t)$ denotes the first fault that occurs (see Figure 5).

Figure 5. A set of series structures representing remaining faults in the program as fault corrections are done.

The number of faults that remain in the program before removal of fault i is $(N - i + 1)$, $i = 1, 2, \dots, N+1$. The structure function for the program when modelled as in Figure 5, and after having removed $i - 1$ faults is

$$\Phi(\mathbf{B}(t)) = \prod_{k=1}^{N-i+1} B_k(t). \quad (3)$$

By introducing the probability that the program will not fail due to fault k as $p_k(t) = E\{B_k(t)\}$ the reliability of the program after detection and removal of $i - 1$ faults is

$$p_s(t) = E\{\Phi(\mathbf{B}(t))\} = \prod_{k=1}^{N-i+1} p_k(t), \quad (4)$$

if the faults B_k , $k=1, \dots, N$ behave independently and have identical distributions. By assuming that each fault has an exponential reliability function $R(t|\varphi) = e^{-\varphi t}$ it can be shown that the program reliability given that $i - 1$ faults have been removed is

$$p_s(t) = \prod_{k=1}^{N-i+1} p_k(t) = e^{-(N-i+1)\varphi t} \quad (5)$$

as above in (1).

The unknown parameters N and φ in the model are estimated numerically by the maximum likelihood approach. The maximum likelihood equations are given in Villemeur [3]. Littlewood & Verrall [51] discuss bounds that must be satisfied for the maximum likelihood estimate of N to exist. By looking at the model it also seems plausible that stability problems can arise when estimating N . N is the 'maximum' number of faults and often difficulties arise when extreme values are estimated.

To summarise the JM-model assumptions:

1. Faults are corrected immediately after detection and debugging does not introduce 'new' additional faults.
2. Failures are detected at inter-failure times $T_1, T_2, \dots$ which are mutually independent from each other.

3. Faults are identically distributed with reliability function $R(t|\varphi) = e^{-\varphi t}$.

Serious criticism of the model concerns assumption 3 since it is not likely that all faults are equally probable at time t . Littlewood [8] claims that if the faults are not equally probable the use of the JM-model may lead to too optimistic predictions. The model is also criticised for assumption 2 that the debugging does not introduce new faults since code changes may introduce additional new faults.

2.4.2 The Schick - Wolverton model

Schick & Wolverton [42], [43] give a model that is based on the same assumptions as the JM-model apart from assumption 3. The model assumes that the rate of detection of faults is proportional to the elapsed time u since discovery of fault $i - 1$. Hence the failure intensity (or intensity for discovering a fault) can be written as $z(u) = \varphi u$, where φ is the amount of the intensity contributed by a fault. This represents a way of accelerated detection time since fault intensity also is proportional to the time spent on debugging. The probability that fault k does not cause a failure before t is

$$R(t|\varphi) = e^{-\int_0^t \varphi u du} = e^{-1/2 \varphi t^2}, \quad (6)$$

so the reliability in assumption 3 above must be changed with the reliability function (6). It can be shown (in a similar way as for the JM-model) that the program reliability given that $i - 1$ faults have been found (and removed) is

$$p_s(t) = \prod_{k=1}^{N-i+1} p(k)(t) = e^{-1/2(N-i+1)\varphi t^2}. \quad (7)$$

The unknown parameters N and φ in the model are estimated by the maximum likelihood approach.

2.4.3 The Geometrical De-Eutrophication Process

Moranda [44] proposed a generalisation of the JM-model that assumes a geometrically decreasing intensity. Ramamoorthy & Bastani [24] show a generalised model by describing the intensity after fault i as $z_i(u) = KD^{i-1}u^{\alpha-1}/(\alpha-1)$. From this the reliability of the program after fault i can be estimated as

$$R_i(t) = e^{-\int_0^t z_i(u) du} = e^{-KD^{i-1}t^\alpha} \quad (8)$$

The parameters to be estimated are K , D and α . The parameters can be estimated by applying the maximum likelihood approach. The generalised model summarises geometrically decreasing failure rate models being developed in the seventies following the concept of Moranda [44].

2.4.4 A Flexible Modelling Approach

In order to summarise some various variants of the JM-model Bittanti et al [11] creates a flexible model with the JM-model as a special case. The model is built up in the same way as the JM-model, using similar assumptions. The CPU-time T_j between

the j 'th and $(j+1)$ 'th failure is exponential $f_j(t_j) = z_j \exp(-z_j t_j)$ where the observed failure times $T_{j,j \in 0, \dots, n-1}$ are independent. The intensity z_j is defined as

$$z_j = k(j) \cdot (N - j), \quad k(j) \geq 0, \quad (9)$$

where $k(j)$ in the following is the so-called Fault Exposure Coefficient (FEC) defined as

$$k(j) = k_i + (k_f - k_i) \frac{j}{N}, \quad k_i > 0, k_f \geq 0, \quad (10)$$

and k_i and k_f are the initial and final values of the FEC at the beginning and at the end of debugging respectively. As usual N is the number of faults in the program. Four special cases of the FEC should be noticed:

- i. Constant FEC: When $k_i = k_f$ the model reduces to the JM-model.
- ii. Decreasing FEC: If $k_i > k_f$ the intensity z_j decreases to zero.
- iii. Vanishing FEC: In the limit case $k_f = 0$ the intensity z_j is proportional to $(N-j)^2$. It is called vanishing since the final value of FEC becomes zero.
- iv. Increasing FEC: If $k_f > k_i$, z_j may not be monotone decreasing. If $k_f > 2k_i + (k_f - k_i)/N$ then the z_j 's exhibits a maximum. This might be explained by increased skills in the testing team or modifications of the testing strategy.

This model is illustrated with real data in Bittanti et al [11]. In the two next paragraphs the maximum likelihood equations will be deduced and discussed followed by a discussion, whether it is appropriate to use the asymptotic normal distribution for this model as suggested by Bittanti et al [11].

The likelihood function for the observations $T_{j,j \in 0, \dots, n-1}$ is given by

$$L(\theta | t_0, \dots, t_{n-1}) = \prod_{i=0}^{n-1} z_j e^{-z_j t_j} \quad \text{where } \theta = [N, k_i, k_f]^T. \quad \text{This gives the log likelihood}$$

$$\ln L(\theta | t_0, \dots, t_{n-1}) = - \sum_{j=0}^{n-1} z_j t_j + \sum_{j=0}^{n-1} \ln z_j. \quad (11)$$

The maximum likelihood estimate is obtained by differentiation of the log-likelihood and equating it to zero to find the critical points. By following this straightforward approach as suggested by Bittanti et al [11] we find the equations that need to be solved to obtain the maximum likelihood estimates,

$$\frac{d \ln L(\theta | t_0, \dots, t_{n-1})}{d \theta} = \sum_{j=0}^{n-1} \frac{1 - z_j t_j}{z_j} \begin{bmatrix} \frac{\partial z_j}{\partial N} \\ \frac{\partial z_j}{\partial k_i} \\ \frac{\partial z_j}{\partial k_f} \end{bmatrix} = \sum_{j=0}^{n-1} \frac{1 - z_j t_j}{z_j} \begin{bmatrix} \frac{k_i N^2 + (k_f - k_i) j^2}{N^2} \\ \frac{(N-j)^2}{N} \\ \frac{j(N-j)}{N} \end{bmatrix} = 0. \quad (12)$$

Bittanti et al [11] do not discuss the concavity of the likelihood function. It is plausible that this model, being a generalisation of the JM-model, has similar

problems to those discussed by Littlewood & Verrall [51] concerning finite estimates of the parameters.

Bittanti et al [11] do not solve (12). However, they suggest using the asymptotic normal distribution of the maximum likelihood estimates to estimate the parameters. In the asymptotic theory n must approach infinity while according to the physical limitations of the model n must be finite since $N > n$. Hence it is questionable whether it is mathematically sound to use asymptotic theory for the model described above. However, if it is insisted that N and n are sufficiently large one should verify that the non-identically distributed random variables $\{T_i\}$ causes problems in the asymptotic theory. For example the Central Limit Theorem may not hold. Feller [78] shows that the Central Limit Theorem holds for not identically distributed random variables if the Lindberg condition holds. The invariance of the maximum likelihood estimate must also hold. The usual way to prove the invariance is to use the weak law of large numbers (see e.g. Rao [79]).

2.4.5 The Bayesian JM-model

Littlewood & Sofer [54] introduce the Bayesian JM-model. They use a slightly different parameterisation than the one used in the JM-model. The frequency of failures or the discovery rate of faults is defined as $\lambda = N\phi$ where λ is described as the initial rate of occurrences prior to testing and ϕ is the improvement that results by a fix. The physical implications in the parameterisation suggest that $\lambda \geq \phi$. A improper prior-distribution

$$\pi(\lambda, \phi) = 1; \lambda, \phi > 0 \quad (13)$$

is assigned to λ and ϕ . Littlewood & Sofer [54] motivate the prior by assuming that λ and ϕ are independent with distributions $\lambda \sim \text{gamma}(b, c)$ and $\phi \sim \text{gamma}(f, g)$ ($\text{gamma}(\alpha, \beta)$ is the probability density function $\pi(\phi|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \phi^{\alpha-1} e^{-\phi\beta}$), and (13) is then the product of these two priors when $c, g \rightarrow 0$ and $b, f \rightarrow 1$. The inter-failure times $(T_1, \dots, T_n)$ are assumed to be independent and have the reliability function $R_k(t) = e^{-(\lambda-(k-1)\phi)t}$. The likelihood function can thus be obtained as

$$L(\lambda, \phi | \mathbf{t}) = \begin{cases} \prod_{k=1}^n [\lambda - (k-1)\phi] e^{-(\lambda-(k-1)\phi)t_k} & \text{if } \lambda \geq (n-1)\phi \\ 0 & \text{otherwise,} \end{cases} \quad (14)$$

where $\mathbf{t} = (t_1, \dots, t_n)$ is the observed inter-failure times. Note that λ is not necessarily an integer multiple of ϕ . By the Bayes' theorem the posterior distribution is

$$\pi(\lambda, \phi | \mathbf{t}) \propto \begin{cases} \prod_{k=1}^n [\lambda - (k-1)\phi] e^{-(\lambda-(k-1)\phi)t_k} & \text{if } \lambda \geq (n-1)\phi \\ 0 & \text{otherwise.} \end{cases} \quad (15)$$

It is reasonable to demand that the posterior (15) is a proper probability distribution in both λ and ϕ . Hence $n \geq 2$, so that more than two observations are needed to enable parameter estimation from the posterior.

From (14) it is reasonable to assume that $\lambda - \varphi(n-1) \geq 0$, otherwise after removing faults the software would be allowed to have negative rate of occurrence, which is nonsense. Note that the likelihood function only allows estimation of both λ and φ if $n \geq 2$. Otherwise, if $n = 1$ the likelihood function is "non-informative" for φ . In general the likelihood should not constrain the prior. However, for $n = 2$ and $\lambda - \varphi(n-1) \geq 0$, we have that $\lambda \geq \varphi$. Hence, for $n = 2$ the likelihood constrains the prior so that $\lambda \geq \varphi$. This contradicts Littlewood's [54] independence assumption of λ and φ and his motivation via independent gamma priors for λ and φ . A more appropriate region for the prior (13) is $\lambda \geq \varphi \geq 0$. However, the prior (13) allows a Bayesian analysis of the problem without introducing any new model assumptions.

For further analysis it is handy to introduce some polynomials, and by using Stirling numbers of first kind (see Abramowitz & Stegun [55]) it can be shown that

$$\prod_{i=1}^n (x+i) = \sum_{i=0}^n a_{i,n} x^{n-i}, \quad (16)$$

where $a_{i,n} = n a_{i-1,n-1} + a_{i,n-1}$; $a_{0,n} = 1$ and $a_{n,n} = n!$. Thus the product $\prod_{k=1}^n [\lambda - (k-1)\varphi]$ can be rewritten as $\sum_{k=0}^{n-1} (-1)^k a_{k,n-1} \lambda^{n-k} \varphi^k$ which makes it possible to rewrite the likelihood in (14) as

$$L(\lambda, \varphi | t) = \begin{cases} e^{-\lambda \sum_{k=1}^n t_k} e^{\varphi \sum_{k=1}^n (k-1)t_k} \sum_{k=0}^{n-1} (-1)^k a_{k,n-1} \lambda^{n-k} \varphi^k & \text{if } \lambda \geq (n-1)\varphi \\ 0 & \text{otherwise} \end{cases} \quad (17)$$

and the posterior distribution as

$$\pi(\lambda, \varphi | t) \propto \begin{cases} e^{-\lambda \sum_{k=1}^n t_k} e^{\varphi \sum_{k=1}^n (k-1)t_k} \sum_{k=0}^{n-1} (-1)^k a_{k,n-1} \lambda^{n-k} \varphi^k & \text{if } \lambda \geq (n-1)\varphi \\ 0 & \text{otherwise} \end{cases} \quad (18)$$

Note that Littlewood's [54] corresponding expressions contain some typographical errors.

The right hand side of (18) needs to be normalised to apply the model. This involves some tricky integration and since similar kind of integration is also needed to calculate the posterior reliability the calculation of the normalising constant is done stepwise. The normalising constant C can be obtained as

$$C = \int_0^\infty \int_{(n-1)\varphi}^\infty e^{-\lambda \sum_{k=1}^n t_k} e^{\varphi \sum_{k=1}^n (k-1)t_k} \prod_{k=1}^n [\lambda - (k-1)\varphi] d\lambda d\varphi. \quad (19)$$

By substituting $u = \lambda - (n-1)\varphi$ the inner integral transforms to

$$C = \int_0^\infty e^{-\varphi \sum_{j=1}^n (n-k)t_j} \int_0^\infty e^{-\lambda \sum_{k=1}^n t_k} \prod_{k=1}^n [u + (n-k)\varphi] du d\varphi. \quad (20)$$

Then by using (16) the product term can be written

$$\prod_{k=1}^n u + (n-k)\varphi = \varphi^{n-1} u \prod_{k=1}^{n-1} (u + k\varphi) = \varphi^{n-1} u \sum_{k=0}^{n-1} a_{k,n-1} \left(\frac{u}{\varphi}\right)^{n-k-1}, \quad (21)$$

which can be inserted into (20). By also changing the order of integration and summing we have

$$C = \sum_{k=0}^{n-1} a_{k,n-1} \int_0^\infty \varphi^k e^{-\varphi \sum_{j=1}^n (n-k)t_j} \int_0^\infty u^{n-k} e^{-u \sum_{j=1}^n t_j} du d\varphi. \quad (22)$$

Using the definition of the gamma function gives

$$C = \sum_{k=0}^{n-1} a_{k,n-1} \frac{k!(n-k)!}{\left[\sum_j (n-j)t_j \right]^{k+1} \left[\sum_j t_j \right]^{n-k+1}}. \quad (23)$$

Prediction of the reliability of fault $n + 1$ with this model needs a little careful thinking. When estimating the reliability it should be conditional on the information given by the n preceding observations. Hence the reliability required is

$$R_{n+1}(t | t) = p(T_{n+1} > t | t) = \iint R_{n+1}(t | \lambda, \varphi) \pi(\lambda, \varphi | t) d\lambda d\varphi. \quad (24)$$

The conditional reliability of T_{n+1} given λ and φ is

$$R_{n+1}(t | \lambda, \varphi) = \begin{cases} e^{-(\lambda + n\varphi)t} & \text{if } \lambda \geq n\varphi \\ 1 & \text{otherwise,} \end{cases} \quad (25)$$

and multiplied with the posterior-distribution (15) gives

$$R_{n+1}(t | \lambda, \varphi) \pi(\lambda, \varphi | t) = C^{-1} \begin{cases} e^{-\lambda(t + \sum_j t_j)} \varphi^{(nt + \sum_j (j-1)t_j)} e^{-\varphi \sum_j t_j} \prod_{k=1}^n [\lambda - (k-1)\varphi] & \text{if } \lambda \geq n\varphi \\ e^{-\lambda \sum_{k=1}^n t_k} \varphi^{\sum_{k=1}^n (k-1)t_k} \prod_{k=1}^n [\lambda - (k-1)\varphi] & \text{if } n\varphi > \lambda \geq (n-1)\varphi \\ 0 & \text{otherwise.} \end{cases} \quad (26)$$

The integration needed to solve (24) is similar to that for the normalisation constant C . Now we will show how the integral (24) can be written in a similar form. The difficult part to solve is

$$\int_0^{\infty} \int_{(n-1)\varphi}^{n\varphi} e^{-\lambda \sum_{k=1}^n t_k} e^{\varphi \sum_{k=1}^n (k-1)t_k} \prod_{k=1}^n [\lambda - (k-1)\varphi] d\lambda d\varphi. \quad (27)$$

This can be rewritten as

$$\int_0^{\infty} \left[\int_{(n-1)\varphi}^{n\varphi} e^{-\lambda \sum_{k=1}^n t_k} e^{\varphi \sum_{k=1}^n (k-1)t_k} \prod_{k=1}^n [\lambda - (k-1)\varphi] d\lambda - \int_{n\varphi}^{\infty} e^{-\lambda \sum_{k=1}^n t_k} e^{\varphi \sum_{k=1}^n (k-1)t_k} \prod_{k=1}^n [\lambda - (k-1)\varphi] d\lambda \right] d\varphi,$$

where each term is similar to that in (19). Solving the integral gives the “posterior” reliability

$$R_{n+1}(t | \mathbf{t}) = C^{-1} \left(\sum_{k=0}^n a_{k,n} \frac{k!(n-k)!}{\left[\sum_j (n-j)t_j \right]^{k+1} \left[\sum_j t_j \right]^{n-k+1}} + \sum_{k=0}^{n-1} a_{k,n-1} \frac{k!(n-k)!}{\left[\sum_j (n-j)t_j \right]^{k+1} \left[\sum_j t_j \right]^{n-k+1}} \right. \\ \left. - \sum_{k=0}^n a_{k,n} \frac{k!(n-k)!}{\left[\sum_j (n-j)t_j \right]^{k+1} \left[\sum_j t_j \right]^{n-k+1}} \right) \quad (28)$$

Littlewood [8] claims that the experience with this model shows that it only offers marginal improvements compared to the JM-model itself. Littlewood concludes that this suggests that there is something wrong with the modelling of the exponential inter-failure times. Littlewood’s argument is unclear for the following reason. It suggests that Bayesian modelling can be used to validate how well the assumed exponential reliability function $R_k(t) = e^{-(\lambda - (k-1)\varphi)t}$ fits to the observed data set. If one question the model assumption one normally would use goodness of fit techniques to measure how well the model fits the data. Kolmogorov distance is for example one such measure.

2.4.6 The Littlewood (L)-model – Faults with different occurrence rates

Littlewood [53] introduces this model. It is a variant of the JM-model. It is different since the φ ’s are not identical, but assumed to be stochastic variables. The following assumptions are made. Faults are corrected immediately after failure without introduction of ‘new’ additional faults. Faults are detected at inter-failure times $T_1, T_2, \dots$ which are mutually independent. As usual λ_i is the current rate of occurrence and inter failure time T_i has reliability function $R(t_i | \lambda_i) = e^{-\lambda_i t_i}$. The initial failure rates $\vartheta_1, \dots, \vartheta_N$ are mutually independent and identically distributed with a gamma($\alpha, \beta + \tau$) prior distribution function

$$\pi(\varphi_k | \alpha, \beta, \tau) = \frac{(\beta + \tau)^\alpha}{\Gamma(\alpha)} \varphi_k^{\alpha-1} e^{-\varphi_k(\beta + \tau)},$$

where N is the number of faults in the program before debugging and τ represents the total program operation time prior to *now* as shown in Figure 6.

Figure 6. Visualisation of the debugging process.

After $(i-1)$ faults (faults = bugs) have been removed the remaining faults gives the failure rate

$$\Lambda_i = \vartheta_1 + \dots + \vartheta_{N-i+1}.$$

Since Λ_i is a sum of independent and identically distributed (i.i.d) with a gamma distribution it can be shown (see Remark below) that the prior $\pi(\lambda_i | \alpha, \beta, \tau)$ is

$$\pi(\lambda_i | \alpha, \beta, \tau) = \frac{(\beta + \tau)^{\alpha(N-i+1)}}{\Gamma(\alpha(N-i+1))} \lambda_i^{\alpha(N-i+1)-1} e^{-\lambda_i(\beta + \tau)}.$$

REMARK: Let $M_k(z)$ denote the moment generating function of ϑ_k . It can be shown (see Casella & Berger [56]) that when $\vartheta_k, k=1, \dots, N-i+1$ are i.i.d. gamma($\alpha, \beta + \tau$) then Λ_i has the moment generating function $M_{\Lambda_i}(z) = \prod_{k=1}^{N-i+1} M_k(z) = \left(\frac{(\tau + \beta)}{(\tau + \beta - z)} \right)^{\alpha(N-i+1)}$. This is the moment generating function of a gamma($\alpha(N-i+1), \tau + \beta$) distribution. Since the moment generating function is unique, the prior distribution $\pi(\lambda_i | \alpha, \beta, \tau)$ is gamma($\alpha(N-i+1), \tau + \beta$).

The conditional failure distribution of T can be obtained from the reliability function by taking the negative of its derivative. This gives

$$f(t_i | \lambda_i) = -R'(t_i | \lambda_i) = \lambda_i e^{-\lambda_i t_i}.$$

One can obtain the marginal failure distribution for T_i by integrating which gives

$$\begin{aligned} f_T(t_i | \alpha, \beta, \tau) &= \int_0^{\infty} f(t_i | \lambda_i) \pi(\lambda_i | \alpha, \beta, \tau) d\lambda_i \\ &= \frac{(N-i+1)\alpha(\beta + \tau)^{\alpha(N-i+1)}}{(\beta + \tau + t_i)^{\alpha(N-i+1)+1}}. \end{aligned} \quad (29)$$

where τ is the total time on test:

$$\tau = \sum_{k=1}^{i-1} t_k.$$

The reliability after $i-1$ faults have been removed is

$$R_i(t) = \left(\frac{\beta + \tau}{\beta + \tau + t} \right)^{\alpha(N-i+1)}, \quad (30)$$

and the failure rate is

$$z(t) = \frac{f_T(t | \alpha, \beta, \tau)}{R_i(t)} = \frac{\alpha(N-i+1)}{\beta+t+\tau}. \quad (31)$$

Littlewood [53] estimates the parameters α , β and N using maximum likelihood function. According to Littlewood [8] a proper analysis of this model has proven difficult largely because of the role played by the parameter β .

2.4.7 The Littlewood and Verrall (LV)-model

Littlewood & Verrall [50] developed the LV-model. It is similar to the L-model since it also uses a Bayesian approach and has a gamma prior distribution and an exponential model as the sample distribution. The exponential sample distribution is what the model has in common to the JM-model. The following assumptions are made:

1. Faults are corrected immediately after failure, however they are not removed with certainty so that debugging allows introduction of 'new' additional faults.
2. Failures are detected at inter-failure times $T_1, T_2, \dots$ which are statistically independent from each other.
3. The detection times for faults are the same as the inter-failure times that are described by the reliability function $R(t_i | \lambda_i) = e^{-\lambda_i t_i}$.
4. The sequence of failure rates λ_i are i.i.d. with a gamma($\alpha, \Psi(i)$) distribution function $\pi(\lambda_i | \alpha, \Psi(i)) = \frac{\Psi(i)^\alpha}{\Gamma(\alpha)} \lambda_i^{\alpha-1} e^{-\lambda_i \Psi(i)}$ where the choice for $\Psi(i)$ is the linear family $\Psi(i) = \beta_1 + \beta_2 i$.

The function $\Psi(i)$ determines the reliability growth. If $\Psi(i)$ is an increasing function² of i , it can be shown that Λ_i is a stochastic decreasing sequence in i .

From 3 and 4 above the joint distribution can be obtained as

$$\pi(t_i, \lambda_i | \alpha, \Psi(i)) = \frac{\Psi(i)^\alpha}{\Gamma(\alpha)} \lambda_i^\alpha e^{-\lambda_i(\Psi(i)+t_i)}, \quad (32)$$

which represents the situation after observing inter-failure times $t_1 \dots t_{i-1}$. The marginal reliability distribution is obtained by integrating the joint distribution over λ_i giving

$$f_T(t_i | \alpha, \Psi(i)) = \frac{\alpha \Psi(i)^\alpha}{(\Psi(i) + t_i)^{\alpha+1}}. \quad (33)$$

The reliability function is

$$R_i(t) = [\Psi(i)/(t + \Psi(i))]^\alpha, \quad (34)$$

and the failure rate is

---

² An increasing function is used to model reliability growth.

$$z(t) = \frac{f_T(t | \alpha, \Psi(i))}{R_i(t)} = \frac{\alpha}{\Psi(i) + t} \quad (35)$$

Estimates for the parameters α , β_1 and β_2 can be obtained via maximum likelihood or least squares methods.

2.4.8 Keiller and Littlewood (KL)-model

Keiller et al [52] developed a model known as the Keiller and Littlewood model. Littlewood [8] describes the model and compares it to other models. The model is essentially the same as the LV-model except that the reliability growth is introduced via the shape parameter of the gamma distribution instead of the shape parameter used in the LV-model. The assumptions here are:

1. Faults are corrected immediately after detection, however they are not removed with certainty so that debugging allows introduction of 'new' additional faults.
2. Failures are detected at inter-failure times $T_1, T_2, \dots$ which are statistically independent from each other.
3. The detection times for faults are the same as the inter-failure times described by the reliability function $R(t_i | \lambda_i) = e^{-\lambda_i t_i}$.
4. The sequence of failure rates λ_i are i.i.d. with $\text{gamma}(\alpha, \Psi(i))$ distribution function $\pi(\lambda_i | \Psi(i), \beta) = \frac{\beta^{\Psi(i)}}{\Gamma(\Psi(i))} \lambda_i^{\Psi(i)-1} e^{-\lambda_i \beta}$ where the choice for $\Psi(i)$ is the linear family $\Psi(i) = (\alpha_1 + \alpha_2 i)^{-1}$.

From 3 and 4 the marginal reliability distribution is obtained similar to (33) above in Section 2.4.7. This gives

$$f_T(t_i | \beta, \Psi(i)) = \frac{\Psi(i) \beta^{\Psi(i)}}{(\beta + t_i)^{\Psi(i)+1}} \quad (36)$$

The reliability function is

$$R_i(t) = [\beta / (\beta + t)]^{\Psi(i)} \quad (37)$$

and the failure rate is

$$z(t) = \frac{f_T(t | \beta, \Psi(i))}{R_i(t)} = \frac{\Psi(i)}{\beta + t} \quad (38)$$

Estimates for the parameters α_1 , α_2 and β can be obtained via maximum likelihood approach.

2.4.9 Weibull Order Statistics

Littlewood [8] briefly describes the Weibull order statistics as a modelling approach for software reliability. The approach is based on the set of order statistics $X_{(i)}$ that are generated by program failures. The inter-failure times T_i can be calculated by $T_i = X_{(i)} - X_{(i-1)}$. The process that generates the order statistics is assumed to have a Weibull distribution

$$f(x|\gamma, \beta) = \beta x^{\gamma-1} e^{-\beta x^\gamma} \quad (39)$$

Littlewood [8] suggests exploring Weibull order statistics as a *new* modelling technique for software reliability.

2.4.10 Non-homogenous Poisson process models

Some of the reliability models proposed are based on the non-homogenous Poisson process (NHPP). The concept of reliability growth is still being the main assumption thus the failure intensity caused by faults is decreasing. More about the Poisson process can be found in Taylor & Karlin [57] and Høyland & Rausand [58].

Let us introduce the non homogenous Poisson counting process $\{N(t), t \geq 0\}$ with expected number of failures $m(t) = E\{N(t)\}$. The process is defined by the following assumptions:

1. For any time points $t_0 = 0 < t_1 < t_2 < \dots < t_n$ the process increments $N(t_1) - N(t_0)$, $N(t_2) - N(t_1)$, ..., $N(t_n) - N(t_{n-1})$ are independent.
2. For $s \geq 0$ and $t \geq 0$, the random variable $N(s+t) - N(s)$ has the Poisson distribution $P(N(t+s) - N(s) = k) = (m(t+s) - m(s))^k e^{-(m(t+s) - m(s))} / k!$
3. $N(0) = 0$.

The intensity is obtained from the derivative of the expected number of failures

$$\lambda(t) = m'(t). \quad (40)$$

In the special case of a homogenous Poisson process, the expected number of failures is $m(t) = \text{constant} \cdot t$. In the field of software reliability it is common to assume that $\lim_{t \rightarrow \infty} m(t) = a < \infty$. This assumption corresponds to the assumption that it is finite number of faults in the program.

Goel & Okumoto [60] give a general description of how to obtain the distribution of the number of remaining faults and the software reliability for an NHPP-model. The approach is as follows: The software reliability is predicted by obtaining the distribution for $X_k = T_k - T_{k-1}$ given the time to $k-1$ failures t_{k-1}

$$\begin{aligned} P(X_k | t_{k-1}) &= P(X_k > x | T_{k-1} = t_{k-1}) \\ &= P(N(t_k) - N(t_{k-1}) = 0) = e^{-[m(t_{k-1}+x) - m(t_{k-1})]}. \end{aligned} \quad (41)$$

To obtain the distribution of the remaining faults is a little tricky. First let us introduce the variable $\overline{N}(t) = N(\infty) - N(t)$ and look at the distribution of this variable given that $N(t) = y$. The assumption $\lim_{t \rightarrow \infty} m(t) = a < \infty$ and the independence assumption of the process increments above gives

$$p(N(\infty) - N(t) = x | N(t) = y) = p(N(\infty) - N(0) = x + y). \quad (42)$$

The distribution in (42) is important for deciding when to release the program since it predicts the remaining number of faults in the software.

In the two next sub-sections we will look closer at the distribution of the number of remaining faults and the software reliability for "The exponential intensity model"

and "The logarithmic expectation model". In other words we will specify $m(t)$ and solve equations (41) and (42).

Littlewood [8] points out that software reliability increases at not deterministic times (when the fixes are done) whereas the NHPP rate changes continuously. In a sense this is a property that makes the NHPP model inappropriate.

The exponential intensity model

Goel & Okumoto [60] introduced the exponential intensity model in 1979. However, Musa [48] already introduced a similar model in the basic execution model concept in 1975. Later, Musa's concept is described as an exponential intensity Poisson model by for example Musa & Okumoto [10]. The main assumptions of Goel & Okumoto [60] are:

1. The expected number of faults has the bounds $m(t) = \begin{cases} 0, & t = 0 \\ a, & t \rightarrow \infty \end{cases}$.
2. The intensity is proportional with the expected number of remaining faults $\lambda(t) = b(a - m(t))$, where b is the proportionality constant also interpreted as the occurrence rate of an error.

Solving the differential equation in 2 (remembering that $\lambda(t) = m'(t)$) gives the intensity

$$\lambda(t) = abe^{-bt}, \quad (43)$$

and the expected number of failures

$$m(t) = a(1 - e^{-bt}). \quad (44)$$

Musa & Okumoto [10] use a slightly different parameterisation and motivation when they describe the basic execution model.

From Assumption 2 and (41) the reliability of the program is given by

$$R(x | t_{k-1}) = e^{a[\text{EXP}(-b(t_{k-1}+x)) - \text{EXP}(-bt_{k-1})]}. \quad (45)$$

From Assumption 1 and (42) the distribution of the remaining faults given that y failures have occurred is

$$p(\overline{N}(t) = x | N(t) = y) = \frac{a^{x+y}}{(x+y)!} e^{-a}. \quad (46)$$

Musa & Okumoto [10] determine the additional execution time Δt and the expected additional failures Δm to be experienced before one reach failure intensity goal $\lambda(t_F)$ from the present failure intensity $\lambda(t_P)$. From (43) the additional time can be found as

$$\Delta t = t_F - t_P = 1/b \cdot \ln[\lambda(t_P)/\lambda(t_F)]. \quad (47)$$

From (43) and (44) the expected number of failures as a function of the intensity can be obtained, which can be used to find the additional expected failures as

$$\Delta m = m(t_F) - m(t_P) = 1/b \cdot (\lambda(t_P) - \lambda(t_F)) \quad (48)$$

Musa & Okumoto [10] discuss a model for calendar time versus execution time. The model is compared with various other published models by using real data. It is concluded that this model is superior to the other software reliability models in capability and applicability. Parameter estimation and a practical example considering the Naval Tactical Data System (NTDS) data, published 1972 by Jelinsky and Moranda, are presented by Goel & Okumoto [60]. It is also concluded that the model gives good fit to the NTDS data.

The logarithmic expectation model

The logarithmic Poisson model was introduced by Musa & Okumoto [61]. It is compared with the exponential intensity model in Musa & Okumoto [10]. The logarithmic model has a failure intensity that decays exponentially with respect to the expected number of faults. The main assumption is

$$\lambda(t) = \lambda_0 e^{-\theta m(t)}, \quad (49)$$

where θ is the rate of decrease per failure. Recall that $m'(t) = \lambda(t)$ which gives

$$m(t) = 1/\theta \cdot \ln[\lambda_0 \theta t + 1]. \quad (50)$$

From (41) and (50) the reliability of the program is given by

$$R(x | t_{k-1}) = e^{\frac{1}{\theta} \ln[(\lambda_0 \theta t_{k-1} + 1) / (\lambda_0 \theta (t_{k-1} + x) + 1)]}. \quad (51)$$

Recall from the exponential intensity model above that additional execution time Δt and the expected additional failures Δm can be obtained. From differentiation of (50) the time t can be found as a function of λ which gives the additional time

$$\Delta t = t_F - t_P = 1/\theta \cdot (1/\lambda(t_F) - 1/\lambda(t_P)). \quad (52)$$

The additional expected number of faults can be found from (49) as

$$\Delta m = m(t_F) - m(t_P) = 1/\theta \cdot \ln[\lambda(t_P) / \lambda(t_F)]. \quad (53)$$

Musa & Okumoto [10] compare the logarithmic expectation and exponential intensity models. It is concluded that the logarithmic expectation model performs worse than the exponential intensity model but better than various other models. Later Musa [23] claims that the logarithmic intensity model often performs better than the exponential intensity model. Therefore he recommends trying the logarithmic model first.

The Duane (D) Model

This model originated from hardware reliability studies done by Duane in 1964. Littlewood [8] describes the model as follows: It was noticed that if the cumulative failure rate versus the cumulative testing time were plotted on log-log paper, it formed a straight line. Crow (1977) discovered that a Weibull process could model this behaviour. This process is a NHPP for which the failure intensity function has the same form as the hazard rate for the Weibull distribution i.e. kbt^{b-1} , $k, b, t > 0$.

2.4.11 The Nelson model - The Basic Data Domain model

Thayer et al [47] describe the Nelson model. The domain of input data of the software program is partitioned into a set of subsets $I = \{I_1, \dots, I_n\}$ where each element I_i contains the input data to run the program once. The program is considered as a function $P: I \rightarrow O$ as shown in Figure 2 in Section 2.1.2. Recall that an input from the subset I_f creates a software failure. The probability p that the program will cause a software failure is equal to the probability that the input I_i causes a software failure i.e. $I_i \in I_f$. If one assumes that all inputs in I are equally likely then an intuitive estimator is $p = n_f / n$ where n_f is the number of inputs in I_f . The reliability is

$$R = 1 - p = 1 - \frac{n_f}{n}. \quad (54)$$

In general, it is unlikely that every input data is equally likely so it is rather assumed that the probability of selecting input I_i a priori is p_i . Introduce the auxiliary variables

$$Y_i = \begin{cases} 0 & \text{if run with } I_i \text{ is successful} \\ 1 & \text{otherwise.} \end{cases} \quad (55)$$

This gives the failure probability

$$p = \sum_{i=1}^n p_i y_i. \quad (56)$$

If it is assumed that each run is mutually independent from other runs so that the inputs are selected according to p_i then the reliability after k -runs is

$$R(k) = (1 - p)^k = \left(\sum_{i=1}^n p_i (1 - y_i) \right)^k. \quad (57)$$

Thayer et al [47] use the Nelson model to define software reliability as *the probability that the program has no execution failure in k runs*. They also give some asymptotic conditions that allow theoretical comparison to exponential failure rate assumptions used in the JM-model. In fact they show asymptotically how an exponential model can be deduced from (57).

The Nelson model is often considered impractical for programs with large number of input sets. The model needs also a large number of computer runs to enable estimations (see Ramamoorthy & Bastani [24]). Debugging strategies often lead to situations where the inputs are not selected according to a priori probabilities. This makes the model unsuitable for debugging. However, the model should be credited for the fact that if the a priori probabilities change over time the estimate of the reliability changes accordingly.

2.4.12 A Bayesian Data Domain Model

Miller et al [65] suggest to use a Bayesian modification to Nelson's model to model software that does not reveal any failures during testing. The model is similar to the Nelsons model in (56) and (57). The model is not considered for use during debugging and should be used for validation of the software reliability later. Recall the probability of failure p in (56). The probability p can be regarded as the proportion

of software faults in the program. If the program is tested k times each test being independent of each other and the only thing being recorded is whether the program failed or succeeded an execution (or run) then the process is equivalent to a Binomial sampling model

$$f(x | p) = \binom{k}{x} p^x (1-p)^{k-x}. \quad (58)$$

The probability of failure is p in each trial and the variable X counts the number of failures. Assume that the prior of p is Beta(a, b) distributed

$$\pi(p | a, b) = \frac{p^{a-1} (1-p)^{b-1}}{B(a, b)}, \quad 0 \leq p \leq 1, \quad (59)$$

where $B(a, b)$ is the normalising constant. Using Bayes Theorem gives the beta posterior distribution

$$\pi(p | x, a, b) = \frac{p^{x+a-1} (1-p)^{k-x+b-1}}{B(x+a, k-x+b)}, \quad 0 \leq p \leq 1, \quad x = 0, 1, \dots, k. \quad (60)$$

Taking the expectation of the posterior gives the estimator

$$\hat{p} = E\{p | x, a, b\} = \frac{x+a}{k+a+b}. \quad (61)$$

The parameters a and b must also be estimated. Miller et al [65] suggest to determine them by giving the prior mean and variance of the beta distribution (59). In general it would be hard to estimate a and b since there is little experimental data.

Miller et al [65] also suggest to refine the input domain by partitioning it into sub-domains where the same modelling is used to verify the input for each sub-domain. The result from each sub-domain is combined later to give an overall prediction of the reliability.

2.4.13 The Hypergeometric Data Domain Model

Schick & Wolverton [66] provide a good discussion of software reliability models that includes a hypergeometric model. We will first briefly describe the hypergeometric distribution. The model is typically understood through the classic example of an urn model. Suppose that we have a urn filled with N balls that are identical in every way except that M are black and $N - M$ are white. If K balls are drawn at random without replacement from the population of N balls what is the probability that exactly x of the balls are black? Using the following combinatorial argument gives the answer: The total number of samples of size K that can be drawn

from the N balls is $\binom{N}{K}$. This can be accomplished in $\binom{M}{x}$ ways, leaving $\binom{N-M}{K-x}$ ways of filling out the sample with $K - x$ white balls. Let X denote the number of white balls then X is hypergeometrically distributed:

$$p(X=x) = \frac{\binom{N-M}{K-x} \binom{M}{x}}{\binom{N}{K}}. \quad (62)$$

In software reliability the model can be used to determine the number of software faults. Assume that: N is the total number of faults in the software, M is the number of "labelled faults" and that the number of labelled faults in a sample of size K is X . A fault is labelled if it is known prior to sampling. The maximum likelihood estimate of the number of faults obtained from (62) is

$$\hat{N} = \left\lfloor \frac{MK}{x} \right\rfloor. \quad (63)$$

There are several ways to label faults before sampling and two of the ways shown by Schick & Wolverton [66] are:

1. Mills' labelling approach is to first debug the software until M faults have been detected and labelled. Then a sample of K faults is taken where x is the number of labelled errors in the sample. The maximum likelihood estimate is shown in (63).
2. Seeding: Let n_1 denote the number of faults in the software before inserting (or seeding) n_2 labelled faults into the software. The total number of faults is $N = n_1 + n_2$ and the number of labelled faults is $M = n_2$. We want to determine, n_1 , the number of faults in the software prior to insertion of n_2 faults. The maximum likelihood estimate of n_1 is

$$\hat{n}_1 = \left\lfloor \frac{n_2(K-x)}{K} \right\rfloor. \quad (64)$$

To provide a stable and good estimate Schick & Wolverton [66] suggest sampling several times and compute the number of faults via the maximum likelihood estimate for each sample. Taking the mean of the maximum likelihood estimates increases the accuracy of the estimate. The method is not much used and one of the reasons is that it takes some effort to make use of the method (e.g. see Ramamoorthy & Bastani [24]). The model makes the assumption that each fault has the same probability of being detected. This suggests that each fault should have the same probability of being detected. This would not be the case if during testing one only test and verify the main functions of the program according to the operational profile. Hence one might expect the model to underestimate the number of faults in the program. The model does not use information about the failure times so that the model can not be used to evaluate the effort it takes in time to discover faults or for mean time to failure calculations. However, the model can be used to validate software reliability in well debugged programs by computing the number of faults.

2.4.14 Markov-models

There are several models mentioned in literature that concerns Markov modelling of software. Two areas in this field are presented in Villemeur [3], and they are summarised below.

Reliability of a software system

Software can be considered as a set of components. Transitions between modules are modelled via Markov and Semi-Markov state graphs. The main assumption for this approach is that the probability of calling a module from a given one depends only on these two modules. It can also be formulated that the time spent in each module is a random variable, characterised only by the source and target modules. Applying the state-space method for a software module it can be modelled as shown in Figure 7. In this Section we summarise Villemeur [3] and Thunem [36].

Figure 7: The diagram outlines the states of a software component obtained via the state-space method.

In Figure 7 the state “Software component in a inactive period” models the period in which the software component is not in use and the state “Software component in an execution period” models the period in which the software component is in use. The state “Failed Software component” models software component failures. Note that a failure can occur only while the component is executing an instruction.

Under the assumption that the times in these states are exponential one can derive a homogeneous Markov model for the components as shown in Figure 8.

Figure 8: A Markov-model of the state-space model of a software component.

The demand rate is η , the execution rate is γ and the failure rate is λ . Villemeur [3] gives the reliability prediction model for this system.

The above mentioned basic model can be extended to create a model for more components. We will use the parameters:

- γ_i : Execution rate of component i .
- λ_i : Failure rate of component i .
- q_{ij} : The probability that component j is run after component i if no failure occurs.
Note that $\sum_j q_{i,j} = 1, \forall i$.

A model for a system made up of three components is shown in Figure 9.

Figure 9: A model for a software system made up of three components.

It is easy to understand this state approach and to build up Markov-models and Villemeur [3] extends the model to a system of n components where each component is scheduled in sequence. It seems that this approach would work if only a few components are defined. For even a simple program there is such a large number of procedures that it is impractical to define each procedure as a component. It would be interesting to know about work that shows how a component is defined in order to limit the number of components.

Availability and maintainability of a software system

When software fails there are several ways to maintain it or repair it. The software can be reinitiated. If the input configuration is different from the one that caused it to fail the software will run until the next failure. The software may be debugged in an offline copy running on another computer so users can operate the software system while the system managers are debugging.

A qualitative behaviour model for maintainable software is shown in Figure 10.

Figure 10: Operating and failed state model of a software system that is being reinitiated or debugged.

The model illustrates that after a failure the program can either be reinitiated or debugged. If it is debugged the program is altered so the version S_1 that failed is not exactly the same as the new debugged version S_2 . If the software is debugged after each failure a process as illustrated in Figure 11 is created.

Figure 11: Illustration of continues debugging cycle.

Graphs as illustrated above are solved in the framework of the state space methods as indicated by Villemeur [3]. Some mathematical asymptotic formulas are also given for the model illustrated above, however no real data examples are provided.

2.4.15 Reliability and Debugging Data

A lot of models seem to exist, however practical model verification seems to be a problem. One reason is that sufficient data are missing. That does not necessarily mean that data are not gathered. Sometimes data are considered to be property and for competitive reasons data are not shared. This also happens for reliability data of hardware components. Software reliability, however, seems to be less used so it is also reasonable to think that fewer data exist. There are references to data studies and data in Bittanti [9], Thayer et al [47] and Ramamoorthy & Bastani [24]. One easily obtainable report where raw data is supplied by the McDonell-Douglas Corporation is Moranda [45].

2.5 Development Phases and Reliability Models

Software reliability models are often referenced without trying to relate applicability to the life cycle phases (outlined in Section 2.2.4). Different approaches should also be taken for different kinds and sizes of software. Ramamoorthy & Bastani [24] argue for differentiation and Figure 12 shows a classification of reliability models into various categories. Each category indicates when to apply a reliability model to a development project.

Figure 12: Classification of software reliability models

So far most effort in the development of software reliability models has been spent on finding models that apply to the development phase and maintenance phase. Models that capture reliability growth fall into this category. It is thought that these models work well for large software systems where the software reliability is measured at an

overall high level e.g. system level. The models applied to the validation phase are emphasised for software of medium size such as control software. Note that in critical software one can even risk that the software is rejected if only one fault is discovered during a validation test. The models proposed here are simple and test demanding. If robust reliability estimates are required a lot of effort has to be put into testing and data collection. More specifically the classification in Figure 12 refers to the following concepts.

The development and debugging phase: The deterministic error-counting models are the JM-model and its related models that are described in Sections 2.4.1 to 2.4.4 and the Poisson models described in Section 2.4.10. The Bayesian error-counting models are mainly developed by Littlewood and are outlined in Sections 2.4.5 and 2.4.6. The Markov error-counting models are introduced in Section 2.4.14, and there are more references in Ramamoorthy & Bastani [24] to for example models by Trivedi and Shooman.

A deterministic non-error-counting model is given in Section 2.4.3 and references to the Halden Project model are given by Ramamoorthy & Bastani [24]. Littlewood and Verrall have also introduced some Bayesian non-error counting models that are described in Sections 2.4.7 and 2.4.8.

The validation phase: The Nelson model and other data domain models apply in this phase. They are briefly introduced in Sections 2.4.11 to 2.4.13.

Operational phase models: The data domain models that are described in Section 2.4 can be applied. As a Markov approach to a process control systems, the operating environment can be viewed as consisting of a number of various distributions in which transitions do occur between the various distributions. Littlewood [71] and Cheung [75] have modelled the input distribution selection mechanism as a Markov process.

Maintenance phase models: For the maintenance phase there are only suggested data domain models as described in Section 2.4.

Correctness measures (test reliability) models: Software for critical applications has its own status. The confidence in the estimate is very important and there exist a few techniques one may apply to measure the correctness. Error seeding techniques as in Section 2.4.13 can be used to verify various test strategies. Phenomenological models are also sometimes referenced as axiomatic approaches. Some ideas are given in the introduction to Section 2.4. It is also assumed that data domain models as in Section 2.4 are useful since usually no errors are removed when the reliability is measured via correctness techniques.

2.6 Conclusions

We have given an overview of software reliability from its birth in the early seventies and covered basic disciplines of models until the beginning and up to 1995. New models arise continuously however from the models given in Section 2.4. One should be able to quickly understand new approaches since they often are minor extensions of these basic models. This is also partly the reason why we have not covered the last years of development of software reliability. Furthermore we can conclude that the

value of information approach we use in the following chapters represents a new way to think about software architecture.

In the remainder of this Section we will conclude this Chapter by discussing some various aspects of software reliability. We shall cover reliability concepts, the place for software reliability in an organisation in practise and what the models in Section 2.4 measure. Finally we shall look at some future areas where one can expect new methodology to be developed.

2.6.1 Reliability Concepts

Elsewhere in literature the failure, fault and error definitions have different interpretations than what we have used here. The definitions of these fundamental concepts are important since reliability is measured by probability of failure. We chose to use Musa's [23] definitions since they are consistent and are specifically defined for software. His definitions agree with IEC 61508 [14], however in the IEC standard they are more generally formulated. Note that the failure, fault and error concepts we have used can have other interpretations elsewhere in literature.

We shall return to the concepts of software reliability in Section 2.1 since software reliability first of all quantifies failure. After twenty-six years of research on software reliability Chillarege [80] questions what software failure is in the IEEE Transactions on Reliability editorial. There are models for software reliability, however there is still no consensus of exactly what a failure is. However our failure concept in Section 2.1.1 seems valid enough since it gives humans freedom to express failure individually and subjectively. For application software Chillarege [80] suggests prioritising the analysis of problems encountered at customer service centres in order to understand software failure better. It is not easy to define software failure. One problem is individual interpretation and different perceptions arise since different users evaluate severity differently. A new more user oriented definition should be expected in the future. For control software embedded in avionics or in space-crafts it becomes more difficult. This is for example illustrated in our control software example in Section 2.1. Only a precise requirement specification would enable us to specify software failure versus hardware failure at system level.

O'Connor [12] claims that software reliability does not exist since a program failure given certain circumstances and input data give a failure probability of unity, and whenever a failure is experienced the causing fault is removed. I think O'Connor means to say that as long as you always permanently remove the faults that causes failures you do not need a reliability model since in the end you would have software that is fault free and hence failure free. It is then a question about resources such as time and money. If one accepts that it is a question of resources to achieve failure free software then the interesting question is whether we always need 100% failure free software. If we are willing to give that up and accept software failures then software reliability engineering should be able to quantify to what degree we accept failure. Musa's [23] book on software engineering uses this reasoning. It follows that tests for reliability become tests to see whether we have achieved our predefined reliability goal.

So far our attempts to describe software failure are based on approaches that are successful for hardware. Analogies are often drawn to the assessment of reliability of hardware items that have not been used before. This is an intuitive analogy based on the fact that we do not have prior knowledge of a specific new software design in

operation just as we do not have prior knowledge of new hardware that has not been used before. The best information is obtained when the software or hardware is being used during normal operation. However that is not always possible before the system is fully operational. The second best we can do is to simulate operation via testing. Based on test information and expert judgement one can attempt to generalise and predict how the system operates during normal operation.

2.6.2 Software Reliability Practise

Today software reliability issues often belong under the quality assurance umbrella where it is given attention as one important characteristic for software engineering (see Section 2.2). Software quality assurance is in many ways the multidisciplinary effort to improve not only the software product itself, but also the entire process of making it. It is largely believed that a clear, easy and well-defined development process in the end improves the product. Adoption of new technology and standards are essential in order to follow market demands.

It is my feeling that software reliability models are generally not used in practical applications even though they are being considered and researched via pilot and experimental projects. There are various reasons why this is the case, and I would like to emphasise three such reasons: Primary the software reliability models as outlined in Section 2.4 do not help the programmer to find any of the faults that are in the code. Secondly the prediction models do not cover aspects such as evaluation of the soundness of the overall design, and finally they do not help the programmer to get a better understanding of how the software should be correctly coded. For software projects that have to meet certain cost limits and deadlines it is difficult to convince managers to spend time and money on software reliability modelling. The managers do not have a direct feeling of the benefits and from the experimental projects and in literature it is lack of hard evidence of direct benefits. The latest version of the IEC 61508 [14] standard recommends statistical methods for the higher safety integrity levels. It is pointed out that the system or program should be tested with input data selected according to the expected statistical distribution of real operating inputs which is called the operational profile (see also Musa [23]). It is important for software reliability modelling that the IEC standard acknowledges quantification of software reliability for safety purposes.

2.6.3 Reliability Models – What do they Measure

Many later books on software reliability attempts to cover the models developed over the last 20 years. Many models developed from mid seventies to mid eighties are the basis for how reliability is estimated for software. The models are often quite similar and have evolved from just a few concepts by adding additional assumptions or using Bayesian extensions. We have summarised various models in Section 2.4. There exists little consensus for an overall model and many authors recommend trying out several models before choosing a particular one for a particular set of reliability data. Research focused on the issue of finding one best model has so far only given inconclusive results and different authors recommend different models. It seems that several experts in the field agree that at least 2 or 3 models should be compared and the best one chosen (see Musa [23] and Littlewood [8]). It should also be emphasised that it might be useful to apply different models for different phases in the lifecycle. Ramamoorthy & Bastani [24] have proposed a system where various models are suggested for various development phases as discussed in Section 2.5.

Most of the models in software reliability assume the concept of reliability growth. This originates from the idea that when a software failure has occurred the fault causing the failure will be repaired. Hence, the overall software reliability of the computer system increases. This scenario is only useful as long as the software is being repaired and if the fault can be permanently removed. This assumption has analogies to the bathtub curve that is often referred to for hardware reliability. The bathtub curve reflects parts of the lifecycle for a product where it is experienced that systems or components have a burn in period where the failure intensity is decreasing before the intensity becomes constant during the systems operational time. At the end of the operational time one often experiences increasing failure intensity as the components are ageing. Software however does not age so the ageing process is not a factor. It is mostly assumed that the software improves its reliability over time due to fault removal. This is similar to the burn in time that is described in the bathtub curve.

To measure software reliability is not just to gather data from testing and plunge them into a software reliability model and use whatever predictions come out of the model. The tests need to be carefully planned according to the operational profile (e.g. see IEC 61508 [14] and Musa [23]) that ensures that the tests represent the real use of the software as closely as possible. Clearly a test team that exhaustively test the extreme workloads of the software to see what failures exist under extreme conditions will experience different reliability than a test team that test the software under normal operating conditions. Chen et al [37] give an example that shows the impact of test techniques on the estimation of reliability.

Lalli & Malec [2] claim that there is a “missing link” between software failure prediction and consequence prediction. They emphasise that there are several methods to quantify the occurrence of software failures while there are few that relate failures to consequences or severity. It is also my impression that severity of software failure often is not emphasised when software reliability models are published. However, when data is collected one should address consequences and one should categorise each failure in a severity classification system. We adopted Musa’s [23] severity classification system in Section 2.1.1. A distinct severity classification system should be consistent with the consequences that are experienced. If this consistency is ensured then severity classification enables consequence prediction. It should be emphasised that the early papers on software reliability were mostly written to show whether software reliability data could be statistically modelled and early papers do often not address consequences or severity of failures.

Lalli & Malec [2] believe that there is a paucity of public reports showing the successful applications of software reliability models. They believe that reports are often classified by the companies that successfully develop or use them and that successful reliability models are important for the competitiveness of the companies because they can measure and control their products. It is however my belief that such techniques have so far been too expensive, and that the management has not seen any direct benefits of knowing the reliability of their software. Management has often lacked confidence in the correctness of the predictions. This is supported since the results of pilot studies often are rather academic and come long after the software has been designed, or too late to have any practical value for the management.

2.6.4 Future Control Software Reliability Modelling

Twenty-six years of software reliability is indeed a short time, and to see it in perspective think about the law of gravity. Computers and their software are a part of a totally human created system, nevertheless we are searching to describe the process mathematically to increase understanding and to improve the engineering process.

At the ESREL'99 conference in Munich I participated in a round table discussion on "Significance of Process Attributes for Software Reliability". During this discussion existing software reliability models were summarised and it was questioned why the models are not more used in practise. It was concluded that this is somewhat illogical in comparison to modelling of hardware reliability. One should however remember that the hardware architecture often is analysed via fault-tree analysis or other techniques that gives confidence that the design is properly covered. This is somewhat different for software since the models consider software as a black box and do not question its architecture. Therefore I posed the question: "Why has not more effort been spent on development of methods that are able to quantify the decision structure in the control flow diagrams?" The reason for my question is that existing models measure software reliability after the code is generated via black box testing and I have not encountered mathematical methods that have attempted to describe or quantify the internal decision structure of the black box in a systematic manner. Existing methods talk about writing up the software requirements and to describe what the software shall do as clearly as possible. Afterwards the software is coded and we verify that the system behaves as required. We never question whether the required architecture represents a sound rational decision process that uses information optimally. It is my feeling that when we are able to model the decision structure for control software in a rational manner one would also be able to speak about measuring software reliability in a more fruitful and precise manner than current approaches are doing. As was pointed out during our discussion it does not help to validate software or measure software reliability if the overall architecture quality is bad.

In the remaining Chapters we consider decision analysis and attempt to use it to model rational decision making for control-software. This is to be interpreted as an attempt to quantify control flow diagrams being developed typically in an architectural design phase before the software is coded. It is my hope that improved mathematical engineering at an early stage can help engineers to design better software architecture, and enable more automatic code generating algorithms in the future, as well as give a well balanced system design for control systems.

2.7 References

- [1] M. R. Lyu: 'Handbook of Software Reliability Engineering', IEEE Computer Society Press, McGraw-Hill, 1996.
- [2] V. R. Lalli, H. A. Malec, "Reliability Training", NASA Reference Publication 1253, 1992.
- [3] Alain Villemeur: "Reliability, Availability, Maintainability and Safety Assessment", John Wiley & Sons, W. Sussex 1992.
- [4] Editors N. Fenton, R. Whitty and Y. Iizuka: "Software Quality Assurance and Measurement, A worldwide Perspective". International Thomson Computer Press, 1995.
- [5] T. Gilb: "Principles of software engineering management". Addison-Wesley, 1988.
- [6] F. J. Redmil (eds): "Dependability of Critical Computer Systems 2", Elsevier, 1989.
- [7] P. G. Bishop (eds): "Dependability of Critical Computer Systems 3", Elsevier, 1990.
- [8] B. Littlewood: "Forecasting Software Reliability", In S. Bittanti (eds), "Software Reliability Modelling and Identification", Springer-Verlag, Berlin 1988.
- [9] S. Bittanti: "Forecasting Software Reliability", Springer-Verlag, Berlin 1988.
- [10] J. Musa, K. Okumoto: "Application of Basic and Logarithmic Poisson Execution Time Models in Software Reliability Measurement", In S. Bittanti (eds), "Software Reliability Modelling and Identification", Springer-Verlag, Berlin 1988.

- [11] S. Bittanti et al: "A Flexible Modelling Approach for Software Reliability Growth", In S. Bittanti (eds), "Software Reliability Modelling and Identification", Springer-Verlag, Berlin 1988.
- [12] P. D. T. O'Connor: "Practical Reliability Engineering", Wiley, 3rd-ed 1994, pp. 233-254.
- [13] R. S. Pressman: "Software Engineering, A practitioner's approach", McGraw-Hill, London, Third Edition (European Adaptation), 1994.
- [14] IEC 61508: "Functional Safety of electrical/electronic/programmable electronic safety related systems", International Electrotechnical Commission, 1998.
- [15] 'Space Project Management, Project Phasing and Planning', European Cooperation for Space Standardisation (ECSS), ECSS-M-30A, April, 1996.
- [16] Report by the Inquiry Board: "Ariane5, Flight 5.01 Failure", Chairman of the board Prof. J. L. Lions, Paris 19 July 1996.
- [17] E. Yourdon: 'Techniques of Program Structure and Design', Prentice-Hall, 1975.
- [18] M. L. Shooman: 'Software Engineering-Design, Reliability, Management', McGraw-Hill 1983.
- [19] T. Anderson, P. A. Lee: 'Fault Tolerance Principles and Practice', Prentice Hall International, 1981.
- [20] J. Kelly, A. Avizienis: "A specification oriented multi-version failure software experiment digest", 13'th Int. Symposium on Fault Tolerant Computing, Milan, 1983, pp 120-6.
- [21] T. Anderson et al.: "An Evaluation of Software Fault Tolerance in a practical System Centre for Software Reliability", In Digest of the 15'th Int. Symp. on Fault Tolerant Computing, Ann Arbor, 1985.

- [22] T. Takano, T. Yamada, K. Shutoh, N. Kanekawa: "In-Orbit Experiment on the Fault-Tolerant Space Computer Aboard the Satellite Hiten", IEEE Trans. On Reliability, vol. 45, no. 4. 1996, pp 624-631.
- [23] J. Musa: "Software Reliability Engineering", McGraw-Hill, 1998.
- [24] C. V. Ramamoorthy, F. B. Bastani: "Software Reliability - Status and Perspectives", IEEE Trans. Software Eng., vol SE-8, no. 4. Jul. 1982, pp 354-371.
- [25] A. L. Goel: "Software Reliability Models: Assumptions, Limitations, and Applicability", IEEE Trans. Software Eng., vol. SE-11, NO. 12, 1985, pp 1411-1423.
- [26] "Sneak analysis methods and procedures for ESA space programmes", ESA PSS-01-411, Issue 1, January 1994.
- [27] J. G. Norstrøm: "Sneak Path Analysis for Computer Controlled Systems", ESTEC Working Paper No: EWP 1871, December 1995.
- [28] B. Dore, J.G. Norstrøm: "Pilot Application of Sneak Analysis on Computer Controlled Satellite Equipment", In P.C Cacciabue & I. A. Papazoglou (eds), "Probabilistic Safety Assesment and Management", Springer, 24-28 June, 1996, pp 1590-1596.
- [29] A. Jalashgar: "Application of Functional Modelling Methodologies to Failure Analysis of Process Control Systems", In P.C Cacciabue & I. A. Papazoglou (eds), "Probabilistic Safety Assesment and Management", Springer, 24-28 June, 1996, pp 1054-1060.

- [30] "Software/Hardware Integrated Critical Path Analysis (ICPA)", In Proc. of the Fourth International System Safety Conference, 1979, pp 47-49.
- [31] "Sneak Circuit Analysis", Department of the US navy, NAVSO P-3634, August 1987.
- [32] C. P. Whetton: "Sneak Analysis Of Process Systems", Whetton, Trans IChemE, Vol 71, Part B, August 1993.
- [33] D. L. Buratti, W.E. Pinkston, R.O. Simikins: "Sneak Software Analysis", 4th Digital Avionics Systems Conference, November 1981.
- [34] P.T. Ward, S.J. Mellor: "Structured development for real-time systems", Vol. 1, Vol 2, Yourdon, January 1985.
- [35] "Clue list for Sneak Analysis", ESA STM-252, Issue 1, December 1993.
- [36] H. Thunem: "Review of software testing and reliability assessment methods", OECD Halden Reactor Project, HWR-425, 1995.
- [37] M. H. Chen, A. P. Mathur, V. J. Rego: "Effect of Testing Techniques on Software Reliability Estimates Obtained Using A Time-Domain Model", IEEE Trans, Rel., vol 44, no. 1, 1995, pp 97-103.
- [38] A. Thakur, R. K. Iyer: "Analyse-Now – An Environment for Collection & Analysis of Failures in a Network of Workstations", IEEE Trans, Rel., vol 45, no. 4, 1996, pp 561-570.
- [39] Z. Jelinsky, P. B. Moranda: "Application of a probability based method to a code reading experiment", In Record 1973: IEEE Symposium on software computer software reliability, IEEE, New York 1973 pp. 78-82.
- [40] M. L. Shooman: "Probabilistic Models for Software Reliability Predicion", In Record 1973: IEEE Symposium on

software computer software reliability, IEEE, New York 1973
pp. 51-57.

- [41] J. Angus, R. E. Schafer and A. Sukert: "Software Reliability Validation", Proceedings Annual Reliability and Maintainability Symposium, San-Francisco, 1980, pp 191-199.
- [42] G. J. Shick and R. W. Wolverton: "Assessment of software reliability", Proceedings, Operation Research, Physica-Verlag, Wurzburg-Wien, 1973, pp. 395-422.
- [43] G. J. Shick and R. W. Wolverton: "An Analysis of Competing Software Reliability - Status and Perspectives", IEEE Trans. Software Eng., SE-4 (2), 1978, pp. 104-120.
- [44] P. B. Moranda: "Prediction of Software Reliability During Debugging", In Proceedings, Annual Reliability and Maintainability Symposium, 1975, pp 327-332.
- [45] P. B. Moranda: "Quantitative Methods For Software Reliability", McDonell-Douglas Corp, Rep. no. AD/A-0-35 585, 1976.
- [46] M. Lipow: "Estimation of Software Package Residual Errors", TRW Software Series and RTW -55-72-09, Redondo Beach, CA, 1972
- [47] T. A. Thayer, M. Lipow and E. C. Nelson: "Software Reliability: A study of a Large Project Reality", North Holland 1978.
- [48] J. D. Musa: "A theory of software reliability and its applications", IEEE Trans, Software Eng., vol SE-1, no. 3. Sept. 1975, pp 312-327.
- [49] I. Mijamoto: "Software Reliability in Online Real Time Environment", International Conference on Reliable Software, IEEE/ Automation Industries, Inc., Silver Spring. MD, 1975, pp. 518-527.

- [50] B. Littlewood and J. L. Verral: "A Bayesian Reliability Growth Model for Computer Software", Royal Statis. Soc., C (Applied Statistics), 2, 1973, pp 332-346.
- [51] B. Littlewood, J. L. Verrall: "Likelihood Function of a Debugging Model for Computer Software Reliability", IEEE Trans, Rel., vol R-30, no. 2, Jun. 1981, pp 145-148.
- [52] P. A. Keiller, B. Littlewood, D. R. Miller and A. Sofer: "Comparison of software reliability predictions", Digest FTCS 13 (13'th International Symposium on Fault-Tolerant Computing), 1983, pp 128-134.
- [53] B. Littlewood: "Stochastic Reliability-Growth: A Model for Fault-Removal in Computer-Programs and Hardware-Designs", IEEE Trans. Reliability, vol R-30, 1981 Oct, pp 313-320.
- [54] B. Littlewood and A. Sofer: "A Bayesian modification to the Jelinsky-Moranda software reliability growth model" IEE/BCS, Software Engineering J, Vol2, Pt.2, March 1987, pp 30-41.
- [55] M. Abramowitz, I. A. Stegun: "Handbook of Mathematical Functions", Dover, 1965.
- [56] G. Casella and R. L. Berger: "Statistical Inference", Wadsworth & Brooks, Inc., Belmont, 1990.
- [57] H. M. Taylor, S. Karlin: "An Introduction to Stochastic Modelling", Academic Press, 1984, pp. 173-178.
- [58] A. Høyland, M. Rausand: "System Reliability Theory, Models and Statistical Methods", Wiley, Series in Probability and Mathematical Statistics, 1994.
- [59] N. F. Schneidewind: "Analysis of Error Processes in Computer Software ", In Proceedings, International Conference on Reliable Software, Los Angeles, pp. 337-346.

- [60] A. L. Goel, K. Okumoto: "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures", IEEE Trans. Reliability, vol R-28, 1979 Aug, pp 206-211.
- [61] J. D. Musa, K. Okumoto: "A logarithmic Poisson execution time model for software reliability measurement", Proc. 7th International Conf. on Soft. Eng., Orlando, 1984 Mar, pp 74-82.
- [62] M. Kaaniche, K. Kanoun, M. Cukier, M. Bastos Martini: "Software Reliability Analysis of Three Successive Generation of a Switching System", First European Conference on Dependable Computing, Berlin, Germany, 1994, pp 473-490.
- [63] M. Kaaniche, K. Kanoun: "Reliability of a Commercial Telecommunications System", In Proc. 7th Symposium on Software Reliability Engineering, USA, 1996, pp 207-212.
- [64] S. Brocklehurst, P. Y. Chan, B. Littlewood, J. Snell: "Recalibrating Software Reliability Models", IEEE Trans. Reliability, VOL. 16, NO. 4, 1990, pp 458-469.
- [65] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill and J. M. Voas: "Estimating the Probability of Failure When Testing Reveals No Failures", IEEE Trans, Software Eng., VOL. 18, NO. 1. 1992, pp 33-42.
- [66] G. J. Schick and R. W. Wolverton: "An Analysis of Competing Software Reliability Models", IEEE Trans, Software Eng., VOL. SE-4, NO. 2, 1978, pp 104-120.
- [67] Y. Tohma, K. Tokunaga, S. Nagase and Y. Murata: "Structural Approach to the Estimation of the Number of Residual Software Faults Based on The Hyper-geometric Distribution", IEEE Trans. Software Eng., VOL. 15, NO. 3, 1989, pp 345-355.
- [68] Y. Tohma, H. Yamano, M. Ohba and R. Jacoby: "The Estimation of Parameters of the Hypergeometric Distribution and Its Application to the Software Reliability Growth

Model", IEEE Trans. Software Eng., VOL. 17, NO. 3, 1991, pp 483-489.

- [69] R. H. Huei Hou, S. Kuo, Y. Chang: "Optimal Release Policy for Hyper-Geometric Distribution Software-Reliability Growth Model", IEEE Trans. Reliability, VOL. 45, NO. 4, 1996, pp 646-651.
- [70] D. Avresky, J. Arlat, J.C. Laprie, Y. Crouzet: "Fault Injection for Formal Testing of Fault Tolerance", IEEE, Trans. Rel, Vol 45, NO 3, 1996, pp 443-455
- [71] B. Littlewood: "A Reliability Model for Systems with Markov Structure", Applied Statistics (J. Royal Statist. Soc., Series C) vol 24, 1975, pp. 172-177.
- [72] B. Littlewood: "Software Reliability Model for Modular Program Structure", IEEE, Trans. Rel, Vol R-28, NO 3, 1979, pp 241-246.
- [73] A. L. Goel, K. Okumoto: "A Markovian model for reliability and other performance measures of software systems", In Proc. National Computer Conference, 1979, pp 769-774.
- [74] A. Costis, C. Landrault, J.C. Laprice: "Reliability and Availability Models for Maintained Systems Featuring Hardware Failures and Design faults", IEEE Trans. Comput., vol. C-27, 1978, pp 548-560.
- [75] R. C. Cheung: "A user-oriented software reliability model", in Proc. COMPSAC 1978 Chicago, IL., Nov. 1978, pp. 565-570.
- [76] Commemorative Issue of Dr. M. H. Halstead, IEEE Trans. Software Eng. vol. SE-5, NO. 2, 1979.
- [77] L. Cornell, M.H. Halstead: "Predicting the number of bugs expected in a program module", Purdue University, CSD TR-202, 1976.
- [78] W. Feller: 'An Introduction to Probability Theory and its Applications', Joh Wiley & Sons, Inc., 1970.

- [79] C. R. Rao: "Linear Statistical Inference and Its Applications", Joh Wiley & Sons, Sec. edition, 1973.
- [80] R. Chillarege: "What Is Software Failure ?", IEEE Trans on Reliability, vol. 45, no. 3, 1996, pp 354-355.
- [81] B. W. Kernighan, P. J. Plauieger: 'The elements of Programming Style', McGraw-Hill, 1974.
- [82] E. A. Davis, P. K. Giloth: "Performance Objectives and service Experience", Bell Syst. Tech. J., vol. 60, no 6, July-Aug 1981.
- [83] J. E. Angus, L. E. James: "Combined Hardware/Software Reliability Models", Annual Reliability and Maintainability Symposium, IEEE, New York, 1982, pp. 176-181.
- [84] D. J. Smith, K. B. Wood: "Engineering Quality Software", Elsevier, 2nd ed., 1989.
- [85] R. Aarø, L. Bodsberg, P. Hokstad: "Reliability prediction handbook; computer-based Process Safety systems", SINTEF, Safety and Reliability, 1989.
- [86] S. Guarro, M. Yau, M. Motamed: "Development of Tools for Safety Analysis of Control Software in Advanced Reactors", NUREG/CR-6465, 1996.
- [87] ESA: "Abstracts", Product Assurance Symposium and Software Product Assurance Workshop, ESTEC, 19-21 March 1996, The Netherlands.
- [88] ESA: "Guide to software reliability & safety assurance for ESA space systems", Draft 2 of ESA PSS-01-213 Issue 1, ESA, Product Assurance & Safety Department, Aug. 1993.
- [89] V. R. Basili: "Tutorial on Models and Metrics for Software Management and Engineering"; IEEE Computer Society Press, 1980.

- [90] A. J. Perlis, F. G. Sayward, M. Shaw: 'Software Metrics: An analysis and Evaluation', MIT Press, 1981.
- [91] E. N. Adams: "Optimising preventive service of software products", IBM Journal of Research and Development, Vol R-28, 1979, June pp. 103-110.
- [92] P. M. Nagel, J. A. Skrivan: 'Software reliability: repetitive run experimentation and modelling', BCS-40399, Boeing Computer Services Company, Seattle, Washington, December 1981.

Chapter 3

Decision Theoretic Background

In this chapter basic notation will be defined and the mathematical foundation needed to discuss Savage's decision theory will briefly be introduced (see for example Savage [1], Cooke [2] and Tucker [3]).

In Section 3.1 we will use the following notation. S is the non-empty set of all states of the world. T is a vector of tests $T = (T_1, \dots, T_n)$; $T_i : S \rightarrow \tau_i$. Later in Section 3.2 we expand the notation to include the finite set of all consequences (states of an acting subject) which is denoted by C and the set of all acts F , that is the set of functions from S to C . We speak about preference relation on acts which is denoted by a partial ordering " $\leq$ " and the utility a function $U: C \rightarrow \mathbb{R}$.

A state is a single and complete description of the world. The set of states (or equivalently the set of possible worlds) is the set that contains all possible single states. An element of S (that is a single state) is normally denoted s . In many applications the description of the possible worlds will be truncated and the possible worlds will only be described up to a predefined level. *An event is a subset of the states S .* Note that if A is an event it might as well be thought of as the *proposition* asserting that event A occurs. This is equivalent to saying that *the real world belongs to A* . Equivalently, A occurs if the realised state s satisfies $s \in A$. The empty set $\emptyset$ corresponds to an event that is not possible in any state (or world). A set of subsets of S is called a field if it contains the empty set $\emptyset$.

Definition 1 (Field $\mathcal{F}$)

A set $\mathcal{F}$ of subsets of S is a field of subsets of S if

- i. $\emptyset \in \mathcal{F}$
- ii. If $A \in \mathcal{F}$ and $B \in \mathcal{F}$ then $A \cup B \in \mathcal{F}$
- iii. If $A \in \mathcal{F}$ then $\neg A \in \mathcal{F}$

■

The pair $(S, \mathcal{F})$ is called a measurable space. It is sometimes convenient to assume that $\mathcal{F}$ is closed under countable unions, in which case $\mathcal{F}$ is called a *sigma field*.

Definition 2 (Sigma field)

A set of subsets $\sigma \subset S$ is called a sigma field if:

- i. For every A , if $A \in \sigma$, then also $\neg A \in \sigma$.
- ii. If $A_1, A_2, \dots, A_n, \dots$ is a countable sequence of elements of σ , then $\cup A_n \in \sigma$.
- iii. $\emptyset \in \sigma$.

From this it can be seen that the definition of a sigma field is more restrictive than that of a field. The closure under countable unions is required to define a countable additive probability measure on the sigma field.

Definition 3 (Probability)

A probability p on a measurable space (S, σ) is a function $p: \sigma \rightarrow [0, 1]$ satisfying:

- i. $\forall A \in \sigma, 0 \leq p(A) \leq 1$
- ii. $p(S) = 1$
- iii. If $\{A_n\}$ is any denumerable union of disjoint events in σ , then

$$p(\cup A_n) = \sum_{n=1}^{\infty} p(A_n).$$

The object (S, σ, p) is called a probability space.

Definition 4 (Smallest sigma field)

Let ν be a collection of subsets of S . The smallest sigma field containing ν or the sigma field generated by ν is denoted, $\sigma(\nu)$ or $\sigma\{\nu\}$, is the intersection of all sigma fields containing ν .

Definition 4 allows us to speak about the sigma fields generated by a random variable. This will be important in the modelling later. But first let us define what a random variable is.

Definition 5 (Random Variable)

A random variable X is a real-valued function on S which is σ -measurable, that is for every real number x , $\{s \in S \mid X(s) \leq x\} \in \sigma$.

Definition 5 ensures that X is measurable with respect to σ . We use the notation

$$[X \leq x] = \{s \in S \mid X(s) \leq x\},$$

and in general for any Borel set B we write

$$[X \in B] = \{s \in S \mid X(s) \in B\}.$$

Definition 6

If X is a random variable, the sigma field induced by X , $\sigma\{X\}$, is defined by

$$\sigma\{X\} = \sigma\{[X \leq x], -\infty < x < \infty\}.$$

■

REMARK: The measurability of real valued functions will always be taken with respect to the Borel sigma-algebra unless stated otherwise. Note that B is a Borel set if B can be obtained by a countable number of operations, starting from open sets, each operation consisting in taking unions, intersections or complements (see Rudin [4]).

3.1 Some Properties of Events and Observations

In the remainder of this chapter an observation or a test T is a random variable taking a countable number of values. We will explore the impact the observations have on conditional probabilities. Here we will investigate self-conditional statements as: "The expected probability of a false negative prediction, $E\{p(\neg A)|T\}$ given any observation $T = t$ ". We use $A \perp B$ to denote that event A is independent of event B (note that events A and B are independent if $p(A \cap B) = p(A) \cdot p(B)$). Further $A \perp T_2 \mid T_1$ means that given test outcome T_1 then A is conditionally independent of T_2 .

LEMMA 1 (Conditionally Constant & Statistically Independent)

$p(A \mid T = t)$ is constant $\forall t \in \tau$ is equivalent to $A \perp T$, that is A is statistically independent of T .

PROOF: Put $p(A \mid T = t) = \alpha$. Then

$$p(A) = \sum_{t \in \tau} p(A \mid T = t) p(T = t) = \alpha \sum_{t \in \tau} p(T = t) = \alpha = p(A \mid T = t).$$

■

THEOREM 1 (Self-Conditional Expectation)

- i. $E\{p(A \mid T) \mid A\} \geq p(A)$ and equality holds if and only if $A \perp T$.
- ii. $E\{p(A \mid T) \mid \neg A\} \leq p(A)$ and equality holds if and only if $A \perp T$.

PROOF: First we will prove (i). We have that

$$\begin{aligned} E\{p(A \mid T) \mid A\} &= \sum_{t \in \tau} p(A \mid T = t) p(T = t \mid A) \\ &= \frac{1}{p(A)} \sum_{t \in \tau} p^2(A \mid T = t) p(T = t). \end{aligned}$$

Since the quadratic function is convex Jensen's inequality gives

$$\sum_{t \in \tau} p^2(A|T=t)p(T=t) \geq \left(\sum_{t \in \tau} p(A|T=t)p(T=t) \right)^2 = p^2(A).$$

Simplifying gives

$$\frac{1}{p(A)} \sum_{t \in \tau} p^2(A|T=t)p(T=t) \geq p(A).$$

Equality holds in Jensen's inequality if and only if $p^2(A|T=t)$ is linear as a function of t . Since the quadratic function is nowhere linear this only holds when $p(A|T=t)$ is constant as a function of t and by Lemma 1 it follows that $A \perp T$.

Proof of (ii): Let $B = \neg A$. Apply part (i) on B to get

$$p(\neg B) = 1 - p(B) \geq 1 - E\{p(B|T)|B\} = E\{1 - p(B|T)|B\} = E\{p(\neg B|T)|B\}.$$

■

If $T \in \{0, 1\}$ and $T = 1$ "predicts" A then Theorem 1(i) says that the probability of a "true positive" prediction for A is greater than or equal to probability of A . Similarly Theorem 1(ii) says that the probability of a "false positive" prediction is less or equal to the probability of A .

COROLLARY 1

Let A be an event and T be a test, then

$$1 \geq E\{p(A|T)|A\} \geq p(A).$$

The upper bound holds when the test detects the event A with certainty ($\forall t \in \tau$, $p(A|T=t) = \{0 \text{ or } 1\}$) and the lower bound holds when $A \perp T$.

PROOF: The lower bound is established when equality holds in Theorem 1. Since $p(A|T=t) \leq 1$ we have that

$$E\{p(A|T)|A\} = \sum_{t \in \tau} p(A|T=t)p(T=t|A) \leq \sum_{t \in \tau} p(T=t|A) = 1.$$

and equality holds if and only if $\forall t \in \tau, p(A|T=t) = \{0 \text{ or } 1\}$.

■

Now $E\{p(A|T_1, T_2)|A\}$ will be compared with $E\{p(A|T_1)|A\}$.

REMARK: Bayes Theorem gives:

$$p(T_1 = t_1, T_2 = t_2|A) = p(T_1 = t_1|A, T_2 = t_2)p(T_2 = t_2|A)$$

and

$$p(A|T_1 = t_1, T_2 = t_2) = \frac{p(T_1 = t_1, T_2 = t_2|A)p(A)}{p(T_1 = t_1, T_2 = t_2)}.$$

Let us look at an example where $p(A) = p(\neg A) = \frac{1}{2}$ and for $i = 1, 2$, $p(T_i = 0 | A) = p(T_i = 0 | \neg A) = \frac{1}{2}$ and $T_i \in \{0, 1\}$. Further it will be assumed that given A then $T_1 \perp T_2$ and given $\neg A$ then $T_1 = T_2$. This gives the following probabilities:

$$p(T_1 = 0, T_2 = 0 | \neg A) = p(T_1 = 1, T_2 = 1 | \neg A) = \frac{1}{2},$$

$$p(T_1 = 1, T_2 = 0 | \neg A) = p(T_1 = 0, T_2 = 1 | \neg A) = 0$$

and

$$p(T_1 = 0, T_2 = 0 | A) = p(T_1 = 1, T_2 = 1 | A) = \frac{1}{4},$$

$$p(T_1 = 1, T_2 = 0 | A) = p(T_1 = 0, T_2 = 1 | A) = \frac{1}{4}.$$

This gives

$$p(T_1 = 0, T_2 = 0) = p(T_1 = 1, T_2 = 1) = \frac{3}{8}$$

$$p(T_1 = 1, T_2 = 0) = p(T_1 = 0, T_2 = 1) = \frac{1}{8}$$

Taking the conditional expectation gives

$$E\{p(A|T_1, T_2) | A\} = p(A) \sum_{t_1 \in \tau_1, t_2 \in \tau_2} \frac{p^2(T_1 = t_1, T_2 = t_2 | A)}{p(T_1 = t_1, T_2 = t_2)} = \frac{2}{3} \geq p(A).$$

Similarly we have that

$$E\{p(\neg A|T_1, T_2) | \neg A\} = \frac{2}{3} \geq p(\neg A)$$

and

$$E\{p(\neg A|T_1) | \neg A\} = E\{p(A|T_1) | A\} = \frac{1}{2}.$$

Hence, one should expect that

$$E\{p(A|T_1, T_2) | A\} \geq E\{p(A|T_1) | A\}$$

holds.

THEOREM 2 (Two tests at least as good as one)

Let T_1 and T_2 be random variables, then

$$E\{p(A|T_1, T_2) | A\} \geq E\{p(A|T_1) | A\}$$

and equality holds if and only if $A \perp T_2 | T_1$.

PROOF: From the Remark above we have that

$$E\{p(A \mid T_1, T_2) \mid A\} = \frac{1}{p(A)} \sum_{t_1 \in \tau_1, t_2 \in \tau_2} \frac{p^2(A, T_1 = t_1, T_2 = t_2)}{p(T_1 = t_1, T_2 = t_2)}$$

and also

$$E\{p(A \mid T_1) \mid A\} = \frac{1}{p(A)} \sum_{t_1 \in \tau_1} \frac{1}{p(T_1 = t_1)} p^2(T_1 = t_1, A).$$

It suffices to show that for all $t_1 \in \tau_1$,

$$\sum_{t_2 \in \tau_2} \frac{p^2(A, T_1 = t_1, T_2 = t_2)}{p(T_1 = t_1, T_2 = t_2)} \geq \frac{p^2(T_1 = t_1, A)}{p(T_1 = t_1)} = \frac{\left(\sum_{t_2 \in \tau_2} p(A, T_1 = t_1, T_2 = t_2) \right)^2}{\sum_{t_2 \in \tau_2} p(T_1 = t_1, T_2 = t_2)} \quad (65)$$

Take $d_i(t_1) = \sqrt{p(T_1 = t_1, T_2 = i)}$ and $c_i(t_1) = p(A, T_1 = t_1, T_2 = i)/d_i(t_1)$ where $i \in \tau_2$ contains the elements $\{1, \dots, n\}$. Substituting $d_i(t_1)$ and $c_i(t_1)$ into (65) gives

$$\sum_{i \in \{1, \dots, n\}} c_i^2(t_1) \geq \frac{\left(\sum_{i \in \{1, \dots, n\}} c_i(t_1) d_i(t_1) \right)^2}{\sum_{i \in \{1, \dots, n\}} d_i^2(t_1)}, \text{ or equivalently}$$

$$\sum_{i \in \{1, \dots, n\}} d_i^2(t_1) \sum_{i \in \{1, \dots, n\}} c_i^2(t_1) \geq \left(\sum_{i \in \{1, \dots, n\}} c_i(t_1) d_i(t_1) \right)^2.$$

This is the Cauchy-Schwartz inequality. The Cauchy-Schwartz inequality holds with equality if and only if $c_i(t_1) = \beta(t_1) d_i(t_1)$, that is

$$p(A \mid T_1 = t_1, T_2 = i) = \beta(t_1) \cdot p(T_1 = t_1, T_2 = i)$$

for all $i \in \{1, \dots, n\}$ and some constant $\beta(t_1)$. Simplifying gives

$$\beta(t_1) = p(A \mid T_1 = t_1, T_2 = i).$$

Hence for $\beta(t_1)$ to be constant, $p(A \mid T_1 = t_1, T_2 = i)$ must be constant for all i . From Lemma 1 $\beta(t_1)$ is constant for all t_1 if and only if $A \perp T_2 \mid T_1$. ■

In Corollary 2 we will try to understand Theorem 2 better.

COROLLARY 2

Assume $A \perp T_2 \mid T_1$. Then $A \perp T_1 \mid T_2$ if and only if $A \perp T_1$ and $A \perp T_2$.

PROOF: Assume that $A \perp T_2 \mid T_1$ holds. We have that

$$p(A \cap T_2 \mid T_1) = p(A \mid T_1) \cdot p(T_2 \mid T_1).$$

If $A \perp T_1 \mid T_2$ then:

$$p(A \cap T_1 | T_2) = p(A | T_2) \cdot p(T_1 | T_2). \quad (66)$$

Then

$$p(A \cap T_2 \cap T_1) = p(A | T_1) \cdot p(T_2 | T_1) \cdot p(T_1) = p(A | T_2) \cdot p(T_1 | T_2) \cdot p(T_2). \quad (67)$$

Simplify gives that for all t_1 and t_2 then

$$p(A | T_1 = t_1) = p(A | T_2 = t_2).$$

Fix t_1 and use Lemma 1 to obtain $A \perp T_2$ then fix t_2 and use Lemma 1 to obtain $A \perp T_1$. If $A \perp T_2$, $A \perp T_1$ and by assumption $A \perp T_2 | T_1$ then

$$p(A, T_1 | T_2) = \frac{p(T_1)}{p(T_2)} p(A, T_2 | T_1) = \frac{p(T_1)}{p(T_2)} p(A | T_1) p(T_2 | T_1) = p(A | T_2) p(T_1 | T_2)$$

so that $A \perp T_1 | T_2$.

■

Theorem 2 has the following interpretation. If $A \perp T_2 | T_1$, then it makes no sense to perform test T_2 after T_1 since equality in Theorem 2 holds. However if test T_2 is executed first then performing test T_1 might be sensible.

THEOREM 3 (Sub-additivity)

If $A \cap B = \emptyset$ then

$$E\{p(A \cup B | T) | A \cup B\} \leq E\{p(A | T) | A\} + E\{p(B | T) | B\},$$

and equality holds if and only if $(A \perp T \text{ and } B \perp T) \text{ or } p(T = t | A) = p(T = t | B)$ for all $t \in \tau$

PROOF: We have that

$$E\{p(A \cup B | T) | A \cup B\} = \sum_{t \in \tau} \frac{p^2(A \cup B | T = t)}{p(A \cup B)} p(T = t)$$

and that

$$\begin{aligned} & E\{p(A | T) | A\} + E\{p(B | T) | B\} \\ &= \sum_{t \in \tau} \left(\frac{p^2(A | T = t)}{p(A)} + \frac{p^2(B | T = t)}{p(B)} \right) p(T = t). \end{aligned}$$

By looking at the inequality term-wise in t , it suffices to show that

$$\frac{p^2(A \cup B | T = t)}{p(A \cup B)} \leq \frac{p^2(A | T = t)}{p(A)} + \frac{p^2(B | T = t)}{p(B)}. \quad (68)$$

Since $A \cap B = \emptyset$ we have that

$$\frac{p^2(A \cup B | T = t)}{p(A \cup B)} = \frac{(p(A | T = t) + p(B | T = t))^2}{p(A) + p(B)}.$$

Writing $c_A(t) = p(A | T = t)/d_A$, $d_A = \sqrt{p(A)}$ and $c_B(t) = p(B | T = t)/d_B$, $d_B = \sqrt{p(B)}$, the inequality (68) simplifies to

$$\frac{(c_A(t)d_A + c_B(t)d_B)^2}{d_A^2 + d_B^2} \leq c_A^2(t) + c_B^2(t),$$

which is just the Cauchy-Schwartz inequality

$$(c_A(t)d_A + c_B(t)d_B)^2 \leq (c_A^2(t) + c_B^2(t))(d_A^2 + d_B^2).$$

Equality holds if and only if $c_A(t) = \beta(t)d_A$ and $c_B(t) = \beta(t)d_B$ for some non-zero constant $\beta(t)$ (recall that we consider a fixed t). This gives that

$$p(A | T = t) = \beta(t)p(A) \text{ and } p(B | T = t) = \beta(t)p(B)$$

must hold for all $t \in \tau$. Apply Bayes Theorem to get

$$p(A | T = t) = p(T = t | A) \cdot p(A)/p(T = t)$$

and

$$p(B | T = t) = p(T = t | B) \cdot p(B)/p(T = t).$$

This gives that

$$\beta(t) = p(T = t | A)/p(T = t) = p(T = t | B)/p(T = t).$$

This leads to two cases where equality will hold. The first is the trivial case when $A \perp T$ and $B \perp T$ which gives $\beta(t) = 1$. The only other possibility is when $p(T = t | A) = p(T = t | B)$ for all $t \in \tau$.

■

Theorem 3 can be used as an argument to claim that it is better to obtain information about $p(A | T = t)$ and $p(B | T = t)$ separately rather than to obtain information about $p(A \cup B | T = t)$. In other words it is better to relate test results to a finer partition of events (if possible) see also Corollary 3.

Let us consider an example to illustrate Theorem 3. Let $A = [0, \frac{1}{2})$, $B = [\frac{1}{2}, 1]$, and define $T = I_{[a,b]}(x)$, $x \in [0, 2]$. Let $0 \leq a \leq \frac{1}{2} \leq b \leq 1$. Think about an unknown point X drawn from a uniform distribution on $[0, 2]$. Now we want to predict where this point is.

Figure 13: Illustration of the example.

We have

$$p(A) = p(B) = 1/4, p(A \cup B) = 1/2, p(T = 1 | A) = 1 - 2a, p(T = 1 | B) = 2b - 1, \\ p(T = 1 | A \cup B) = b - a \text{ and } p(T = 1) = 1/2(b - a).$$

If $b = 1 - a$ then $p(T = 1 | A) = p(T = 1 | B)$ so that equality holds in Theorem 3. Substituting for b in the conditional probabilities above and taking the conditional expectation of $p(A \cup B | T = t)$, $p(A | T = t)$ and $p(B | T = t)$ gives

$$E\{p(A \cup B | T) | A \cup B\} = \sum_{t \in \{0,1\}} \frac{p^2(T=t | A \cup B)}{p(T=t)} p(A \cup B) \\ = \frac{(1-(b-a))^2}{1-1/2(b-a)} + \frac{(b-a)^2}{(b-a)} = \frac{1}{1+2a},$$

$$E\{p(A | T) | A\} = \sum_{t \in \{0,1\}} \frac{p^2(T=t | A)}{p(T=t)} p(A) \\ = \frac{(2a)^2}{4(1-1/2(b-a))} + \frac{(1-2a)^2}{2(b-a)} = \frac{1}{2(1+2a)}$$

and similarly we have that

$$E\{p(B | T) | B\} = \sum_{t \in \{0,1\}} \frac{p^2(T=t | B)}{p(T=t)} p(B) \\ = \frac{(2-2b)^2}{4(1-1/2(b-a))} + \frac{(2b-1)^2}{2(b-a)} = \frac{1}{2(1+2a)}.$$

This gives

$$E\{p(A | T) | A\} + E\{p(B | T) | B\} = E\{p(A \cup B | T) | A \cup B\}$$

as should be expected from Theorem 3. Note that

$$p(A) = 1/4 \neq p(A | T = 1) = 1/2$$

which implies that A is not independent of T .

COROLLARY 3 (Partition Inequality)

If T is a random variable and we are given the partition $A = \bigcup_{i=1}^n A_i$ of the event A then

$$E\left\{p\left(\bigcup_{i=1}^n A_i | T\right) \middle| \bigcup_{i=1}^n A_i\right\} \leq \sum_{i=1}^n E\{p(A_i | T) | A_i\}$$

PROOF: Put $B_1 = A_1$ and $B_2 = \bigcup_{i=2}^n A_i$. Apply Theorem 3 to B_1 and B_2 ,

$$E\{p(B_1 \cup B_2 | T) | B_1 \cup B_2\} \leq E\{p(B_1 | T) | B_1\} + E\{p(B_2 | T) | B_2\}. \quad (69)$$

For $k = 2, \dots, n-1$ iterate (69) with $B_1 = A_k$ and $B_2 = \bigcup_{i=k+1}^n A_i$ to prove the Corollary. ■

3.2 Savage's Decision Theory

Here we will introduce Savage's [1] decision theory. Cooke [2] gives an overview of the theory including some simplified proofs. Recall the notation in the introduction to Chapter 3, and note that the set of consequences or states of the acting subject must be finite, $|C| < \infty$. This must hold in order for the decision problem to be well defined (a model containing infinite number of consequences can not be evaluated). The set of available acts is $F = \{f | f: S \rightarrow C\}$. It is our intention to show the basic axioms so that we can evaluate the acts in F based on the expected utility

$$v(f) = \sum_{s \in S} U(f(s))p(s) = \sum_{c \in C} U(c)p(f^{-1}(c)) \quad (70)$$

where the uncertainty of S is described by the probability function $p(\bullet)$. The evaluation of for example two acts $f, g \in F$ will be to prefer act f rather than g if and only if $v(f) \geq v(g)$. Note that we speak about the act $f \in F$ when we write f while we speak about the consequence of the act when we write $f(s)$.

We assume existence of a partial ordering " $\geq$ " on F so that for $f_1, f_2 \in F$ we say that f_1 is preferable to f_2 if $f_1 \geq f_2$. We will write $f_1 \simeq f_2$ if $f_1 \leq f_2$ and $f_1 \geq f_2$, and $f_1 > f_2$ if $f_1 \leq f_2$ and not $f_1 \geq f_2$.

Weak order – Axiom 1

$\forall f_1, f_2 \in F, f_1 \geq f_2$ or $f_2 \geq f_1$ or both ($\geq$ is connected).

$\forall f_1, f_2, f_3 \in F$, if $f_1 \geq f_2$ and $f_2 \geq f_3$, then $f_1 \geq f_3$ ($\geq$ is transitive). ■

The weak order axiom can be regarded as a prediction of behaviour of people in decision situations. It is also a logic-like criterion of consistency in decision situations which is more important to us since it can be thought of as normative. For example the transitivity would prevent a rational person to be inconsistent when he evaluates acts according to preference.

A rational person would also like to consider an act for different events in S . For example consider that you must decide between taking a loan and buying a house or renting an apartment. You may ask yourself whether you would buy a house if the interest rate is low and you may ask yourself if you still would buy the house if the interest would be high. If you figure out that you would buy the house regardless of high or low interest rate then you should buy the house. The sure thing principle states that the preference between two acts is independent of the values the two acts take on any set where the two acts agree.

Sure thing principle – Axiom 2

Consider an event $A \subseteq S$. If $f, g, f', g' \in F$ are such that:

i. Given $\neg A, f \simeq g$ and $f' \simeq g'$

ii. Given $A, f \simeq f'$ and $g \simeq g'$

iii. $f \leq g$

then $f' \leq g'$

■

If an event A is irrelevant to the decision in question then it is said to be null. That is A is **null**, if and only if, for all f and $g, f \geq g$ given A . Note that this is also written $A = \emptyset$.

Acts that are **constant** are acts whose consequences are independent of the state of the world. They are of special interest since they naturally define preference among consequences in terms of preference among acts. Consider consequences $c, c' \in C$ where c' is preferable to c . We write the relationship between preference among acts and preference among consequences as $c \leq c'$ if and only if $f \leq f'$ when for every $s \in S, f(s) = c$ and $f'(s) = c'$.

We use the interpretation of null and the relationship between preference of acts and preference of consequences to state *the principle of dominance*. The principle of dominance is made to assert that the knowledge of an event cannot alter a rational decision-maker's preference among consequences.

Dominance principle – Axiom 3

Consider an event $B \neq \emptyset$. If we have that $f(s) = c, f'(s) = c'$ for every $s \in B$ then $f \leq f'$ given B if and only if $c \leq c'$. If also $f(s) > f'(s)$ for some $s \in B$, then $f > f'$ given B .

■

Consider the event *it will rain tomorrow*. If you offer a rational person a dollar if he can guess correctly that it would be *rain tomorrow* we often tacitly assume that he does not change his prediction if we raise the offer to 10 dollar. This assumption will be stated formally in *the principle of definition*. To offer a prize in case A means to make available to the person an act f_A such that $f_A(s) = c$ for $s \in A$ and $f_A(s) = c'$ for $s \in \neg A$, where $c' < c$.

Principle of definition – Axiom 4

Consider events A and B. If $f_A, f_B, g_A, g_B \in F$ and $c_f, c_f', c_g, c_g' \in C$ are such that

- i. $c_f > c_f'$ and $c_g > c_g'$.
- ii. For $s \in A$ then $f_A(s) = c_f$, $g_A(s) = c_g$ and for $s \in \neg A$ then $f_A(s) = c_f'$, $g_A(s) = c_g'$.
- iii. For $s \in B$ then $f_B(s) = c_f$, $g_B(s) = c_g$ and for $s \in \neg B$ then $f_B(s) = c_f'$, $g_B(s) = c_g'$.
- iv. $f_A \leq f_B$

then $g_A \leq g_B$.

From the principle of definition it will be said that A is not more probable than B abbreviated $A \leq B$ if and only if $f_A \leq f_B$. Definition 7 states this more formally. The principle of definition states that the subject's preference is independent of our particular choice of good and bad. Note that for the principle of definition not to fail due to triviality Savage proposes the following principle.

Principle of Consequences – Axiom 5

There is at least one pair of consequences $c, c' \in C$ such that $c < c'$.

DEFINITION 7 ($\geq$. relation)

For $A, B \subseteq S$, $A \geq B$ if and only if $f_A \geq f_B$.

We will write $A \simeq B$ if $A \leq B$ and $A \geq B$; and $A > B$ if $A \leq B$ and not $A \geq B$. Note that Definition 7 is an elegant way to link the relation $\geq$. to the principle of definition. The use of Definition 7 always ensures the consistence with the principle of definition.

DEFINITION 8 (Qualitative Probability)

A relation $\prec$ between events is a qualitative probability if and only if for all events B, C, D,

- i. $\prec$ is a weak order
- ii. $B \prec C$, if and only if $B \cup D \prec C \cup D$, provided $B \cap D = C \cap D = \emptyset$.
- iii. $\emptyset \prec A \prec S$

Savage [1] shows by using Axioms 1 to 5 that the relation " $\geq$." in Definition 7 is a qualitative probability.

3.2.1 Quantitative Probability

The aim is to represent the qualitative probability relation by a unique probability that is consistent with a rational person's preference relation. In order to do this the refinement principle is needed.

Refinement principle – Axiom 6

For any $B, C \subset S$ with $B < C$. There exists a partition $\{A_i\}_{i=1, \dots, n}$ of S such that $B \cup A_i < C$ for $i = 1, \dots, n$.

The following Theorem and Lemmas are not proven in detail. They are shown to give the reader a better understanding for what is needed to prove Theorem 4. The more detailed proofs are outlined in Cooke [2] and Savage [1].

THEOREM 4 (Quantitative Probability)

If $\succeq$ on $F \times F$ satisfies Axioms 1 to 6, then there exists a unique probability p on $(S, \mathcal{F})$, such that for any $A, B \in \mathcal{F}$ we have $p(A) \geq p(B) \Leftrightarrow A \succeq B$.

The refinement principle is important when proving that an event can be divided into a uniform partition $\{A_i\}_{i=1, \dots, 2^n}$ such that $A_i \simeq A_j, \forall i, j = 1, \dots, 2^n$. More precisely is formulated in Lemma 2.

LEMMA 2 (Existence of subsets)

$\forall B \in 2^S$ there exist subsets B_1 and B_2 such that $B_1 \cap B_2 = \emptyset, B_1 \cup B_2 = B$, and $B_1 \simeq B_2$

For our purpose Lemma 2 must be generalised so we can split S into any number of elements. This is ensured by

Lemma 3 (Partition into 2^n elements)

$\forall n \in 2, 3, \dots$ there exists a uniform partition of S into 2^n elements.

Now, since it is possible to partition S into any number of equal parts, this can be used to define (quantify) the probabilities of any event.

DEFINITION 9 (Quantification)

$\forall B \in 2^S$, let $B(n)$ denote the largest k such that $\bigcup_{i=1}^k A_i \leq B$ where $\{A_i\}_{i=1, \dots, 2^n}$ is a uniform partition of S , and let $p(B)$ be defined by

$$p(B) = \lim_{n \rightarrow \infty} B(n)/2^n.$$

Cooke [2] proves Theorem 4 by using refinement and Definition 9. ■

3.2.2 Utility

In the foregoing derivation of p preference was only treated by the principle of definition. The need for a utility function for the representation of preference needs another last axiom that explicitly uses p .

Strengthened refinement principle – Axiom 7

For $f, h, f', g' \in F$, if $f > g$, then there exists an α , $\alpha > 0$, such that for event A with $p(A) < \alpha$, $f \simeq f'$ given $\neg A$ and $g \simeq g'$ given $\neg A$, it follows that $f' > g'$. ■

The principle of strengthened refinement induces refinement in Axiom 6. This can be seen since $f > g$ then there are (by the principle of dominance) $B, C \subset S$ such that $B \succ C$. By the principle of strengthened refinement there exists $A \subset S$, such that $p(A) < \alpha$ where the preference can take any value without altering that $f > g$. This means that a sufficiently 'small' A can be chosen such that $A \cap B \subset B$. Since the preference is unaltered A can be added to both sides of $B \succ C$ which gives $B \succ C \cup A$.

THEOREM 5 (Utility)

Let C, F , and S be as defined above, let $\mathbb{R}$ denote the real numbers. If ' $\leq$ ' satisfies the 7 axioms above and let p be a quantitative probability representing ' $\leq$ '. Let $U: C \rightarrow \mathbb{R}$ be a utility function and let $v(f) = \sum_{c \in C} p(f^{-1}(c)) U(c)$ represent the expected utility of $f \in F$ with respect to p (sometimes later referred to as $E(f)$), then:

(1) There exists a function $U: C \rightarrow \mathbb{R}$ such that for all $f, h \in F$ we have that $f \geq h \Leftrightarrow Uf \geq Uh$.

(2) If $U^*: C \rightarrow \mathbb{R}$ is another function on C satisfying (1) then constants $a, b \in \mathbb{R}$, $a > 0$ exists such that $U = aU^* + b$. ■

Statement (2) says that utility U is unique up to positive affine transformations. The proof of Theorem 5 is quite complex and a proof is given in Cooke [2].

The seven axioms above are necessary to develop the notion of qualitative probability and ensure that the qualitative probability can be extended to a unique quantitative probability. This finally lets us evaluate the various acts in terms of expected utility. Together they define rational preference. Recall that conditions on $\leq$ was needed to state $f \leq g$ if and only if $v(f) \leq v(g)$. They can be summarised as follows:

DEFINITION 10 (Rational Preference)

A relation $\geq$ on F is a rational preference if the following hold:

- (1) $\geq$ is a weak order.
- (2) The sure thing principle.
- (3) The principle of definition.
- (4) The principle of dominance.
- (5) For some $c, d \in C$, then $c > d$.
- (6) Strengthened refinement principle.

■

We saw above in Section 3.2.2 that the strengthened refinement induces refinement via dominance.

3.2.3 Observation and The Value of Information

An observation is represented by a random variable $T : S \rightarrow \mathbb{R}$, $T(s) = \tau \subseteq \mathbb{R}$. Before performing an observation the degree of belief in event $B \in S$ is $p(B)$. After observing $T = t$, $t \in \tau$ the belief is determined via Bayesian revision $p(B|t) = p(t | B)p(B)/p(t)$. An observation is said to be definite if $p(B|t) = 0$ or 1. For any two disjoint events A, B with $p(A | t) \neq 0$, the likelihood ratio is

$$\frac{p(B|t)}{p(A|t)} = \frac{p(t|B)}{p(t|A)} \frac{p(B)}{p(A)} \quad (71)$$

If B is really the case it should expected that $\frac{p(B|t)}{p(A|t)} > \frac{p(B)}{p(A)}$ after observing $t \in \tau$.

This is stated more formally in Theorem 6.

THEOREM 6 (The effect of observation)

Let $R(t) = \frac{p(t|B)}{p(t|A)}$, then $E(R(t) | B) \geq 1$ and equality holds if and only if $p(t | B)/p(t | A) = 1$ for $t \in \tau$

PROOF: Theorem 6 will be proven via Jensen's inequality. Note that

$$E(1/R(T) \mid B) = \sum_{t \in \tau} \frac{p(t|A)}{p(t|B)} p(t|B) = 1.$$

Then using convexity of $-\log$ we can apply Jensen's inequality to get

$$\log[E(R(T) \mid B)] \geq E(\log[R(T)] \mid B) = E(-\log[1/R(T)] \mid B)$$

and

$$E(-\log[1/R(T)] \mid B) \geq -\log E(1/R(T) \mid B) = -\log(1) = 0.$$

Hence we have that $\log[E(R(T) \mid B)] \geq 0$, thus $E(R(T) \mid B) \geq 1$. $\log$ is a strictly concave function so that equality holds for Jensen's inequality if and only if $\log(R(t))$ is linear in t . This gives

$$\log[E(R(T) \mid B)] = E\{\log[R(T)] \mid B\}$$

if and only if

$$R(t) = \frac{p(t|B)}{p(t|A)}$$

is constant for all t given B . This gives $p(t|B) = kp(t|A)$ so that

$$\log[E(R(T) \mid B)] = \log k.$$

Above we have that $E(1/R(T) \mid B) = 1$ which gives $\log k = 0$, hence $k = 1$. This gives equality if and only if $p(t|B) = p(t|A)$.

■

Theorem 6 does not answer why one should observe, however it says what happens when observations are available. To answer the question of why observing we need to introduce some other properties.

DEFINITION 11 (Expected Value of a set of acts)

$\forall f \in F$ the value of F is $v(F) = \max_{f \in F} v(f)$.

■

It might be considered to observe T before choosing an element of F . After observing $T = t$, $t \in \tau$ the probability involved changes from $p(\bullet)$ to $p(\bullet \mid T = t)$ and the value of F would be computed with respect to this conditional probability.

DEFINITION 12 (Conditional Expected Value)

Given T with possible values $t \in \tau$ and $\forall f \in F$, the expected value of F with respect to $p(\bullet \mid T = t)$ is $v(F \mid t) = \max_{f \in F} v(f \mid t)$.

■

In applications where one has the choice to do a test the observation is unknown and thus random. Therefore as Definition 12 holds $\forall t \in \tau$ the expected value of F given T a priori can be determined as

$$v(F|T) = \sum_{t \in \tau} \max_{f \in F} v(f|t) p(t). \quad (72)$$

Observations are normally not free. It is only beneficial to observe if the expected gain $v(F | T) - v(F)$ from performing an observation is greater than the *cost of observation*. Theorem 7 is useful since it says something about the relation between $v(F | T)$ and $v(F)$.

THEOREM 7 (Increasing Value of Information)

For any random variable T and any $f \in F$, $v(F | T) \geq v(F)$.

PROOF:

$$v(F | T) = \sum_{t \in \tau} p(t) \max_{f \in F} [E(f|t)] \geq \sum_{t \in \tau} p(t) E(f|t) = E(f)$$

■

What Theorem 7 says is that by observing any T the value of the expected utility can only be better than what we have. This means that if observations are for free one should do as many as possible. In the real world however, observations are not for free, they are time consuming and an act must normally be chosen within a certain time.

3.3 Decision Trees and Influence Diagrams

Decision trees and influence diagrams are useful for communication purposes. Sometimes they provide a better understanding of the decision problem. We have adopted the notation used by Ada Decision Systems [5] for the software DPL. In this notation square nodes represent decisions and circular nodes uncertain variables.

Figure 14. Decision tree.

In Figure 14 we show a decision tree with a decision node with two acts f_1 and f_2 (note that a decision node can contain more than two acts), and a chance node with two states s_1 and s_2 (note that a chance node can contain more than two states). On each outgoing branch of the chance node the probability of the state/event is assigned. For example in Figure 14 $p(s_1) = 0.2$ and $p(s_2) = 0.8$. An expected utility is associated with each chance-node (see number in brackets next to chance nodes in Figure 14). The utility of consequences $f_1(s_1)$ is 100 and $f_1(s_2)$ is 0 i.e. see the squared bracket next to the triangular node. The bolded line f_2 indicates the optimal decision i.e. the decision that gives the highest expected utility. The highest expected utility is 80 (see number

in squared brackets next to the decision-node), and is obtained by act f_2 . This expected value equals the value of F in Definition 11. The decision tree in Figure 14 can be written as the influence diagram in Figure 15.

Figure 15. Influence diagram.

Again in this diagram square nodes represent available acts F and circular or elliptic ones uncertain variables. The utility of the consequences can be represented by the value-node. The decision node represents the available acts, here $f_1, f_2 \in F$, and the chance node the set of states $s_1, s_2 \in S$. The arrows 1, 2 and 3 are influence relations. In software tools as DPL, the user is allowed to specify the influence by specifying conditional probabilities and/or temporal relationships. For example arrow 1 in Figure 15 only indicates the order that is the decision node is before the chance node. Arrow 2 and 3 are influence nodes that also affects the value node so that the value node contains the states $F \times S$ for which the utility of consequences $f_i(s_j)$ must be specified for all i and j .

The advantage of influence diagrams is that one can build up a Bayesian network of chance nodes, see Jensen [6].

Figure 16. Influence diagram with states S and one test.

Influence diagrams can be used to model events and tests/observations. For example in Figure 16 the test outcomes *ok* and *not ok* are assigned conditional probabilities given the states s_1 and s_2 . This allows automatic Bayesian revision with the DPL software in the decision tree as shown in Figure 17.

Figure 17. Decision-tree with one inspection.

When Figure 14 is compared to Figure 17 one can see that the expected utility increases from 80 to 90 by performing the test. Note that the expected utility is the expected value in (72). If the cost of carrying out the test is less than 10 one should inspect the test. This is often defined as the value of information. This will be treated in more detail in Chapter 4 and in Section 4.1.2.

3.4 References

- [1] L. Savage: "The Foundations of Statistics". Dover, New York, 1972.
- [2] R. M. Cooke: "Experts in Uncertainty", Oxford University Press, New York, 1991.
- [3] H. G. Tucker: "A Graduate Course in Probability", Academic Press Inc., New York, 1967.
- [4] W. Rudin: "Principles of Mathematical Analysis", McGraw-Hill, New York, 1953.
- [5] Ada Decision Systems 1992: "DPL Advanced Version User Guide". 2710 Sand Hill Road, Menlo Park, CA94025, USA.
- [6] F. V. Jensen: "An introduction to Bayesian networks", UCL Press, London, 1996.

- [7] S. French, 'Decision Theory: An introduction to the Mathematics of Rationality', Ellis Horwood Limited, 1993.

Chapter 4

Value of Information and Scheduling of Tests

In this chapter we will in detail define the mathematical concepts that were used when we created the examples shown in Norström et al [2], [3]. The strict page restrictions that is required in conference proceedings does not always allow a throughout discussion of mathematical details. Here it is our intention to show how the notation is defined and to give the important proof of Theorem 15 that relates the contents in this chapter to results discussed about optimal statistical decision by De Groot [6].

The case is to design a system out of various "*off the shelf*" tests. One can think of cost of a test in terms of building in cost (set-up cost) and then latter also in terms of cost of performing a pre-set-up test. Grouping into these two categories of costs makes it easier to describe the behaviour of cost of testing.

Set-up cost

Set-up cost is also called mount cost, and the variable μ is used to describe such costs. Some characteristics concerning the behaviour of the mount costs are:

1. It might be cost savings to implement a group of tests in the design, e.g. $\mu_1 + \mu_2 \neq \mu_{1,2}$.
2. Set up cost might differ if set up of several tests is not contemporary, e.g. set up T_1 ; latter set up T_2 then costs $\neq$ costs of setting up T_1 and T_2 together.
3. The set up cost could depend on the order the tests are set up, e.g. $\mu_{1,2} \neq \mu_{2,1}$.
4. It does not seem reasonable that the set up costs depend on a particular outcome of the test itself or on any other test.

Consider a set of tests $\psi = \{T_1, \dots, T_k\}$. Let π be a permutation of the tests. In first approximation it is assumed that the tests $(T_{\pi_1}, \dots, T_{\pi_k})$ are set-up together such that the set-up cost is:

$$\mu_{\pi_1, \dots, \pi_k} = \text{set-up cost of the group of tests } (T_{\pi_1}, \dots, T_{\pi_k}).$$

Since the cost does not depend on the test results, it is sufficient to consider μ as a function of $\bigcup_{k=0}^{\infty} \psi^k$. Thus the set up cost is a function³

---

³ Note: Doing the same test 2 times is covered by the notation (regard as 2 distinct tests).

$$\mu: \bigcup_{k=0}^{\infty} \Psi^k \rightarrow \mathbb{R},$$

and if a test sequence is not permissible then $\mu = \infty$.

Running cost of performing a test:

The variable κ is used to describe performance costs. Some characteristics that describe the performance costs are:

1. It could depend on the test results.
2. It might depend on the order: $\kappa_{1,2} \neq \kappa_{2,1}$ or $\kappa_{1,2,3,4} \neq \kappa_{1,\{2,3\},4}$ (the last $\kappa_{1,\{2,3\},4}$ mean that tests 2 and 3 are performed at the same time).
3. Test results may come in different order than testing. For instance say that first tests 1 and then 2 have been performed then the results can come in the order 2 and then 1.
4. It might depend on the test results. In case the order of test outcomes have importance for κ .

A sufficient notation is of the kind $\kappa(t_1, t_2, r_1, r_2, t_3, r_3)$. It is the performance cost of first doing test 1, then test 2, receiving the test result of test 1, receiving the result of test 2, doing test 3 and receiving its outcome. The running cost of performing a test is a function³

$$\kappa: \bigcup_{k=0}^{\infty} (2^{\Psi} \cup 2^{\tau(\Psi)})^k \rightarrow \mathbb{R},$$

with $\kappa = \infty$ if the test results come before the test or if the test sequence is not permissible. Recall that $\tau(\Psi)$ is the set of outcomes for the set of tests Ψ . 2^{Ψ} is the power set of all possible collections of tests (when grouping tests together) and $2^{\tau(\Psi)}$ is the set of all possible collections of test results. k is denoting the number of tests in Ψ .

The problem of designing control system

A control system is build up of three parts the control software, the tests or sensors and the actuators. The control software reasons from the information provided by the tests or sensors. Before using a sensor or test, the cost for making a sensor available must be paid. Such costs are the mount costs. Every time a test or sensor is used then an additional performance cost must be paid. Based on this information with knowledge of the possible consequences the control software activates actuators that influence the state of the world. The problem of designing the control system is divided into two steps. The first step is to apply the performance cost and to find the control software structure (the control flow diagram) that is optimal with respect to the expected utility. Then the second step is to use the expected utility from this optimal control software together with the mount costs in the selection of the tests to be mounted in the system. This chapter is completely devoted to the first step.

4.1 The Choice to do an Observation

The most general utility function U that can be thought of that is capable of covering the performance costs as well as the consequences can be imagined over the set of all sequences of all possible collections of tests and test results. This function is written

$$U : C \times \bigcup_{k=0}^{\infty} (2^{\Psi} \cup 2^{\tau(\Psi)})^k \rightarrow \mathbb{R},$$

with $U = -\infty$ if the test result comes before the test or if the test sequence is not permissible. C is the set of all possible consequences. Savage [1] defined the utility as a function $U : C \rightarrow \mathbb{R}$ (note $f : S \rightarrow C$). The utility function that is required in our problem is different since it also covers the cost of testing. Using the notation above opens for three levels of generality:

1. *Most general*: Utility depends on the test sequence, the test results and the order of both test results and tests.
2. *Less general*: The test result is received before making any new tests which gives a utility function of the kind $U(c, t_1, r_1, t_2, r_2, t_3, r_3, \dots)$. This is covered by the notation above by in addition letting $U = -\infty$ if the test result is not the first to follow the test, e.g. $U(c, t_1, t_2, r_1) = -\infty$.
3. *Simple*: The utility may only depend on whether a test is executed or not, on the order the tests are arranged and on the grouping of tests. In this case a notation of the kind $U_{1,2,3}(c)$ and $U_{1,(2,3)}(c)$ (see below) is sufficient.

For the moment the utility will be limited to the simple situation 3. by assuming:

ASSUMPTION 1: *The cost of a test does not depend on the test results.*

ASSUMPTION 2: *After a test has been executed the test result is inspected immediately.*

DEFINITION 13 (Utility)

Let $U_{\pi_1, \dots, \pi_k}(c)$ be defined as the utility of consequence c when the tests $(T_{\pi_1}, \dots, T_{\pi_k}) \in \Psi$ have been performed in the order of permutation π

■

Given that the tests $(T_{\pi_1}, \dots, T_{\pi_k})$ have been executed the utility $U_{\pi_1, \dots, \pi_k}(c)$ is not different from the utility function as suggested by Savage [1], and it has the same properties. However, it is different since it reflects the utility when the tests $(T_{\pi_1}, \dots, T_{\pi_k})$ have been executed sequentially one by one. The utility only depends on which of the tests in Ψ that has been performed and on the order in permutation π . For example:

$U_{i_1, \dots, i_n}(c)$ = utility as a function of $c \in \mathbf{C}$ after having performed the test sequence $(T_{i_1}, T_{i_2}, \dots, T_{i_n})$ one by one in the order of permutation i .

Definition 13 also covers grouping of tests into a meta-test that is performed as one test. By applying the following trick meta-test T_4 will be created. Assume that $\psi = \{T_1, T_2, T_3\}$. Assume that tests T_2 and T_3 are built into a meta-test that always is performed together as one test T_4 , then we can expand ψ to include a fourth test $T_4 = (T_2, T_3)$.

The expected utility of doing n tests $T_{i_1}, T_{i_2}, \dots, T_{i_n}$ in the order of permutation i is

$$v(F | T_{i_1} = t_{i_1}, \dots, T_{i_n} = t_{i_n}) = \max_{f \in F} \sum_{c \in \mathbf{C}} p(f^{-1}(c) | (T_{i_1} = t_{i_1}, \dots, T_{i_n} = t_{i_n})) U_{i_1, \dots, i_n}(c).$$

The conditional expected utility given $T_{i_1} = t_{i_1}$ and meta-test (T_{i_2}, T_{i_3}) is written:

$$\begin{aligned} & v(F | T_{i_1} = t_{i_1}, (T_{i_2}, T_{i_3})) \\ &= \sum_{t_{i_2}, t_{i_3}} p(t_{i_2}, t_{i_3} | T_{i_1} = t_{i_1}) \max_{f \in F} \sum_{c \in \mathbf{C}} p(f^{-1}(c) | (T_{i_1} = t_{i_1}, T_{i_2} = t_{i_2}, T_{i_3} = t_{i_3})) U_{i_1, (i_2, i_3)}(c) \end{aligned}$$

Meta-test T_{i_2}, T_{i_3} is indicated in the notation by using brackets. Consider a set of tests $\psi = \{T_1, \dots, T_n\}$ and a fixed permutation π of the tests. Assume when nothing else is specified that $(T_{\pi_1}, \dots, T_{\pi_n})$ is simply written $(T_1, \dots, T_n)$.

The following discussion assumes that every test outcome is inspected and used to decide whether it is cost efficient to perform a second test or to act on the information that is already available. The issue that is whether a higher expected utility can be obtained by executing a second test given that one test is already executed. To answer this, it is logical to compare:

$$v(F | T_1 = t_1) = \max_{f \in F} \sum_{c \in \mathbf{C}} p(f^{-1}(c) | T_1 = t_1) U_1(c)$$

to

$$v(F | T_1 = t_1, T_2) = \sum_{t_2 \in \tau_2} (p(t_2 | T_1 = t_1) v(F | T_1 = t_1, T_2 = t_2)).$$

Note that the cost of testing is implemented in the utility by Definition 13. A rational decision-maker will execute two tests given the outcome of the first test if and only if

$$v(F | T_1 = t_1, T_2) > v(F | T_1 = t_1).$$

Let $v(F | T_1 = t_1; \emptyset \rightarrow T_2)$ be defined as

$$v(F | T_1 = t_1; \emptyset \rightarrow T_2) = \max\{v(F | T_1 = t_1, T_2), v(F | T_1 = t_1)\}$$

The expected utility given that the control software can choose to do a second test or perform the action after having read the outcome of the first test is given by:

$$\begin{aligned}
v(F|T_1 \rightarrow T_2) &= \sum_{t_1 \in \tau_1} p(t_1) v(F|T_1 = t_1; \emptyset \rightarrow T_2) \\
&= \sum_{t_1 \in \tau_1} p(t_1) \max \{v(F|T_1 = t_1), v(F|T_1 = t_1, T_2)\} \\
&= \sum_{t_1 \in \tau_1} p(t_1) \max \left\{ \begin{aligned} &\max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|T_1 = t_1) U_1(c), \\ &\sum_{t_2 \in \tau_2} p(t_2|T_1 = t_1) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|T_1 = t_1, T_2 = t_2) U_{1,2}(c) \end{aligned} \right\}
\end{aligned}$$

Note: When the maximum between $v(F|T_1 = t_1)$ and $v(F|T_1 = t_1, T_2)$ is taken the optimal act may change. This is the wanted effect that, say the control software, is able to reason from the available information and decide which test or act to do next (while running).

This notation also works when starting from nothing. Let the "null" test be the denoted by $\emptyset$. When the decision-maker has the choice to do or not to do T_1 this will be written $v(F|\emptyset \rightarrow T_1)$. It can be seen that this is consistent with the notation above since

$$\begin{aligned}
v(F|\emptyset \rightarrow T_1) &= \max \{v(F), v(F|T_1)\} \\
&= \max \left\{ \begin{aligned} &\max_{f \in F} \sum_{c \in C} p(f^{-1}(c)) U(c), \\ &\sum_{t_1 \in \tau_1} p(t_1) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1) U_1(c) \end{aligned} \right\}.
\end{aligned}$$

Now a third test is made available. According to the definition above the third test can be implemented in one of the following two ways: $[T_1 \rightarrow T_2] \rightarrow T_3$ or $T_1 \rightarrow [T_2 \rightarrow T_3]$. Verbally $[T_1 \rightarrow T_2] \rightarrow T_3$ would have the interpretation that the tests in the brackets have been considered and some of them (either T_1 or both T_1 and T_2) have been executed. Based on the possible information available from having the ability to consider these two tests and to see the outcome of them it is decided whether or not the next test, here T_3 , should be executed. This is in contrast to what we try to express by $T_1 \rightarrow [T_2 \rightarrow T_3]$. This has the interpretation that only test T_1 has been executed and that we are about to consider to do up to two new tests in $[T_2 \rightarrow T_3]$. The brackets in $[T_2 \rightarrow T_3]$ indicate that one is choosing to do T_3 only after having done T_2 and seen the outcome of it. Note that this does not allow us to do T_3 without first having done T_2 .

The brackets are clarifying the information that is possibly available at the time a test is being decided. The brackets in $[T_1 \rightarrow T_2] \rightarrow T_3$ indicate that T_3 is considered given test results t_1 or both t_1 and t_2 , while in $T_1 \rightarrow [T_2 \rightarrow T_3]$ they indicate that T_3 is only considered given both t_1 and t_2 .

Intuitively, in the first case, where $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$ is to be defined, there can be one out of two possible situations. The situation in which only one test has been executed, or the situation in which two tests has been executed. Thus it is logical to compare either

$$v(F|T_1 = t_1) = \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|T_1 = t_1) U_1(c) \quad (73)$$

or

$$v(F|T_1 = t_1, T_2 = t_2) = \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|T_1 = t_1, T_2 = t_2) U_{1,2}(c) \quad (74)$$

to what the expected benefit of making a third test is. In the situation where only T_1 was observed, then (73) is compared to

$$v(F|T_1 = t_1, T_3) = \sum_{t_3 \in T_3} (p(T_3 = t_3|T_1 = t_1) v(F|T_1 = t_1, T_3 = t_3)).$$

While in the situation that both T_1 and T_2 are observed, (74) is compared to

$$v(F|T_1 = t_1, T_2 = t_2, T_3) = \sum_{t_3 \in T_3} \left(p(t_3|T_1 = t_1, T_2 = t_2) v(F|T_1 = t_1, T_2 = t_2, T_3 = t_3) \right).$$

Using this, it is rational to perform test T_3 given that the result of the first test is $T_1 = t_1$ if and only if

$$v(F|T_1 = t_1, T_3) > v(F|T_1 = t_1)$$

or it is rational to perform test T_3 given that the results of both the first two tests are $T_1 = t_1$ and $T_2 = t_2$ if and only if

$$v(F|T_1 = t_1, T_2 = t_2, T_3) > v(F|T_1 = t_1, T_2 = t_2).$$

Note that $\emptyset$ is the empty test or to do a test which does not give any information. Recall

$$v(F|T_1 = t_1; \emptyset \rightarrow T_3) = \max\{v(F|T_1 = t_1, T_3), v(F|T_1 = t_1)\}.$$

The arrow between $\emptyset$ and T_3 indicates that the choice to do or not to do T_3 is being made given the observation $T_1 = t_1$. Recall

$$v(F|T_1 = t_1, T_2 = t_2; \emptyset \rightarrow T_3) = \max\{v(F|T_1 = t_1, T_2 = t_2, T_3), v(F|T_1 = t_1, T_2 = t_2)\}.$$

The arrow between $\emptyset$ and T_3 indicates that the choice to do or not to do T_3 is being made given the observations $T_1 = t_1$ and $T_2 = t_2$.

If at a level where only $T_1 = t_1$ is known then it is rational to do test T_2 and consider T_3 , rather than to only consider T_3 if and only if

$$v(F|T_1 = t_1, T_2 \rightarrow T_3) > v(F|T_1 = t_1; \emptyset \rightarrow T_3).$$

where

$$v(F|T_1 = t_1, T_2 \rightarrow T_3) = \sum_{t_2} p(t_2|T_1 = t_1) v(F|T_1 = t_1, T_2 = t_2; \emptyset \rightarrow T_3).$$

Let $v(F|T_1 = t_1; [\emptyset \rightarrow T_2] \rightarrow T_3)$ be defined as

$$v(F|T_1 = t_1; [\emptyset \rightarrow T_2] \rightarrow T_3) = \max \left\{ v(F|T_1 = t_1; \emptyset \rightarrow T_3), \right. \\ \left. v(F|T_1 = t_1; T_2 \rightarrow T_3) \right\}.$$

The arrow between $\emptyset$ and T_2 indicates that the choice to do or not to do T_2 is being made given $T_1 = t_1$.

Finally this makes it possible to define, $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$ as the expected benefit of first having the choice to do or not to do T_2 and based on this choice also having the choice to do or not to do T_3 latter, as

$$v(F|[T_1 \rightarrow T_2] \rightarrow T_3) = \sum_{t_1 \in \tau_1} p(t_1) v(F|t_1; [\emptyset \rightarrow T_2] \rightarrow T_3)$$

which is the same as

$$= \sum_{t_1 \in \tau_1} p(t_1) \max \{ v(F|t_1; \emptyset \rightarrow T_3), v(F|t_1, T_2 \rightarrow T_3) \}$$

$$= \sum_{t_1 \in \tau_1} p(t_1) \max \left\{ \max \{ v(F|t_1, T_3), v(F|t_1) \}, \right. \\ \left. \sum_{t_2 \in \tau_2} p(t_2 | t_1) v(F|t_1, t_2; \emptyset \rightarrow T_3) \right\}$$

$$= \sum_{t_1 \in \tau_1} p(t_1) \max \left\{ \max \{ v(F|t_1, T_3), v(F|t_1) \}, \right. \\ \left. \sum_{t_2 \in \tau_2} p(t_2 | t_1) \max \left\{ v(F|t_1, t_2, T_3), \right. \right. \\ \left. \left. v(F|t_1, t_2) \right\} \right\}$$

or as

$$= \sum_{t_1 \in \tau_1} p(t_1) \max \left\{ \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1\} U_1(c), \right. \\ \left. \sum_{t_2 \in \tau_2} p(t_2 | t_1) \max \left\{ \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1, t_2\} U_{1,2}(c), \right. \right. \\ \left. \left. \sum_{t_3 \in \tau_3} p\{t_3 | t_1, t_2\} \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1, t_2, t_3\} U_{1,2,3}(c) \right\} \right\} \\ \left. \sum_{t_3 \in \tau_3} p\{t_3 | t_1\} \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1, t_3\} U_{1,3}(c) \right\}$$

In the second case $v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$ is defined. Now it is given that T_1 is known and up to two more tests are considered. It is logical to compare

$$v(F|T_1 = t_1) = \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | T_1 = t_1\} U_1(c)$$

to

$$v(F|T_1 = t_1, T_2 \rightarrow T_3) = \sum_{t_2 \in T_2} p(t_2 | T_1 = t_1) \max \left\{ \begin{array}{l} v(F|T_1 = t_1, T_2 = t_2) \\ v(F|T_1 = t_1, T_2 = t_2, T_3) \end{array} \right\}$$

The arrow between T_2 and T_3 indicates that this is the choice to do or not to do T_3 given $T_1 = t_1$ and T_2 . From this it is rational to do $[T_2 \rightarrow T_3]$ if and only if

$$v(F|T_1 = t_1, T_2 \rightarrow T_3) > v(F|T_1 = t_1)$$

Define or even recall that

$$v(F|T_1 = t_1; \emptyset \rightarrow [T_2 \rightarrow T_3]) = \max \left\{ \begin{array}{l} v(F|T_1 = t_1), \\ v(F|T_1 = t_1, T_2 \rightarrow T_3) \end{array} \right\}$$

The arrow between $T_1 = t_1$ and $[T_2 \rightarrow T_3]$ indicates that this is the choice to do or not to do $[T_2 \rightarrow T_3]$ given $T_1 = t_1$.

Recall:

$$v(F|T_1 \rightarrow [T_2 \rightarrow T_3]) = \sum_{t_1 \in T_1} p(t_1) v(F|t_1; \emptyset \rightarrow [T_2 \rightarrow T_3])$$

which can be written as

$$\begin{aligned} &= \sum_{t_1 \in T_1} p(t_1) \max \left\{ \begin{array}{l} v(F|t_1), \\ v(F|t_1, T_2 \rightarrow T_3) \end{array} \right\} \\ &= \sum_{t_1 \in T_1} p(t_1) \max \left\{ \begin{array}{l} v(F|t_1), \\ \sum_{t_2 \in T_2} p(t_2 | t_1) \max \left\{ \begin{array}{l} v(F|t_1, t_2) \\ v(F|t_1, t_2, T_3) \end{array} \right\} \end{array} \right\} \\ &= \sum_{t_1 \in T_1} p(t_1) \max \left\{ \begin{array}{l} \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1\} U_1(c), \\ \sum_{t_2 \in T_2} p(t_2 | t_1) \max \left\{ \begin{array}{l} \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1, t_2\} U_{1,2}(c) \\ \sum_{t_3 \in T_3} p(t_3 | t_1, t_2) \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c) | t_1, t_2, t_3\} U_{1,2,3}(c) \end{array} \right\} \end{array} \right\} \end{aligned}$$

4.1.1 Formalism

Let us go through the definitions in Section 4.1 and state them more formally. Vectors are bolded so that $\mathbf{T}$ is a random test vector.

DEFINITION 14

The conditional expected utility given test $T = t$ is defined by:

$$v(F|T = t) = \max_{f \in F} \sum_{c \in C} p\{f^{-1}(c)|T = t\} U_T(c)$$

■

DEFINITION 15

The conditional expected utility given tests $T = t$ and T' is defined by:

$$v(F|T = t, T') = \sum_{t' \in T'} p\{t'|T = t\} v(F|T = t, T' = t')$$

■

Note, since the cost is dependent on the order of the tests then $v(F|T = t, T') \neq v(F|T', T = t)$ (the interpretation of the right hand side is obvious).

DEFINITION 16

Given $T = t$, the expected utility of having the choice to do or not to do test T' is defined as:

$$v(F|T = t; \emptyset \rightarrow T') = \max\{v(F|T = t), v(F|T = t, T')\}$$

■

The arrow in Definition 16 indicates the choice between only acting given $T = t$ or to do T' first (and then act!).

REMARK: Use the convention to only do test T when

$$v(F|T = t) = v(F|T = t, T') \quad (75)$$

in Definition 16. This ensures a unique test schedule later with as few tests as possible.

DEFINITION 17

The expected utility of the test schedule with the choice to do or not to do T' after T is defined as:

$$v(F|T \rightarrow T') = \sum_{t \in T} p(t) v(F|T = t; \emptyset \rightarrow T')$$

■

Now three (one-dimensional) tests T_1 , T_2 and T_3 are used to develop the concepts further. Theorem 8 describes a situation where two ordered tests always are better than only one test.

THEOREM 8

If $\forall t_1 \ v(F|T_1 = t_1) < v(F|T_1 = t_1, T_2)$ then

$$v(F|T_1 \rightarrow T_2) = v(F|T_1, T_2).$$

PROOF: Since $\forall t_1$ we have that $v(F|T_1 = t_1) < v(F|T_1 = t_1, T_2)$ Theorem 8 follows directly from Definition 16 and Definition 17. ■

COROLLARY 4

Let the cost structure be such that $U_{1,2}(c) = U_{2,1}(c)$ and assume that $\forall t_1 \ v(F|T_1 = t_1) < v(F|T_1 = t_1, T_2)$ and $\forall t_2 \ v(F|T_2 = t_2) < v(F|T_2 = t_2, T_1)$ then

$$v(F|T_2, T_1) = v(F|T_1, T_2). \quad (76)$$

PROOF: From Theorem 8 it follows that

$$v(F|T_1 \rightarrow T_2) = v(F|T_1, T_2)$$

and that

$$v(F|T_2 \rightarrow T_1) = v(F|T_2, T_1).$$

The definition of $v(F|T_2, T_1)$ gives

$$v(F|T_2, T_1) = \sum_{t_1, t_2} p(t_2, t_1) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | T_2 = t_2, T_1 = t_1) U_{2,1}(c),$$

and the definition of $v(F|T_1, T_2)$ gives

$$v(F|T_1, T_2) = \sum_{t_1, t_2} p(t_1, t_2) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | T_1 = t_1, T_2 = t_2) U_{1,2}(c).$$

Hence, if $U_{1,2}(c) = U_{2,1}(c)$ then there is equality in (76). ■

In Definition 18 a situation is given where perfect information of the observation $T_1 = t_1$ is available prior to decide whether to do or not to do observation T_2 or where perfect information of the observations $T_1 = t_1$ and $T_2 = t_2$ are available prior to decide whether to do or not to do T_3 .

DEFINITION 18

Given $T_1 = t_1$, the expected utility of having the choice to do or not to do test T_3 after the schedule $[\emptyset \rightarrow T_2]$ is defined as:

$$v(F | T_1 = t_1; [\emptyset \rightarrow T_2] \rightarrow T_3) = \max \left\{ \begin{array}{l} v(F | T_1 = t_1; \emptyset \rightarrow T_3) \\ v(F | T_1 = t_1, T_2 \rightarrow T_3) \end{array} \right\}$$

■

The arrow between $[\emptyset \rightarrow T_2]$ and T_3 in Definition 18 indicates the choice to do or not to do test T_3 given $[\emptyset \rightarrow T_2]$. The expression in the brackets must be solved before deciding about T_3 , and the arrow between $\emptyset$ and T_2 indicates that one has the choice to do or not to do T_2 . The convention to do tests T_1 and maybe T_3 is adopted when

$$v(F | T_1 = t_1; \emptyset \rightarrow T_3) = v(F | T_1 = t_1, T_2 \rightarrow T_3).$$

This is to maintain the policy to do the lowest number of tests in the Remark after Definition 16.

DEFINITION 19

The expected utility given the test schedule with the choice to do or not to do T_3 after $[T_1 \rightarrow T_2]$ is defined as:

$$v(F | [T_1 \rightarrow T_2] \rightarrow T_3) = \sum_{t_1} p(t_1) v(F | T_1 = t_1; [\emptyset \rightarrow T_2] \rightarrow T_3)$$

■

As for Definition 18 a situation with a test strategy of three tests is described, however now it is not possible to consider T_3 without first having done T_2 .

DEFINITION 20

Given $T_1 = t_1$, the expected utility of having the choice to do or not to do the test schedule $[T_2 \rightarrow T_3]$ is defined as:

$$v(F | T_1 = t_1; \emptyset \rightarrow [T_2 \rightarrow T_3]) = \max \left\{ \begin{array}{l} v(F | T_1 = t_1) \\ v(F | T_1 = t_1, T_2 \rightarrow T_3) \end{array} \right\} \quad (77)$$

■

The arrow between $\emptyset$ and $[T_2 \rightarrow T_3]$ indicates that this is the choice to do or not to do $[T_2 \rightarrow T_3]$. If $v(F | T_1 = t_1) = v(F | T_1 = t_1, T_2 \rightarrow T_3)$ use the convention to only do test T_1 . This maintains the policy in the Remark below Definition 16 to give a unique test schedule with as few tests as possibly.

DEFINITION 21

The expected utility given the test schedule with the choice to do or not to do $[T_2 \rightarrow T_3]$ after T_1 is defined as:

$$v(F|T_1 \rightarrow [T_2 \rightarrow T_3]) = \sum_{t_1} p(t_1) v(F|T_1 = t_1; \emptyset \rightarrow [T_2 \rightarrow T_3])$$

■

In $v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$ it is impossible to do T_3 without first having done T_2 , where for $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$ this is however possible. One might say that $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$ is a richer choice and intuitively it is expected to have a value that at least is bigger than $v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$. In Norström et al [2] this result is shown.

THEOREM 9

- i) $v(F|[T_1 \rightarrow T_2] \rightarrow T_3) \geq v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$ with equality holding if and only if $\forall t_1$:

$$v(F|t_1, T_3) \leq \max\{v(F|t_1), v(F|t_1, T_2 \rightarrow T_3)\}.$$

- ii) $v(F|T_1 \rightarrow [T_2 \rightarrow T_3]) \geq v(F|T_1 \rightarrow (T_2, T_3))$ with equality holding if and only if $\forall t_1$:

$$\left\{ \forall t_2 \left\{ \begin{array}{l} v(F|t_1, t_2, T_3) \geq v(F|t_1, t_2) \text{ or} \\ v(F|t_1, t_2, T_3) \leq v(F|t_1, t_2) \end{array} \right\} \right\} \text{ and } v(F|t_1, T_2) \leq v(F|t_1; \emptyset \rightarrow (T_2, T_3))$$

- iii) $v(F|T_1 \rightarrow (T_2, T_3)) \geq v(F|T_1, T_2, T_3)$ with equality if and only if $\forall t_1$:

$$v(F|t_1) \leq v(F|t_1, T_2, T_3).$$

PROOF of i): From the definitions it can be seen that

$$\max \left\{ \begin{array}{l} v(F|t_1; \emptyset \rightarrow T_3) \\ v(F|t_1; T_2 \rightarrow T_3) \end{array} \right\} \geq \max \left\{ \begin{array}{l} v(F|t_1) \\ v(F|t_1; T_2 \rightarrow T_3) \end{array} \right\}.$$

From Definition 16 it follows that

$$v(F|t_1; \emptyset \rightarrow T_3) = \max\{v(F|t_1), v(F|t_1, T_3)\}.$$

Hence it follows that the inequality in i) is proven with equality holding if and only if

$$v(F|t_1, T_3) \leq \max\{v(F|t_1), v(F|t_1, T_2 \rightarrow T_3)\}.$$

PROOF of ii): Looking term-wise for t_1 , it suffices to show that

$$v(F | t_1; \emptyset \rightarrow [T_2 \rightarrow T_3]) \geq v(F | t_1; \emptyset \rightarrow (T_2, T_3)).$$

Substituting

$$v(F | t_1; T_2 \rightarrow T_3) = \sum_{t_2} p(t_2 | t_1) \max \begin{cases} v(F | t_1, t_2), \\ v(F | t_1, t_2, T_3) \end{cases}$$

into (77) and applying Jensen's inequality to the sum over t_2 gives

$$v(F | t_1; \emptyset \rightarrow [T_2 \rightarrow T_3]) \geq \max \begin{cases} v(F | t_1), \\ v(F | t_1, T_2), \\ v(F | t_1, T_2, T_3) \end{cases}.$$

Hence

$$v(F | t_1; \emptyset \rightarrow [T_2 \rightarrow T_3]) \geq \max \begin{cases} v(F | t_1, T_2) \\ v(F | t_1; \emptyset \rightarrow (T_2, T_3)) \end{cases} \geq v(F | t_1; \emptyset \rightarrow (T_2, T_3))$$

with equality holding if and only if

$$\left\{ \forall t_2 \begin{cases} v(F | t_1, t_2, T_3) \geq v(F | t_1, t_2) \text{ or } \\ v(F | t_1, t_2, T_3) \leq v(F | t_1, t_2) \end{cases} \right\} \wedge v(F | t_1, T_2) \leq v(F | t_1; \emptyset \rightarrow (T_2, T_3)).$$

This completes the proof of *ii*) above.

PROOF of *iii*): Looking term-wise for t_1 it suffices to show that

$$v(F | t_1; \emptyset \rightarrow (T_2, T_3)) \geq v(F | t_1, T_2, T_3).$$

The left-hand side is

$$v(F | t_1; \emptyset \rightarrow (T_2, T_3)) = \max \begin{cases} v(F | t_1) \\ v(F | t_1, T_2, T_3) \end{cases}$$

which is obviously greater or equal to $v(F | t_1, T_2, T_3)$ with equality holding if and only if $v(F | t_1) \leq v(F | t_1, T_2, T_3)$. This proves statement *iii*). ■

Theorem 9 can be used to make conclusions about other test schedules, for example:

$$v(F | [T_1 \rightarrow [T_2 \rightarrow [(T_3, T_4) \rightarrow T_5]]) \leq v(F | [T_1 \rightarrow T_2] \rightarrow [(T_3, T_4) \rightarrow T_5]).$$

To see this, let $\Theta = [(T_3, T_4) \rightarrow T_5]$. Θ is to do T_3 and T_4 together and then having the choice to do T_5 . Substituting Θ into the inequality gives

$$v(F | [T_1 \rightarrow [T_2 \rightarrow \Theta]]) \leq v(F | [T_1 \rightarrow T_2] \rightarrow \Theta).$$

This inequality follows directly from Theorem 9, *i*).

The conditions for equality in Theorem 9 can be rewritten by using the definitions above in Corollary 5.

COROLLARY 5

- i) Equality in $v(F|[T_1 \rightarrow T_2] \rightarrow T_3) \geq v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$ is holding if and only if $\forall t_1$:

$$v(F|t_1, T_3) \leq v(F|t_1; \emptyset \rightarrow [T_2 \rightarrow T_3]).$$

- ii) Equality in $v(F|T_1 \rightarrow [T_2 \rightarrow T_3]) \geq v(F|T_1 \rightarrow (T_2, T_3))$ is holding if and only if $\forall t_1$:

$$v(F|t_1, T_2) \leq v(F|t_1; \emptyset \rightarrow (T_2, T_3)) \text{ and } \forall t_2 \left\{ \begin{array}{l} v(F|t_1, t_2; \emptyset \rightarrow T_3) = v(F|t_1, t_2) \text{ or} \\ v(F|t_1, t_2; \emptyset \rightarrow T_3) = v(F|t_1, t_2, T_3) \end{array} \right\}.$$

- iii) Equality in $v(F|T_1 \rightarrow (T_2, T_3)) \geq v(F|T_1, T_2, T_3)$ is holding if and only if $\forall t_1$:

$$v(F|t_1; \emptyset \rightarrow (T_2, T_3)) = v(F|t_1, T_2, T_3).$$

PROOF: Follows directly from the above definitions and Theorem 9. ■

If the tests are for free the problem of testing is simply to take as many tests as possible (see Theorem 7 in Section 3.2.3). If for example test T_2 is for free when tests T_1 and T_3 are executed then one should expect a higher expected utility by always doing T_2 before T_3 . Theorem 10 is covering situations like the one just described.

THEOREM 10

- i) If $U_{1,3}(c) = U_{1,2,3}(c)$, that is test 2 is free when tests 1 and 3 are carried out, then there is equality in Theorem 9 i) and one should always do test 2 before choosing to do test 3.
- ii) If $U_{1,2}(c) = U_{1,2,3}(c)$, that is test 3 is free when tests 1 and 2 are carried out, then there is equality in Theorem 9 ii) and one should always do tests 2 and 3 together.
- iii) If $U_1(c) = U_{1,2,3}(c)$, that is tests 2 and 3 are free when test 1 is carried out, then there is equality in Theorem 9 iii) and one should always do tests 1, 2 and 3 together.

PROOF of i) Looking term-wise for t_1 it suffices to show that

$$v(F|t_1, T_3) \leq v(F|t_1; T_2 \rightarrow T_3)$$

Substituting $U_{1,2,3}(c) = U_{1,3}(c)$ into the right-hand side and using Jensen's inequality twice gives

$$\begin{aligned}
v(F|t_1; T_2 \rightarrow T_3) &= \sum_{t_2} p(t_2|t_1) \max \left\{ \begin{array}{l} v(F|t_1, t_2) \\ v(F|t_1, t_2, T_3) \end{array} \right\} \\
&\geq \max \left\{ \begin{array}{l} v(F|t_1, T_2) \\ \sum_{t_2} \sum_{t_3} p(t_2|t_1) p(t_3|t_1, t_2) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1, t_2, t_3) U_{1,3}(c) \end{array} \right\} \\
&\geq \max \left\{ \begin{array}{l} v(F|t_1, T_2) \\ v(F|t_1, T_3) \end{array} \right\}
\end{aligned}$$

This shows that the condition for equality in Theorem 9 i) is met, and hence it is always better to do test 2 before doing test 3.

PROOF of ii) Looking term-wise for t_1 and t_2 it suffices to show that

$$v(F|t_1, t_2, T_3) \geq v(F|t_1, t_2).$$

Substituting $U_{1,2,3}(c) = U_{1,2}(c)$ into the left-hand side and applying Jensen's inequality over T_3 gives

$$\begin{aligned}
v(F|t_1, t_2, T_3) &= \sum_{t_3} p(t_3|t_1, t_2) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1, t_2, t_3) U_{1,2}(c) \\
&\geq \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1, t_2) U_{1,2}(c) \\
&= v(F|t_1, t_2)
\end{aligned} \tag{78}$$

It can be seen from (78) that $\forall t_1$

$$v(F|t_1, T_2) \leq v(F|t_1; \emptyset \rightarrow (T_2, T_3))$$

and that $\forall t_1, t_2$

$$v(F|t_1, t_2, T_3) \geq v(F|t_1, t_2).$$

Hence the conditions for equality in Theorem 9 ii) are fulfilled. This shows that it is always better to do tests 2 and 3 together.

PROOF of iii) Looking term-wise for t_1 it suffices to show that

$$v(F|t_1) \leq v(F|t_1, T_2, T_3).$$

Substituting $U_{1,2,3}(c) = U_1(c)$ into the right-hand side and applying Jensen's inequality over T_2, T_3 gives

$$\begin{aligned}
v(F|t_1, T_2, T_3) &= \sum_{t_2, t_3} p(t_2, t_3|t_1) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1, t_2, t_3) U_1(c) \\
&\geq v(F|t_1)
\end{aligned}$$

Hence the condition for equality in Theorem 9 ii) is fulfilled. This shows that it is better to do all three tests together.

■

$U_1(c) = U_{1,2}(c) = U_{1,3}(c) = U_{1,2,3}(c) = U(c)$ has the interpretation that the tests T_1 , T_2 and T_3 has a constant cost or is for free.

COROLLARY 6

If for all c , $U_1(c) = U_{1,2}(c) = U_{1,3}(c) = U_{1,2,3}(c)$ then equality is holding in all i), ii) and iii) of Theorem 9.

PROOF: Corollary 6 follows directly from Theorem 10 by inserting $U_1(c) = U_{1,2}(c) = U_{1,3}(c) = U_{1,2,3}(c)$.

The cost structure of the utility in Corollary 6 has a structure such that there is nothing to gain by having the choice to do tests T_2 or T_3 . For example if three tests are available Corollary 6 says that $v(F|[T_1 \rightarrow T_2] \rightarrow T_3) = v(F|T_1, T_2, T_3)$.

4.1.2 Graphical Representation - Choice Tree.

Event trees can be used as a graphical representation for the definitions and the notation that was introduced in Section 4.1.1. This graphical representation will help understand these decision problems.

The expected utility of having the choice to do or not to do T_2 after having observed T_1 is $v(F|T_1 \rightarrow T_2)$. This can be illustrated graphically by an event tree as shown in Figure 18.

Figure 18: Choice tree of $v(F|T_1 \rightarrow T_2)$.

The decision whether to do or not to do T_2 for each test-result t_1 is illustrated by having the decision node T_2 after every outcome t_{11} to t_{1z} . If T_2 is not chosen (the Not Do branch) then an act $f \in F$ is decided. On the other hand if T_2 is chosen (the Do branch) then a test result t_2 is also available before choosing an act $f \in F$.

Generally we have adopted the notation used by Ada Decision Systems [4]. In this notation square nodes represent decisions and circular nodes uncertain variables. An expected utility is associated with each node (see number in brackets in Figure 27). An optimal test-schedule-partition is a path in the decision tree made up of the nodes with the highest expected utility. The bold paths in Figure 27 are an example of such a test-schedule-partition.

REMARK: The variable a in Figure 18 represents the branch of the decision tree from decision node T_2 to node S , and similarly variable b is the branch from node F to node S . These variables represent the decision structure graphically and not the conditional probability structure in $v(F|T_1 \rightarrow T_2)$.

In the same way as in Figure 18 $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$ and $v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$ have graphical representations as shown in Figure 19 and in Figure 20 respectively.

Figure 19: Choice tree of $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$.

Figure 20: Choice tree of $v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$.

The graphical representations above can be used to give graphical understanding to expressions that looks like $v(F|[T_1 \rightarrow T_2] \rightarrow [(T_3, T_4) \rightarrow T_5])$. This is shown in Figure 21.

Figure 21. Graphical representation of $v(F|[T_1 \rightarrow T_2] \rightarrow [(T_3, T_4) \rightarrow T_5])$.

The test-schedule $[T_1 \rightarrow T_2] \rightarrow T_3$ is shown in Figure 19 above. An alternative way to represent this graphically is shown in Figure 22.

Figure 22: Test-outcome-partition of $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$.

Various tests in this test-schedule are executed for various test-outcomes. Fix a test schedule and consider the test-sequence (T_1, T_2, T_3) given the decision problem F and the probability p over S . Assume that test T_1 has been observed. Let a subset $A_{11} \subseteq \tau_1$ be defined as the subset where it is optimal to stop testing. For some other second subset A_{12} of τ_1 it is optimal to observe test T_2 , and for some third subset A_{13} of τ_1 one should observe test T_3 without T_2 . Now consider the set of outcome τ_2 . It can be partitioned into subsets in a similar manner, however each subset is also a function of t_1 since it is known at this stage. Hence $A_{22}(t_1) \subseteq \tau_2$ is the subset where it is optimal to stop after test T_2 and $A_{23}(t_1) \subseteq \tau_2$ is the subset where one should also do test T_3 . We might consider such a partition of the test outcomes to be a test-outcome-partition. It describes the test to do next for the various test-results.

4.2 Sequential Testing

In the previous Section 4.1 the choice to do an observation is discussed. The expected utilities $v(F|[T_1 \rightarrow T_2] \rightarrow T_3)$ and $v(F|T_1 \rightarrow [T_2 \rightarrow T_3])$ are defined. The arguments (e.g. $[[T_1 \rightarrow T_2] \rightarrow T_3]$ and $[T_1 \rightarrow [T_2 \rightarrow T_3]]$) of the expected utilities are test-schedules. Before we can create a good definition of this object let us describe various properties of it.

A test-sequence reflects the order of the tests in the test-schedule. For instance the test-schedule $[T_1 \rightarrow T_3] \rightarrow T_2$ implies the test-sequence (T_1, T_3, T_2) . The set of tests $\psi = \{T_1, T_2, T_3\}$ can be scheduled according to six test-sequences, corresponding to the six permutations.

DEFINITION 22 (Test-Sequence)

A test-sequence is an ordered set of tests.

■

The brackets and the arrows in $[[T_1 \rightarrow T_2] \rightarrow T_3]$ represent decision logic. One way to describe it is to use a binary tree.

DEFINITION 23 (Binary Tree)

A Binary Tree of degree m is a non-empty set $W \subset \bigcup_{k=0}^m \{0,1\}^k$ such that any word $w \in W$ satisfies:

- i. $w1 \in W \Rightarrow w0 \in W$ and $w \in W$
- ii. $w0 \in W \Rightarrow w1 \in W$

■

We denote the empty word, $\{0, 1\}^0$, by $w_\emptyset$. Since $W \neq \emptyset$ then $w_\emptyset \in W$. A word $w \in W$ is maximal if $w1 \notin W$.

Consider the decision logic of $\Phi = [[[[T_1 \rightarrow [T_2 \rightarrow T_3]] \rightarrow T_4] \rightarrow [T_5 \rightarrow T_6]]]$. Breaking down the decision logic of Φ it can be seen that it is built up of the permissible

elements shown in the binary tree in Figure 23. Note, $N = \{1, \dots, n\}$ corresponds to the set of tests $\psi = \{T_1, \dots, T_n\}$.

Figure 23: The elements of the decision logic.

Consider two different branch elements A, B at the same level in Figure 23, then it can be seen that A and B have properties such that $A \cap B = \emptyset$. Denote the decision logic as $\mathcal{L}_n$ where the n is the number of tests.

DEFINITION 24 ($\mathcal{L}$ - Decision Logic)

A set $\mathcal{L}$ of subsets of $N = \{1, \dots, n\}$ is a decision logic tree if, the elements of $\mathcal{L}$, written A_w , can be indexed by a binary tree, W , such that:

- i. $N = A_{\emptyset}$ and for all $w \in W$ such that $i \in A_{w0}$ and $j \in A_{w1}$ we have that $i < j$.
- ii. $A_{w0} \cap A_{w1} = \emptyset$ and $A_{w0} \cup A_{w1} = A_w$.
- iii. If $w \in W$ is maximal, then $|A_w| = 1$.

■

Note that the decision logic also covers groups of tests. For example, it is possible to consider T_2 and T_3 as one “meta-test” by expanding ψ to include $T_{n+1} = (T_2, T_3)$ such that $v(F|T_{n+1}) = v(F|T_2, T_3)$.

DEFINITION 25 (Φ - Decision Logic Map)

A decision logic map, Φ , is as a map from $\mathcal{L}$ into the set of words built from brackets, arrows and $T_1, \dots, T_n$ such that when w is not maximal then

$$\Phi(A_w) = [\Phi(A_{w0}) \rightarrow \Phi(A_{w1})]$$

and when w is maximal then

$$\Phi(A_w) = T_i \text{ when } A_w = \{i\}.$$

Finally, the decision logic map makes it possible to speak about a test-schedule as a function of a test-sequence and the decision logic $\mathcal{L}$.

DEFINITION 26 (Test-Schedule)

The test-schedule corresponding to a test sequence $(T_1, \dots, T_n)$ with a decision logic tree $\mathcal{L}$ is $\Phi(A_\emptyset)$.

Note that Definition 26 includes the trivial test schedule “no test” if, for example by taking T_1 to be the empty test, $T_1 = \emptyset$.

The arrow, $\rightarrow$, in a test schedule can be thought of as an operator just as $+$ in the addition of two real numbers. The arrow is a binary operator on the space of test schedules. We will use $\mathfrak{S}$ to denote the set of all test-schedules.

DEFINITION 27 (The Choice to do Operator)

The choice to do operator is the binary operator, $\rightarrow$, on the space of test-schedules $\mathfrak{S}$.

DEFINITION 28 (Utility-Greater & Utility-Equal)

Given two test-schedules Φ and Φ' , we say that Φ is utility-greater than Φ' , written $\Phi \geq \Phi'$, if for all $F \subseteq \mathbf{F}$, $v(F|\Phi) \geq v(F|\Phi')$. We say that Φ is utility-equal to Φ' , written $\Phi \equiv \Phi'$, if both $\Phi \leq \Phi'$ and $\Phi \geq \Phi'$.

When the meaning is clear we use dots ($[\dots]$ and $\dots$) to represent brackets, arrows and tests as shown below.

$$\Phi = [[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow T_4 \rightarrow T_5 = [\dots[T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_5,$$

and

$$\hat{\Phi} = T_1 \rightarrow [T_2 \rightarrow [T_3 \rightarrow [T_4 \rightarrow T_5]]] = T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_5] \dots].$$

Consider the test-sequence $(T_1, T_2, \dots, T_n)$ and let $1 < j < k < n$ (note $n \geq 3$). Define

- Φ_1 = any test-schedule of the test-sequence $(T_1, T_2, \dots, T_j)$,
- Φ_2 = any test-schedule of the test-sequence $(T_{j+1}, T_2, \dots, T_k)$ and
- Φ_3 = any test-schedule of the test-sequence $(T_{k+1}, T_2, \dots, T_n)$.

Theorem 9 is written more generally in terms of the test-schedules Φ_1 , Φ_2 and Φ_3 in Theorem 11. To give Theorem 9 (ii) and (iii) a meaningful interpretation for test-schedules define a function Γ as follows. Denote the test-sequence $(T_1, \dots, T_n)$ by $\mathcal{S}_{Seq}$, and denote the set of test-schedules built from a subset of $(T_1, \dots, T_n)$ by $\mathcal{S}_{Sched}$.

DEFINITION 29

Let Γ be the function which maps a test-schedule to the corresponding test-sequence, $\Gamma: \mathcal{S}_{Sched} \rightarrow \mathcal{S}_{Seq}$, such that

$$\Gamma(\Phi(T_1, \dots, T_n)) = (T_1, \dots, T_n).$$

■

In the following $\Gamma(\Phi_1 \rightarrow \Phi_2)$ will be written $\Gamma(\Phi_1, \Phi_2)$. Note that since

$$\Gamma([\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3) = \Gamma(\Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3]),$$

we can write $\Gamma(\Phi_1, \Phi_2, \Phi_3)$ for both cases without confusion. For example $\Gamma(\Phi_2, \Phi_3) = (T_{j+1}, \dots, T_n)$. Hence $v(F | \Phi_1 \rightarrow \Gamma(\Phi_2, \Phi_3))$ is

$$v(F | \Phi_1 \rightarrow (T_{j+1}, \dots, T_n)),$$

where $(T_{j+1}, \dots, T_n)$ is executed simultaneously as one meta-test. It is clear that

$$v(F | \Phi_1) \geq v(F | \Gamma(\Phi_1)). \quad (79)$$

THEOREM 11

- i) $v(F | [\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3) \geq v(F | \Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3]).$
- ii) $v(F | \Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3]) \geq v(F | \Phi_1 \rightarrow \Gamma(\Phi_2, \Phi_3)).$
- iii) $v(F | \Phi_1 \rightarrow \Gamma(\Phi_2, \Phi_3)) \geq v(F | \Gamma(\Phi_1, \Phi_2, \Phi_3)).$

PROOF of i): Expand (i) above up to having done test-schedule Φ_1 and let $(T'_{j_1}, \dots, T'_{j'})$ be any sub-sequence of Φ_1 . This gives a typical term of the left-hand side as

$$\max \left\{ \begin{array}{l} v(F | T'_{j_1}, \dots, T'_{j'}, \emptyset \rightarrow \Phi_3) \\ v(F | T'_{j_1}, \dots, T'_{j'}, \Phi_2 \rightarrow \Phi_3) \end{array} \right\}$$

and a typical term of the right-hand side as

$$\max \left\{ \begin{array}{l} v(F | t'_1, \dots, t'_{j'}) \\ v(F | t'_1, \dots, t'_{j'}; \Phi_2 \rightarrow \Phi_3) \end{array} \right\}.$$

Noting that the lower terms are the same it suffices to compare the upper terms. This gives

$$v(F | t'_1, \dots, t'_{j'}; \emptyset \rightarrow \Phi_3) \geq v(F | t'_1, \dots, t'_{j'}).$$

Hence, it follows that the inequality is proven.

PROOF of *ii*): Let $(T'_1, \dots, T'_{j'})$ be as above and let φ be a test-schedule. Looking term-wise for $t'_1, \dots, t'_{j'}$, it suffices to note that by (79),

$$v(F | t'_1, \dots, t'_{j'}; \emptyset \rightarrow \varphi) \geq v(F | t'_1, \dots, t'_{j'}; \emptyset \rightarrow \Gamma(\varphi)).$$

Inserting $\varphi = \Phi_2 \rightarrow \Phi_3$ into the inequality proves (*ii*).

PROOF of *iii*): Let $\varphi = \Phi_1 \rightarrow \Gamma(\Phi_2, \Phi_3)$. Note that

$$\Gamma(\Phi_1 \rightarrow \Gamma(\Phi_2, \Phi_3)) = \Gamma(\Phi_1, \Phi_2, \Phi_3).$$

It is clear that (79) gives

$$v(F | \varphi) \geq v(F | \Gamma(\varphi)).$$

This proves (*iii*).

■

We consider the test-schedule

$$\Phi = [T_1 \rightarrow T_2] \rightarrow [T_3 \rightarrow [T_4 \rightarrow T_5]].$$

This test-schedule can be written in terms of the test-schedules

$$\Phi_1 = [T_1 \rightarrow T_2] \text{ and } \Phi_2 = T_3 \rightarrow [T_4 \rightarrow T_5]$$

so that $\Phi = \Phi_1 \rightarrow \Phi_2$. We will now alter Φ slightly and compare with what we had. Let us alter the test-schedule by replacing the first test, test T_3 , of Φ_2 with the empty test $\emptyset$. Then the first test of Φ_2 , test T_3 , is inserted at the end of Φ_1 such that it gives

$$\Phi' = [[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow [\emptyset \rightarrow [T_4 \rightarrow T_5]].$$

This sort of shift operations will be written

$$\Phi' = (\Phi_1 + T_3) \rightarrow (-T_3 + \Phi_2).$$

The operation $(\Phi_1 + T_3)$ can be regarded as a right-hand side test-schedule addition, and $(-T_3 + \Phi_2)$ can be regarded as a left-hand side test-schedule addition (or subtraction). This notation will be used in Lemma 4. However first a formal definition is needed.

DEFINITION 30 (Test-Schedule Subtraction)

Let $\Phi(T_1, \dots, T_n)$ be a test-schedule for the test-sequence $(T_1, \dots, T_n)$. Define

$$\Phi(T_1, \dots, T_{i-1}, T_i, T_{i+1}, \dots, T_n) - T_i = \Phi(T_1, \dots, T_{i-1}, \emptyset, T_{i+1}, \dots, T_n).$$

Recall Φ_2 from above. By Definition 30, $\Phi_2 - T_3$ gives

$$\emptyset \rightarrow [T_4 \rightarrow T_5].$$

Note, Definition 30 gives

$$(\dots((\Phi(T_1, \dots, T_n) - T_1) - T_2) \dots - T_n) = \emptyset.$$

In order to make the statement of Lemma 4 more intuitive we introduce the use of the $+$ sign as an alternative for $\rightarrow$.

DEFINITION 31 (Test-Schedule Addition)

Let $\Phi(T_1, \dots, T_i)$ and $\Phi(T_{i+1}, \dots, T_n)$ be test-schedules for the test-sequences $(T_1, \dots, T_i)$ and $(T_{i+1}, \dots, T_n)$. Define addition such that

$$\Phi(T_1, \dots, T_i) + \Phi(T_{i+1}, \dots, T_n) = \Phi(T_1, \dots, T_i) \rightarrow \Phi(T_{i+1}, \dots, T_n)$$

Recall Φ_1 from above. By Definition 31, $(\Phi_1 + T_3)$ gives

$$[T_1 \rightarrow T_2] \rightarrow T_3.$$

Note, Definition 31 gives

$$(\dots(T_1 + T_2) + \dots + T_n) = [\dots[T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n$$

and

$$T_1 + (\dots + (T_{n-1} + T_n) \dots) = T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots].$$

In general note that $(\Phi(T_1, \dots, T_n) - T_i) + T_i \neq \Phi(T_1, \dots, T_n)$.

LEMMA 4

If Φ is written such that $\Phi = \Phi_1 \rightarrow \Phi_2$ where Φ_1 and Φ_2 are test-schedules such that for $1 \leq i \leq n$, Φ_1 is the test-schedule of $(T_1, T_2, \dots, T_i)$ and Φ_2 is the test-schedule of $(T_{i+1}, \dots, T_n)$ then

$$i) (\Phi_1 + T_{i+1}) + (\Phi_2 - T_{i+1}) \geq \Phi_1 + \Phi_2 \text{ and}$$

$$ii) (\Phi_1 - T_i) + (T_i + \Phi_2) \leq \Phi_1 + \Phi_2.$$

PROOF of i) Let us consider the term where the choice to do test T_{i+1} is made. Let $(T_1, \dots, T_{k+1})$ be a sub-sequence of $(T_1, T_2, \dots, T_i)$. A typical term of $\Phi_1 \rightarrow \Phi_2$ is

$$v(F | t'_1, \dots, t'_k; [T'_{k+1} \rightarrow \Phi_2]) = \sum_{t'_{k+1}} p(t'_{k+1} | t'_1, \dots, t'_k) \max \begin{cases} v(F | t'_1, \dots, t'_{k+1}) \\ v(F | t'_1, \dots, t'_{k+1}; \Phi_2). \end{cases}$$

The term of $(\Phi_1 + T_{i+1}) \rightarrow (\Phi_2 - T_{i+1})$ where the choice to do test T_{i+1} is made, is typically

$$\begin{aligned} & v(F | t'_1, \dots, t'_k; [T'_{k+1} \rightarrow T_{i+1}] \rightarrow (\Phi_2 - T_{i+1})) \\ &= \sum_{t'_{k+1}} p(t'_{k+1} | t'_1, \dots, t'_k) \max \begin{cases} v(F | t'_1, \dots, t'_{k+1}; \emptyset \rightarrow (\Phi_2 - T_{i+1})) \\ v(F | t'_1, \dots, t'_{k+1}; T_{i+1} \rightarrow (\Phi_2 - T_{i+1})). \end{cases} \end{aligned}$$

Comparing the upper and the lower terms in the maximisation above give

$$v(F | t'_1, \dots, t'_{k+1}; \emptyset \rightarrow (\Phi_2 - T_{i+1})) \geq v(F | t'_1, \dots, t'_{k+1})$$

and

$$v(F | t'_1, \dots, t'_{k+1}; T_{i+1} \rightarrow (\Phi_2 - T_{i+1})) = v(F | t'_1, \dots, t'_{k+1}; \Phi_2).$$

Hence it has been shown that $(\Phi_1 + T_{i+1}) \rightarrow (\Phi_2 - T_{i+1}) \geq \Phi_1 \rightarrow \Phi_2$.

PROOF of ii) Let us consider the term where the choice to do test T_i is made. Let $(T'_1, \dots, T'_k)$ be a sub-sequence of $(T_1, T_2, \dots, T_{i-1})$. A typical term of $\Phi_1 \rightarrow \Phi_2$ is

$$v(F | t'_1, \dots, t'_k; [T'_k \rightarrow T_i] \rightarrow \Phi_2) = \sum_{t'_k} p(t'_k | t'_1, \dots, t'_{k-1}) \max \begin{cases} v(F | t'_1, \dots, t'_k; \emptyset \rightarrow \Phi_2) \\ v(F | t'_1, \dots, t'_k; T_i \rightarrow \Phi_2). \end{cases}$$

The term of $(\Phi_1 - T_i) \rightarrow (T_i + \Phi_2)$ where the choice to do test T_i is made, is typically

$$\begin{aligned} & v(F | t'_1, \dots, t'_k; [T'_k \rightarrow \emptyset] \rightarrow [T_i \rightarrow \Phi_2]) \\ &= \sum_{t'_k} p(t'_k | t'_1, \dots, t'_{k-1}) \max \begin{cases} v(F | t'_1, \dots, t'_k; \emptyset \rightarrow [T_i \rightarrow \Phi_2]) \\ v(F | t'_1, \dots, t'_k; \emptyset \rightarrow [T_i \rightarrow \Phi_2]). \end{cases} \end{aligned}$$

Simplifying gives

$$\begin{aligned} & v(F | t'_1, \dots, t'_k; [T'_k \rightarrow \emptyset] \rightarrow [T_i \rightarrow \Phi_2]) \\ &= \sum_{t'_k} p(t'_k | t'_1, \dots, t'_{k-1}) \max \begin{cases} v(F | t'_1, \dots, t'_k) \\ v(F | t'_1, \dots, t'_k; T_i \rightarrow \Phi_2). \end{cases} \end{aligned}$$

Comparing upper and lower terms in the maximisation above gives

$$v(F | t'_1, \dots, t'_k; \emptyset \rightarrow \Phi_2) \geq v(F | t'_1, \dots, t'_k),$$

hence

$$(\Phi_1 - T_i) \rightarrow (T_i + \Phi_2) \leq \Phi_1 \rightarrow \Phi_2.$$

This proves the Lemma. ■

THEOREM 12

Let Φ be test-schedule of the test-sequence $(T_1, T_2, \dots, T_n)$. Then

$$i) [\dots[T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n \geq \Phi.$$

$$ii) T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots] \leq \Phi.$$

PROOF of i): Let $\Phi_1 = T_1$ and $\Phi_2 = (\Phi - T_1)$. Then Φ can be written

$$\Phi = \Phi_1 \rightarrow \Phi_2.$$

Apply Lemma 4 to Φ_1 and Φ_2 gives

$$[\Phi_1 + T_2] \rightarrow (\Phi_2 - T_2) \geq \Phi_1 \rightarrow \Phi_2$$

which is equivalent to

$$[T_1 \rightarrow T_2] \rightarrow ((\Phi - T_1) - T_2) \geq \Phi.$$

Reassign Φ_1 and Φ_2 such that

$$\Phi_1 = T_1 \rightarrow T_2 \text{ and } \Phi_2 = (\Phi - T_1) - T_2.$$

Note that Φ_1 is a test-schedule on (T_1, T_2) and Φ_2 is a test-schedule on $(T_3, \dots, T_n)$.

Apply Lemma 4 again gives

$$[[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow (((\Phi - T_1) - T_2) - T_3) \geq [T_1 \rightarrow T_2] \rightarrow ((\Phi - T_1) - T_2).$$

Hence it follows that

$$[[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow (((\Phi - T_1) - T_2) - T_3) \geq \Phi.$$

Iterate the steps above until $\Phi_2 = T_n$. This gives

$$[\dots[T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n \geq \Phi$$

which proves (i) of the Theorem.

PROOF of ii): Let $\Phi_1 = (\Phi - T_n)$ and $\Phi_2 = T_n$. Then Φ can be written

$$\Phi = \Phi_1 \rightarrow \Phi_2.$$

Apply Lemma 4 to Φ_1 and Φ_2 gives

$$[\Phi_1 - T_{n-1}] \rightarrow (T_{n-1} + \Phi_2) \leq \Phi_1 \rightarrow \Phi_2$$

which is equivalent to

$$((\Phi - T_n) - T_{n-1}) \rightarrow [T_{n-1} \rightarrow T_n] \leq \Phi$$

Reassign Φ_1 and Φ_2 such that

$$\Phi_1 = ((\Phi - T_n) - T_{n-1}) \text{ and } \Phi_2 = [T_{n-1} \rightarrow T_n].$$

Note that Φ_1 is a test-schedule on $(T_1, \dots, T_{n-2})$ and Φ_2 is a test-schedule on (T_{n-1}, T_n) .

Apply Lemma 4 again gives

$$(((\Phi - T_n) - T_{n-1}) - T_{n-2}) \rightarrow [T_{n-2} \rightarrow [T_{n-1} \rightarrow T_n]] \leq ((\Phi - T_n) - T_{n-1}) \rightarrow [T_{n-1} \rightarrow T_n].$$

Hence it follows that

$$(((\Phi - T_n) - T_{n-1}) - T_{n-2}) \rightarrow [T_{n-2} \rightarrow [T_{n-1} \rightarrow T_n]] \leq \Phi.$$

Iterate the steps above until $\Phi_1 = T_1$. This gives

$$T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots] \leq \Phi$$

which proves the Theorem. ■

We will now show that $X = (T_1 \rightarrow T_2, F)$ is a random variable,

$$X = (T_1 \rightarrow T_2, F) : S \rightarrow \mathbb{R}^1 \cup \mathbb{R}^2,$$

where F is the decision problem, i.e. the set of accessible acts and $\mathbb{R}^1 \cup \mathbb{R}^2$ is equipped with the σ -algebra $\sigma(\mathcal{B}(\mathbb{R}^1), \mathcal{B}(\mathbb{R}^2))$. Note, without specifying F , $T_1 \rightarrow T_2$ is not a random variable since it is not a function from only S to $\mathbb{R}^1 \cup \mathbb{R}^2$. The same reasoning shows that any test-schedule Φ containing n tests is a random variable such that

$$X = (\Phi, F) : S \rightarrow \mathbb{R}^1 \cup \mathbb{R}^2 \cup \dots \cup \mathbb{R}^n.$$

First of all Lemma 5 establishes that the value of information is a Borel measurable function.

LEMMA 5 (Value of Information is Borel Measurable)

The values of information

$$v(F | t_1) \text{ and } v(F | t_1, T_2)$$

are Borel measurable functions on $t_1 \in \mathbb{R}$.

PROOF: Since the expectations $v(F | t_1)$ and $v(F | t_1, T_2)$ are values in $\mathbb{R}$ given t_1 they can be considered to be functions so that

$$v(F | \cdot) : \mathbb{R} \rightarrow \mathbb{R} \text{ and } v(F | \cdot, T_2) : \mathbb{R} \rightarrow \mathbb{R}.$$

Via Theorem 4.2.8 in Dudley [5] it can be shown that

$$p(f^{-1}(c) | t_1) \text{ and } p(f^{-1}(c) | t_1, t_2)$$

are Borel measurable on $t_1 \in \mathbb{R}$ (note that $T_1 : S \rightarrow \mathbb{R}$ is measurable). Hence

$$p(f^{-1}(c) | t_1) U_1(c) \text{ and } p(f^{-1}(c) | t_1, t_2) U_{1,2}(c)$$

are Borel measurable.

By definition

$$v(F|t_1) = \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1) U_1(c)$$

$$v(F|t_1, T_2) = \sum_{t_2} p(t_2|t_1) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c)|t_1, t_2) U_{1,2}(c).$$

Since summation of Borel measurable functions and maximum over Borel measurable functions give "new" functions that are Borel measurable then $v(F|t_1)$ and $v(F|t_1, T_2)$ are Borel measurable. This proves the Theorem. ■

By definition the outcome of the variable, $X = (T_1 \rightarrow T_2, F)$, is an element in $\mathbb{R}^1 \cup \mathbb{R}^2$, and it is the value of information that determines whether the realisation of X is in $\mathbb{R}^1$ or $\mathbb{R}^2$. This is used in Theorem 13 to show that $\sigma(T_1 \rightarrow T_2, F) \subseteq \sigma(T_1, T_2)$.

THEOREM 13

Given two test schedules $T_1 \rightarrow T_2$ and (T_1, T_2) and the decision problem F then

$$\sigma(T_1 \rightarrow T_2, F) \subseteq \sigma(T_1, T_2) \subseteq \mathcal{F}$$

PROOF: Let X and Y be the random variables such that

$$X = (T_1 \rightarrow T_2, F) : S \rightarrow \mathbb{R}^1 \cup \mathbb{R}^2 \text{ and } Y = (T_1, T_2) : S \rightarrow \mathbb{R}^2,$$

Clearly $\sigma(Y) = \sigma(T_1, T_2) \subseteq \mathcal{F}$ since T_1, T_2 are Borel measurable. $X(s)$ is a random variable such that

$$X(s) = \begin{cases} t_1 & \text{if } v(F|t_1) \geq v(F|t_1, T_2) \\ (t_1, t_2) & \text{otherwise} \end{cases}.$$

We introduce a new function $\tilde{X} : \mathbb{R}^2 \rightarrow \mathbb{R}^1$ such that

$$\tilde{X}(t_1, t_2) = \begin{cases} t_1 & \text{if } v(F|t_1) \geq v(F|t_1, T_2) \\ (t_1, t_2) & \text{otherwise} \end{cases}.$$

By Lemma 5, $\tilde{X}$ is Borel measurable. We also see that $X(s) = \tilde{X}(Y(s))$. Hence

$$\sigma(X) \subseteq \sigma(Y).$$
■

From the proof of Theorem 13, it follows immediately that $X = (T_1 \rightarrow T_2, F)$ is a random variable since it shows that $\sigma(X) \subseteq \sigma(T_1, T_2)$.

COROLLARY 7 ("to do or not to do" is a Random Variable)

$X = (T_1 \rightarrow T_2, F)$ is a random variable. ■

4.2.1 Parallels to Savage

In the introduction to Savage's [1] decision theory, an act f was treated as a function $f: S \rightarrow \mathbb{R}$ and an observation T as a function $T: S \rightarrow \mathbb{R}$. Outcomes f and T are uncertain due to the uncertainty induced by the uncertainty on the set of all states S .

The situation described above as having done a test T_1 and then having the choice to do or not to do a second test T_2 returns a real number given the argument $F \subseteq \mathbb{F}$ and $s \in S$. Indeed when s and F are known the sequence of tests, and their results are determined. We write the function $[T_1 \rightarrow T_2]: F \times S \rightarrow \mathbb{R}$ from which it can be seen that $[T_1 \rightarrow T_2]$ is a random quantity given F due to the uncertainty induced by the uncertainty in S . Note that this is proven above in Section 4.2.

4.2.2 The Optimal Sequence And Its Computation Complexity.

Let us look at a set of tests $\psi = \{T_1, \dots, T_n\}$ where the number of tests is $\#\psi = n$. So far we have considered just one fixed test-sequence. Let π denote a fixed permutation of $\{1, \dots, n\}$. Recall from Theorem 12 that the test-schedule

$$[\dots[\emptyset \rightarrow T_{\pi_1}] \rightarrow \dots] \rightarrow T_{\pi_n} \quad (80)$$

gives an expected utility that is optimal given the test-sequence π . Let $Per(n)$ be the set of all $n!$ permutations of $\{1, \dots, n\}$. An optimal test-sequence can be fixed by trying all permutations in $Per(n)$. The optimal sequence is fixed by

$$\hat{\Phi} = \underset{\pi \in Per(n)}{\text{Argmax}} v(F | [\dots[\emptyset \rightarrow T_{\pi_1}] \rightarrow \dots] \rightarrow T_{\pi_n}). \quad (81)$$

Note however that there can be more than one test-sequence that gives a schedule that attains the maximum thus $\hat{\Phi}$ is not unique.

We will now look at the worst case computational complexity. To determine this we would need to know the complexity of (80). Recall the choice tree representation in Section 4.1.2 and consider the choice tree in Figure 24.

Figure 24. Choice tree of $v(F | [\emptyset \rightarrow T_1] \rightarrow T_2] \rightarrow T_3)$.

Since the expected value depends on which tests we choose to do, the computational complexity of the expected value is dominated by the number of choice-nodes. Hence,

Complexity = Constant · Number of Choice Nodes.

Let us assume that each test has m test outcomes or that m is the upper bound for the number of test outcomes for every test T_i .

LEMMA 6 (Number of Choice Nodes)

If $\forall i \in \{1, \dots, n\}$ we have that $|\tau_i| = m$ then the number of choice nodes is

$$\frac{(m+1)^n - 1}{m}.$$

PROOF: Let $i \in \{1, \dots, n\}$ and consider the i first tests of the test-schedule in (80). When $i = 1$ there is one choice-node of T_1 e.g. Figure 24. For $i = 2$ we have $m+1$ choice nodes of T_2 . For $i = 3$ we have another $m+1$ choice nodes of T_3 for each choice node of T_2 . In total there is $(m+1)^2$ choice-nodes of T_3 . Again for each choice node T_3 we have $m+1$ choice nodes of T_4 in total $(m+1)^3$. Iterating this process we find that there are $(m+1)^{i-1}$ choice nodes of test T_i . Summing gives

$$\sum_{i=1}^n (m+1)^{i-1} = \frac{(m+1)^n - 1}{m}$$

■

Later in the Greedy Counter Example on page 134 it will be shown that expected value depends on the order of the choice nodes. We will now look at computational complexity of (81).

THEOREM 14

The computational complexity of

$$\hat{\Phi} = \underset{\pi \in \text{Per}(n)}{\text{Argmax}} v(F | [\dots [\emptyset \rightarrow T_{\pi_1}] \rightarrow \dots] \rightarrow T_{\pi_n})$$

is $O(n![(m+1)^n-1]/m)$.

PROOF: Since the expected value of

$$v(F | [\dots [\emptyset \rightarrow T_{\pi_1}] \rightarrow \dots] \rightarrow T_{\pi_n})$$

depends on the test-sequence π and for each permutation a choice-tree must be computed then the computational complexity is

$$\text{Computation} = \text{Constant} \cdot \#\text{Permutations } \pi \cdot \#\text{Choice Nodes}$$

Using Lemma 6 and that there is $n!$ permutations give

$$\text{Computation} = \text{Constant} \cdot n![(m+1)^n-1]/m).$$

■

If n is large the computational complexity in Theorem 14 is dominated by $n!$. However for a problem where we have $n = 6$ tests and $m = 2$ outcomes we have

$$O(n![(m+1)^n-1]/m) = O(720 \cdot [729-1]/2) = O(262\,440).$$

A limitation when one optimises (81) is as follows. Let π be a permutation of $\{2, \dots, n\}$. Jensen's inequality gives

$$\begin{aligned} \max_{\pi \in \text{Per}(n \setminus 1)} v(F | [\dots [T_1 \rightarrow T_{\pi_1}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ \leq \sum_{t_1} p(t_1) \max_{\pi \in \text{Per}(n \setminus 1)} v(F | [\dots [t_1 \rightarrow T_{\pi_1}] \rightarrow \dots] \rightarrow T_{\pi_n}). \end{aligned}$$

Equality holds if all permutations in the maximum on the right hand-side gives the same expected value. In (81), for every test outcome $t_i \in \mathcal{T}_i$, we consider the same permutation π . If we allow the permutation to be a function of the test-outcomes we can possibly find better test-schedules than the one suggested in (81).

4.2.3 Sequential Sampling and Conditional Independence

De Groot [6] describes sequential sampling. He considers a decision problem specified by some parameter W in some parameter space $\Omega \subseteq S$, a set of accessible acts $F \subseteq \mathbf{F}$ and a utility function U . The decision-maker is permitted to observe a sequence of tests $T_1, T_2, \dots$ before choosing an act in F . Prior to observing, W has a prior distribution $\xi(w)$. The tests $T_1, T_2, \dots$ are assumed to be independent given $W = w$. The conditional distribution of each observation T_i is $f(\bullet | w)$, and the cost of observing is κ . It is not explicitly mentioned in De Groot [6], however implicitly it is assumed that the utility function has additive properties. By using the notation in Definition 13 the utility is restricted to $U_{1, \dots, k}(c) = U(c) - k \cdot \kappa$. In Section 4.2.4 sequential testing is considered under the restrictions that the tests are exchangeable and have constant cost κ . The conditional independence of the observations gives the conditional joint probability distribution:

$$p((t_1, \dots, t_n) | w) = \prod_{i=1}^n f(t_i | w).$$

Conditional independence for all w implies exchangeability such that the tests are exchangeable⁴. This shows that De Groot's [6] sequential testing falls into our general framework.

REMARK: Above it is argued that conditionally independence implies exchangeability. Note that the reverse implication is not true in general. De Finetti's famous result [7] shows that the reverse implication does hold for $\{0,1\}$ -valued tests.

The problem of optimal stopping is described in De Groot [6]. Optimal stopping problems are describing a repeated sequence of observations which are conditionally independent given W . The decision maker's problem is to find an optimal number of tests that he should do and then stop, or he is searching an expected number of tests to be performed before stopping. It can be seen that the expectation $v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_k] \dots])$ with k approaching infinity describes an equivalent problem to the one of optimal stopping. This follows Theorem 15 and Corollary 10. The number of tests is a random variable N induced by the uncertainty about S .

---

⁴ Later in Definition 32 exchangeability is defined.

De Groot's [6] conditions are not applicable to designing control software. It is rather unlikely that every test will have the same constant cost κ . It might be possible to condition on events for which the tests are exchangeable, however it is unlikely that all the tests in a control system would be exchangeable for all events. Hence, the assumption of conditional independence is rather strong.

4.2.4 Exchangeability and Additive-Cost

In the following discussion in this Section it will be assumed that:

1. The tests $T_1, \dots, T_n$ are exchangeable with respect to the joint distribution $p(T_1, \dots, T_n | f^{-1}(c))$ for all $c \in \mathbb{C}$ and $f \in F$.
2. The performance costs for test T_i is κ_i for all $i \in \{1, \dots, n\}$ and the utility has additive properties such that if k tests are performed then for every $k \in \{1, \dots, n\}$ we have that $U_{1, \dots, k}(c) = U(c) - \sum_{i=1}^k \kappa_i$.

By re-labelling the tests (if needed) such that the test-sequence is according to $\kappa_n \geq \dots \geq \kappa_1$, Theorem 15 below shows that the test-schedule $[\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n$ can be replaced by a simpler test-schedule since

$$v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]) = \max_{\pi \in \text{Per}(n)} v(F | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}).$$

Note that the test-sequence also is optimal. Hence an optimal test-sequence is fixed by ordering the tests so the test costs are in ascending order. As a special case, consider $\kappa_n = \dots = \kappa_1$ and simplify the utility function such that $U_{1, \dots, k}(c) = U(c) - k \cdot \kappa$. A heuristic argument for this special case of Theorem 15 is:

Let $k \in \{1, \dots, n\}$. For any permutation π the exchangeability gives

$$p(f^{-1}(c) | (T_1 = t_1, \dots, T_k = t_k)) = p(f^{-1}(c) | (T_1 = t_{\pi_1}, \dots, T_k = t_{\pi_k})).$$

the utility function $U_{1, \dots, k}(c) = U(c) - k \cdot \kappa$ gives

$$U_{1, \dots, k}(c) = U_{\pi_1, \dots, \pi_k}(c).$$

Equations similar to these two are the basic "building bricks" of $v(F | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n)$ and $v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots])$. Hence the result in Theorem 15 is not really unexpected.

We now define exchangeability. Consider a set of $\{A_i\}_{i=1, \dots, n}$ where 1_i is the indicator of A_i . Exchangeability means that the probability of a sequence, Q , of realisations of these indicators is independent of the order in which the realisations are made. More precisely, an arbitrary sequence, $Q \in \{0, 1\}^n$, of indicator functions $\{1_i\}_{i=1, \dots, n}$ is exchangeable with respect to p if for every permutation π of $1, \dots, n$ we have that

$$p((1_1, \dots, 1_n) = Q) = p((1_{\pi_1}, \dots, 1_{\pi_n}) = Q). \quad (82)$$

Elsewhere in literature (see Cooke [8]), (82) is how exchangeability is defined. Thinking in terms of our notation, this is formalised in the next definition.

DEFINITION 32 (p -exchangeability)

An arbitrary test-sequence $(T_1, \dots, T_n)$ is p -exchangeable if for all $(t_1, \dots, t_n)$,

$$p((T_1 \geq t_1, \dots, T_n \geq t_n)) = p((T_1 \geq t_{\pi_1}, \dots, T_n \geq t_{\pi_n})),$$

where $\pi_i = \pi(i)$.

■

We will in the following use (p, f, c) -exchangeability.

DEFINITION 33 ((p, f, c) -exchangeability)

Tests $(T_1, \dots, T_n)$ are (p, f, c) -exchangeable with respect to the conditional distribution $p((T_1, \dots, T_n) \mid f^{-1}(c))$ if for all $(t_1, \dots, t_n)$,

$$p((T_1 \geq t_1, \dots, T_n \geq t_n) \mid f^{-1}(c)) = p((T_1 \geq t_{\pi_1}, \dots, T_n \geq t_{\pi_n}) \mid f^{-1}(c)).$$

■

We will now show that (p, f, c) -exchangeability implies p -exchangeability. Note that this result is needed in the proof of Lemma 13 in Section 4.3. However, since we have the definitions fresh in memory it is shown here.

LEMMA 7

Assume that $\{T_1, \dots, T_n\}$ are (p, f, c) -exchangeable for all f, c . Then $\{T_1, \dots, T_n\}$ are exchangeable.

PROOF: Let π be a permutation of the test outcomes $\{t_1, \dots, t_n\}$. Using the (p, f, c) -exchangeability in Definition 33 gives

$$\begin{aligned} p(T_1 \geq t_1, \dots, T_n \geq t_n) &= \sum_c p(T_1 \geq t_1, \dots, T_n \geq t_n \mid f^{-1}(c)) p(f^{-1}(c)) \\ &= \sum_c p(T_1 \geq t_{\pi_1}, \dots, T_n \geq t_{\pi_n} \mid f^{-1}(c)) p(f^{-1}(c)) \\ &= p(T_1 \geq t_{\pi_1}, \dots, T_n \geq t_{\pi_n}). \end{aligned}$$

which when compared to the exchangeability in Definition 32 proves the Lemma

■

Note that some lemmas are needed in the proof of Theorem 15. They are in the end of this Section and called up on when needed.

THEOREM 15 (Exchangeable and Additive-Cost Tests)

If the tests $(T_1, \dots, T_n)$ are (p, f, c) -exchangeable for all $c \in \mathbf{C}, f \in \mathbf{F}$, the cost for test T_i is κ_i and the utility of doing k tests is $U_{1, \dots, k}(c) = U(c) - \sum_{i=1}^k \kappa_i$. By re-labelling (if needed) the tests such that $\kappa_n \geq \dots \geq \kappa_1$ then the test-schedule

$$T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]$$

gives the highest expected utility so that

$$v(\mathbf{F} | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]) = \max_{\pi \in \text{Per}(n)} v(\mathbf{F} | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}).$$

PROOF: By Lemma 12 sequencing the tests according to $\kappa_n \geq \dots \geq \kappa_1$ gives

$$v(\mathbf{F} | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) = \max_{\pi \in \text{Per}(n)} v(\mathbf{F} | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}).$$

Hence sequencing the tests according to $\kappa_n \geq \dots \geq \kappa_1$ gives an optimal permutation. Recall from Theorem 12 that

$$v(\mathbf{F} | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) \geq v(\mathbf{F} | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]).$$

Using the optimal sequence it will be shown that equality holds. It is sufficient to show that

$$\begin{aligned} & v(\mathbf{F} | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) \\ &= v(\mathbf{F} | [T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_i] \dots]] \rightarrow [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \end{aligned} \quad (83)$$

for any $i \in \{2, \dots, n-1\}$, because $i = n-1$ is the claim of the theorem. We show (83) by induction. It will first be proven for $i = 2$. Consider

$$v(\mathbf{F} | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n). \quad (84)$$

Lemma 10 gives

$$v(\mathbf{F} | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) = v(\mathbf{F} | T_1 \rightarrow [\dots [T_2 \rightarrow T_3] \rightarrow \dots] \rightarrow T_n).$$

Hence (83) holds for $i = 2$. Assume that (83) holds for i . It will be shown that it must also be true for $i + 1$. The inductive step follows if we show

$$\begin{aligned} & v(\mathbf{F} | [T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_i] \dots]] \rightarrow [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ &= v(\mathbf{F} | [T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_{i+1}] \dots]] \rightarrow [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{aligned}$$

By definition the first term is equal to

$$\sum_{t_1} p(t_1) \max \left\{ \begin{array}{l} v(F | t_1) \\ \sum_{t_2 \in T_2} p(t_2 | t_1) \max \left\{ \begin{array}{l} v(F | t_1, t_2) \\ \vdots \\ \sum_{t_i} p(t_i | t_1 \dots t_{i-1}) \max \left\{ \begin{array}{l} v(F | t_1, \dots, t_i) \\ v(F | t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n), \end{array} \right. \end{array} \right. \end{array} \right.$$

and the second term is equal to

$$\sum_{t_1} p(t_1) \max \left\{ \begin{array}{l} v(F | t_1) \\ \sum_{t_2 \in T_2} p(t_2 | t_1) \max \left\{ \begin{array}{l} v(F | t_1, t_2) \\ \vdots \\ \sum_{t_i} p(t_i | t_1 \dots t_{i-1}) \max \left\{ \begin{array}{l} v(F | t_1, \dots, t_i) \\ v(F | t_1, \dots, t_i; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{array} \right. \end{array} \right. \end{array} \right.$$

These two terms only differ in the last term. By applying Lemma 10 to the last term of the first one gives

$$\begin{aligned} & v(F | t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ &= v(F | t_1, \dots, t_i; T_{i+1} \rightarrow [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{aligned} \quad (85)$$

Hence, the last terms are the same and the theorem is proven. ■

DEFINITION 34 (Utility-Associative)

The operator $\rightarrow$ is utility-associative for a set of acts $F \subseteq \mathbf{F}$ if

$$[\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3 \equiv \Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3]$$

for any test-schedules Φ_1, Φ_2 and Φ_3 such that (Φ_1, Φ_2, Φ_3) do not share any test. ■

Using the results of Lemma 12 and Theorem 15 together with Theorem 9 we conclude.

COROLLARY 8

Under the conditions of Theorem 15,

$$v(F | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) = v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]).$$

PROOF: Given a set of tests $\{T_1, \dots, T_n\}$ sequenced so that $\kappa_n \geq \dots \geq \kappa_1$ then Lemma 12 gives

$$v(F | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) = \max_{\pi \in \text{Per}(n)} v(F | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n})$$

and Theorem 15 gives

$$v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]) = \max_{\pi \in \text{Per}(n)} v(F | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n})$$

such that

$$v(F | [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) = v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]).$$

■

We now motivate Corollary 9. By Corollary 8 we have

$$v(F | [T_1 \rightarrow T_2] \rightarrow T_3) = v(F | T_1 \rightarrow [T_2 \rightarrow T_3]).$$

Considering four tests then Theorem 11 gives

$$[[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow T_4 \geq [T_1 \rightarrow [T_2 \rightarrow T_3]] \rightarrow T_4 \geq T_1 \rightarrow [T_2 \rightarrow [T_3 \rightarrow T_4]].$$

(Recall the meaning of $\Phi \geq \Phi'$ in Section 4.2) Using the Remark above gives that

$$[[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow T_4 = T_1 \rightarrow [T_2 \rightarrow [T_3 \rightarrow T_4]]$$

hence

$$[[T_1 \rightarrow T_2] \rightarrow T_3] \rightarrow T_4 = [T_1 \rightarrow [T_2 \rightarrow T_3]] \rightarrow T_4 = T_1 \rightarrow [T_2 \rightarrow [T_3 \rightarrow T_4]].$$

This illustrates how to prove Corollary 9.

COROLLARY 9 (The choice-to-do is Utility-Associative)

For the test sequence in which the tests are ordered by increasing costs, under the conditions of Theorem 15 the choice-to-do operator is utility-associative.

PROOF: Consider the test-sequence $(T_1, T_2, \dots, T_n)$ and let $1 < j < k < n$ (note $n \geq 3$). Define

- Φ_1 = any test-schedule over the test-sequence $(T_1, T_2, \dots, T_j)$,
- Φ_2 = any test-schedule over the test-sequence $(T_{j+1}, \dots, T_k)$ and
- Φ_3 = any test-schedule over the test-sequence $(T_{k+1}, \dots, T_n)$.

Theorem 11 gives

$$[\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3 \geq \Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3], \quad (86)$$

Theorem 12 i) gives

$$[\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n \geq [\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3$$

and Theorem 12 ii) gives

$$\Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3] \geq T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots].$$

Hence we have

$$\begin{aligned} [\dots [T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n &\geq [\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3 \geq \Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3] \\ &\geq T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]. \end{aligned}$$

Since the tests are exchangeable and are sequenced according ascending test costs (e.g. $\kappa_n \geq \dots \geq \kappa_1$) Corollary 8 gives

$$\begin{aligned} [\dots [T_1 \rightarrow T_2] \rightarrow \dots T_j] \rightarrow T_{j+1} \dots \rightarrow T_k] \rightarrow T_{k+1} \dots \rightarrow T_n &\equiv \\ T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow [T_j \dots \rightarrow [T_k \dots \rightarrow T_n] \dots] \dots], \end{aligned}$$

so that

$$[\Phi_1 \rightarrow \Phi_2] \rightarrow \Phi_3 \equiv \Phi_1 \rightarrow [\Phi_2 \rightarrow \Phi_3].$$

■

We know strengthen Assumption 2 above to assume that the performance costs are the same for each test (i.e. $\kappa_i = \kappa$). Then the utility is $U_{1,\dots,k}(c) = U(c) - k \cdot \kappa$.

DEFINITION 35 (Utility-Commutative)

The operator $\rightarrow$ is utility-commutative for a set of acts $F \in \mathbf{F}$ if

$$\Phi_1 \rightarrow \Phi_2 \equiv \Phi_2 \rightarrow \Phi_1$$

for any test-schedules Φ_1 and Φ_2 of the tests $\{T_1, T_2, \dots, T_n\}$ that do not share the same test.

■

Theorem 15 holds and it can be seen that it does not matter in which order the tests are sequenced (for any permutation π we have that $\kappa_{\pi_n} \geq \dots \geq \kappa_{\pi_1}$).

COROLLARY 10 (The choice to do is Utility-Commutative)

For the test-sequence ordered arbitrarily under the conditions of Theorem 15 and the assumption that the cost is $\kappa_i = \kappa$ for all tests T_i , $i \in \{1, \dots, n\}$ then the operator $\rightarrow$ is utility-commutative and utility-associative with respect to the choice-to-do operator.

PROOF: Utility-associativity follows from Corollary 9. Utility-commutative follows from Theorem 15 with the additional assumption that $\kappa_i = \kappa$. Let

Φ_1 = any test-schedule of the test-sequence $(T_1, T_2, \dots, T_j)$ and

Φ_2 = any test-schedule of the test-sequence $(T_{j+1}, T_2, \dots, T_n)$

Since $\kappa_i = \kappa$ for all tests any test-sequence can be used. In addition Theorem 12 gives

$$[\dots[T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n \geq \max\{\Phi_1 \rightarrow \Phi_2, \Phi_2 \rightarrow \Phi_1\}$$

and

$$T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots] \leq \min\{\Phi_1 \rightarrow \Phi_2, \Phi_2 \rightarrow \Phi_1\}.$$

Corollary 8 gives

$$\begin{aligned} & [\dots[T_1 \rightarrow T_2] \rightarrow \dots T_j] \rightarrow T_{j+1} \dots \rightarrow T_k] \rightarrow T_{k+1} \dots \rightarrow T_n \equiv \\ & T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_j \dots \rightarrow [T_k \dots \rightarrow T_n] \dots], \end{aligned}$$

so that

$$\Phi_1 \rightarrow \Phi_2 \equiv \Phi_2 \rightarrow \Phi_1.$$

■

The following lemmas in the end of this Section are used in the proof of Theorem 15.

LEMMA 8

$$\begin{aligned} & v(F | t_1, \dots, t_i, [\dots[T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ & \geq v(F | t_1, \dots, t_i, [\dots[T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_{n-1}). \end{aligned}$$

PROOF: This trivial inequality follows directly from Definition 16 and Definition 17.

■

LEMMA 9

Under the assumptions of Theorem 15 then for any $i \in \{1, \dots, n\}$:

$$\begin{aligned} & v(F | t_1, \dots, t_{i+1}, [\dots[\emptyset \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n) \\ & \leq v(F | t_1, \dots, t_{i+1}, [\dots[\emptyset \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_{n-1}). \end{aligned}$$

PROOF: For any $i, k \in \{1, \dots, n\}$ such that $i+2 < k \leq n$ the ordering $\kappa_n \geq \dots \geq \kappa_1$ gives

$$\sum_{l=i+2}^{k-1} \kappa_l \leq \sum_{l=i+3}^k \kappa_l.$$

Substituting this into the utility gives

$$U_{1, \dots, i+1, i+3, \dots, k}(c) \leq U_{1, \dots, k-1}(c).$$

If the term

$$v(F | t_1, \dots, t_{i+1}, [\dots[\emptyset \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n)$$

is expanded one can see that a typical term in this expansion is

$$v(F | t_1, \dots, t_{i+1}, t_{i+3}, \dots, t_k) = \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | (t_1, \dots, t_{i+1}, t_{i+3}, \dots, t_k)) U_{1, \dots, i+1, i+3, \dots, k}(c)$$

The exchangeability gives

$$p(f^{-1}(c) | (t_1, \dots, t_{i+1}, t_{i+3}, \dots, t_k)) = p(f^{-1}(c) | (t_1, \dots, t_{k-1}))$$

hence it follows that

$$\begin{aligned} & \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | (t_1, \dots, t_{i+1}, t_{i+3}, \dots, t_k)) U_{1, \dots, i+1, i+3, \dots, k}(c) \\ & \leq \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | (t_1, \dots, t_{k-1})) U_{1, \dots, k-1}(c). \end{aligned}$$

Since this is true for all test-schedules with more than $i+1$ tests then it follows that

$$\begin{aligned} & v(F | t_1, \dots, t_{i+1}, [\dots [\emptyset \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n) \\ & \leq v(F | t_1, \dots, t_{i+1}, [\dots [\emptyset \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_{n-1}). \end{aligned}$$

■

When the tests are exchangeable and have additive costs as in assumption 2 above Lemma 9 says it is better to do a set of tests that have lower index first.

LEMMA 10

Under the assumptions of Theorem 15, then for any $i \in \{0, \dots, n-1\}$:

$$\begin{aligned} & v(F | t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ & = v(F | t_1, \dots, t_i; T_{i+1} \rightarrow [[\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n)). \end{aligned}$$

PROOF: Looking at the sum over t_{i+1} gives

$$\begin{aligned} & v(F | t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ & = \sum_{t_{i+1}} p(t_{i+1} | t_1, \dots, t_i) \max \begin{cases} v(F | t_1, \dots, t_{i+1}; [\dots [\emptyset \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n), \end{cases} \end{aligned}$$

and applying Lemma 9 to the upper term in the max gives

$$\leq \sum_{t_{i+1}} p(t_{i+1} | t_1, \dots, t_i) \max \begin{cases} v(F | t_1, \dots, t_{i+1}; [\dots [\emptyset \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_{n-1}) \\ v(F | t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{cases} \quad (87)$$

By definition the upper term is equal to

$$\max \begin{cases} v(F | t_1, \dots, t_{i+1}; [\dots [\emptyset \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_{n-1}) \\ v(F | t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_{n-1}). \end{cases}$$

Applying Lemma 9 again and inserting this into (87) gives

$$\leq \sum_{t_{i+1}} p(t_{i+1} | t_1, \dots, t_i) \max \begin{cases} v(F | t_1, \dots, t_{i+1}; [\dots [\emptyset \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_{n-2}) \\ v(F | t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_{n-1}) \\ v(F | t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{cases}$$

Lemma 8 gives

$$\begin{aligned} & v(F \mid t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_{n-1}) \\ & \leq v(F \mid t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{aligned}$$

This gives

$$= \sum_{t_{i+1}} p(t_{i+1} \mid t_1, \dots, t_i) \max \begin{cases} v(F \mid t_1, \dots, t_{i+1}; [\dots [\emptyset \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_{n-2}) \\ v(F \mid t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{cases} \quad (88)$$

Iterating the argument between (87) and (88) $n-(i+3)$ times gives

$$\begin{aligned} & v(F \mid t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ & \leq \sum_{t_{i+1}} p(t_{i+1} \mid t_1, \dots, t_i) \max \begin{cases} v(F \mid t_1, \dots, t_{i+1}) \\ v(F \mid t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{cases} \end{aligned}$$

By definition

$$\begin{aligned} & v(F \mid t_1, \dots, t_i; T_{i+1} \rightarrow [[\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n]) \\ & = \sum_{t_{i+1}} p(t_{i+1} \mid t_1, \dots, t_i) \max \begin{cases} v(F \mid t_1, \dots, t_{i+1}) \\ v(F \mid t_1, \dots, t_{i+1}; [\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n). \end{cases} \end{aligned}$$

Hence it has been shown that

$$\begin{aligned} & v(F \mid t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ & \leq v(F \mid t_1, \dots, t_i; T_{i+1} \rightarrow [[\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n]). \end{aligned}$$

Theorem 12 gives that

$$\begin{aligned} & v(F \mid t_1, \dots, t_i; [\dots [T_{i+1} \rightarrow T_{i+2}] \rightarrow \dots] \rightarrow T_n) \\ & \geq v(F \mid t_1, \dots, t_i; T_{i+1} \rightarrow [[\dots [T_{i+2} \rightarrow T_{i+3}] \rightarrow \dots] \rightarrow T_n]). \end{aligned}$$

Hence equality holds and the Lemma is proven. ■

Lemma 11 is used to prove Lemma 12 that is used in the proof of Theorem 15.

LEMMA 11

Let $\mathbf{X} = (x_1, \dots, x_n)$ and consider a function $h \rightarrow \mathbb{R}$. Assume for all $i \in \{1, \dots, n-1\}$ that if $x_{(i+1)} \geq x_{(i)}$ then

$$h(x_1, \dots, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \dots, x_{(n)}) \geq h(x_1, \dots, x_{i-1}, x_{i+1}, x_{i+2}, x_i, \dots, x_n). \quad (89)$$

Re-label so that $x_1 \leq \dots \leq x_n$ (if needed) then

$$h(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n) = \max_{\pi \in \text{Per}(n)} h(x_{\pi_1}, \dots, x_{\pi_n})$$

where $\pi_i = \pi(i)$.

PROOF: The proof is an algorithm to re-label the tests so that (89) is satisfied for all transpositions of two consecutive variables x_i, x_{i+1} . Start with any fixed permutation π of $\mathbf{X}$.

Choose x_1 so that

$$x_1 = \min\{x_1, \dots, x_n\}.$$

Assume that $x_1 = x_{\pi_i}$. Since $x_{\pi_i} \leq \min\{x_{\pi_1}, \dots, x_{\pi_{i-1}}\}$ then from (89) we can increase the value of h applying the following algorithm:

Exchange x_1 with $x_{\pi_{i-1}}$. This gives

$$h(x_{\pi_1}, \dots, x_{\pi_{i-2}}, x_1, x_{\pi_{i-1}}, x_{\pi_{i+1}}, \dots, x_{\pi_n}) \geq h(x_{\pi_1}, \dots, x_{\pi_{i-2}}, x_{\pi_{i-1}}, x_1, x_{\pi_{i+1}}, \dots, x_{\pi_n})$$

Now exchanging x_1 with $x_{\pi_{i-2}}$ gives

$$h(x_{\pi_1}, \dots, x_1, x_{\pi_{i-2}}, x_{\pi_{i-1}}, x_{\pi_{i+1}}, \dots, x_{\pi_n}) \geq h(x_{\pi_1}, \dots, x_{\pi_{i-2}}, x_1, x_{\pi_{i-1}}, x_{\pi_{i+1}}, \dots, x_{\pi_n})$$

$\vdots$

Repeating in this way we finally exchange x_1 with x_{π_1} :

$$h(x_1, x_{\pi_1}, \dots, x_{\pi_{i-1}}, x_{\pi_{i+1}}, \dots, x_{\pi_n}) \geq h(x_{\pi_1}, x_1, x_{\pi_2}, \dots, x_{\pi_{i-1}}, x_{\pi_{i+1}}, \dots, x_{\pi_n}).$$

Having moved x_1 to the front, we now apply the same algorithm to the remaining $(n-1)$ variables to move

$$x_2 = \min(\{x_1, \dots, x_n\} \setminus \{x_1\})$$

to the second place. Iterating this argument $n-2$ more times proves the Lemma. ■

The next result that will be shown is that a test-sequence where the tests are sequenced according to ascending test-costs is an optimal one.

LEMMA 12

The assumptions of Theorem 15 give

$$v(F | [\dots[T_1 \rightarrow T_2] \rightarrow \dots] \rightarrow T_n) = \max_{\pi \in \text{Per}(n)} v(F | [\dots[T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}).$$

PROOF: Consider any permutation π of the tests ψ . $\forall i \in \{1, \dots, n\}$, recall that a decision to do or not to do test $T_{\pi_{i+1}}$ in $v(F | [\dots[T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n})$ is

$$\begin{aligned} & v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}; [\dots[T_{\pi_i} \rightarrow T_{\pi_{i+1}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ &= \sum_{t_{\pi_i}} P(t_{\pi_i} | t_{\pi_1}, \dots, t_{\pi_{i-1}}) \max \begin{cases} v(F | t_{\pi_1}, \dots, t_{\pi_i}; [\dots[\emptyset \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ v(F | t_{\pi_1}, \dots, t_{\pi_i}; [\dots[T_{\pi_{i+1}} \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}). \end{cases} \end{aligned}$$

Exchanging the order of T_{π_i} and $T_{\pi_{i+1}}$ then a decision to do or not to do T_{π_i} is

$$\begin{aligned} & v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}; [\dots[[T_{\pi_{i+1}} \rightarrow T_{\pi_i}] \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ &= \sum_{t_{\pi_{i+1}}} P(t_{\pi_{i+1}} | t_{\pi_1}, \dots, t_{\pi_{i-1}}) \max \begin{cases} v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}, t_{\pi_{i+1}}; [\dots[[\emptyset \rightarrow T_{\pi_i}] \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ v(F | t_{\pi_1}, \dots, t_{\pi_{i+1}}; [\dots[T_{\pi_i} \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}). \end{cases} \end{aligned}$$

Considering the upper term of the max in the equations above. If $t_{\pi_i} = t_{\pi_{i+1}}$ and $\kappa_{\pi_i} \leq \kappa_{\pi_{i+1}}$ then:

$$\begin{aligned} & v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}, t_{\pi_{i+1}}; [\dots[\emptyset \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ & \leq v(F | t_{\pi_1}, \dots, t_{\pi_i}; [\dots[\emptyset \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}). \end{aligned}$$

Considering the lower term of the max in the equations above for all $t_{\pi_i} \in \tau_{\pi_i}$ and all $t_{\pi_{i+1}} \in \tau_{\pi_{i+1}}$ gives

$$\begin{aligned} & v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}, t_{\pi_i}, t_{\pi_{i+1}}; [\dots[\emptyset \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ &= v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}, t_{\pi_{i+1}}, t_{\pi_i}; [\dots[\emptyset \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \end{aligned}$$

because of the additive properties of the utility and the exchangeability of the tests with respect to $P(T_1, \dots, T_n | f^{-1}(c))$ (note: both tests T_{π_i} and $T_{\pi_{i+1}}$ are always executed). Taking the expectation over T_{π_i} and $T_{\pi_{i+1}}$ respectively gives

$$\begin{aligned} & v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}; [\dots[T_{\pi_i} \rightarrow T_{\pi_{i+1}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ & \geq v(F | t_{\pi_1}, \dots, t_{\pi_{i-1}}; [\dots[[T_{\pi_{i+1}} \rightarrow T_{\pi_i}] \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}). \end{aligned}$$

It follows that

$$\begin{aligned} & v(F | [\dots[[\dots[T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_i}] \rightarrow T_{\pi_{i+1}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \\ & \geq v(F | [\dots[[\dots[[\dots[T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_{i-1}}] \rightarrow T_{\pi_{i+1}}] \rightarrow T_{\pi_i}] \rightarrow T_{\pi_{i+2}}] \rightarrow \dots] \rightarrow T_{\pi_n}) \end{aligned}$$

if and only if $\kappa_{\pi_i} \leq \kappa_{\pi_{i+1}}$. Hence it has been shown that the permutation of the tests should be chosen so that the cost is in ascending order. The Lemma is established by using Lemma 11 and re-labelling the tests so that $\kappa_n \geq \dots \geq \kappa_1$.

■

4.3 Greedy Algorithms and The Optimal Test Order

By using a greedy selection strategy it is possible to suggest an approximate algorithm to obtain the test order in shorter than $O(n!)$ time. We use $N = \{1, \dots, n\}$ to refer to the set of tests $\Psi = \{T_1, \dots, T_n\}$. A greedy algorithm to sequence before testing is given below.

Algorithm Greedy-Sequence before Test:

$$\dot{T}_{(1)} = \underset{i \in N}{\text{Argmax}} \ v(F | T_i)$$

$$\dot{T}_{(2)} = \underset{i \in N \setminus \{(1)\}}{\text{Argmax}} \ v(F | \dot{T}_{(1)}, T_i)$$

$\vdots$

$$\dot{T}_{(n)} = \underset{i \in N \setminus \{(1), \dots, (n-1)\}}{\text{Argmax}} \ v(F | \dot{T}_{(1)}, \dots, \dot{T}_{(n-1)}, T_i)$$

The output is the test-sequence $(\dot{T}_{(1)}, \dots, \dot{T}_{(n)})$.

For simplicity we write $(\dot{T}_1, \dots, \dot{T}_n)$ instead of $(\dot{T}_{(1)}, \dots, \dot{T}_{(n)})$ when the meaning is clear. This algorithm selects the first test as the one that gives the highest conditional expected utility when conditioning on only one test. Then it sequences the second test by selecting a test that maximises the conditional expected utility when conditioning on two tests including the first.

NOTATION: The i 'th test in the *Greedy-Sequence before Test* is denoted by $\dot{T}_{(i)}$ or simply $\dot{T}_i$ for $i \in (1, \dots, n)$.

Using Theorem 12 and the output of the *Greedy-Sequence before Test* algorithm, the highest obtainable expected utility is obtained by the test-schedule

$$\Phi_{\text{Gr}} = [\dots [\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dots] \rightarrow \dot{T}_{(n)}.$$

We will now show that Φ_{Gr} is optimal under the conditions of Theorem 15.

LEMMA 13 (Optimal Sequence by Sequence before Test)

Under the conditions of Theorem 15 the Greedy-Sequence before Test algorithm orders the tests according to $\kappa_n \geq \dots \geq \kappa_{(1)}$ so that

$$v(F | \emptyset \rightarrow [\dot{T}_{(1)} \rightarrow [\dots \rightarrow \dot{T}_{(n)}] \dots]) = \max_{\pi \in \text{Per}(n)} v(F | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}).$$

PROOF: From Theorem 15 it follows that if the tests are permuted according to ascending test cost $\kappa_n \geq \dots \geq \kappa_1$ then

$$v(F | \emptyset \rightarrow [T_1 \rightarrow [\dots \rightarrow T_n] \dots]) = \max_{\pi \in \text{Per}(n)} v(F | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}).$$

It suffices to show that the Greedy-Sequence before Test algorithm orders the tests according to $\kappa_n \geq \dots \geq \kappa_1$. The first test selected by the algorithm is $\dot{T}_{(1)}$ such that

$$\begin{aligned} v(F | \dot{T}_{(1)}) &= \max_{i \in N} v(F | T_i) \\ &= \max_{i \in N} \left\{ \left[\sum_{t_i} p(t_i) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | t_i) U(c) \right] - \kappa_i \right\}. \end{aligned}$$

From Lemma 7 the (p, f, c) -exchangeability in the conditions of Theorem 15 implies that tests in ψ are exchangeable. Hence the sum in the brackets is constant for all t_i and $i \in N$. This gives

$$\begin{aligned} v(F | \dot{T}_{(1)}) &= \sum_{t_i} p(t_i) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | t_i) U(c) + \max_{i \in N} \{-\kappa_i\} \\ &= \sum_{t_i} p(t_i) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | t_i) U(c) - \min_{i \in N} \{\kappa_i\}. \end{aligned}$$

Hence the first test $\dot{T}_{(1)}$ sequenced by the algorithm is the one with the lowest cost

$$\kappa_{(1)} = \min_{i \in N} \{\kappa_i\}.$$

Let $i \in (1, \dots, n)$ and assume that the algorithm is optimal for the i first tests. It will be shown that it must also be optimal for the $i+1$ first tests. Consider

$$v(F | \dot{T}_{(1)}, \dots, \dot{T}_{(i)}, \dot{T}_{(i+1)}) = \max_{j \in N \setminus \{(1), \dots, (i)\}} \{v(F | \dot{T}_{(1)}, \dots, \dot{T}_{(i)}, T_j)\}.$$

Lemma 7 implies exchangeability. Hence,

$$\sum_{i_{(1)}, \dots, i_{(i)}, t_j} p(i_{(1)}, \dots, i_{(i)}, t_j) \max_{f \in F} \sum_{c \in C} p(f^{-1}(c) | i_{(1)}, \dots, i_{(i)}, t_j) U(c) - \sum_{k=1}^i \kappa_{(k)} - \min_{j \in N \setminus \{(1), \dots, (i)\}} \{\kappa_j\}.$$

This gives

$$\kappa_{(i+1)} = \min_{j \in N \setminus \{(1), \dots, (i)\}} \{\kappa_j\}.$$

By induction the Lemma is proven.

Sequence before test entails that test before sequence is possible.

Algorithm Greedy-Test before Sequence:

$$T_{(1)} = \text{Argmax}_{j \in N} v(F | \emptyset \rightarrow T_j)$$

$$T_{(2)} = \text{Argmax}_{j \in N \setminus \{(1)\}} v(F | T_{(1)} \rightarrow T_j)$$

$\vdots$

$$T_{(i)} = \text{Argmax}_{j \in N \setminus \{(1), \dots, (i-1)\}} v(F | (T_{(1)}, \dots, T_{(i-1)}) \rightarrow T_j)$$

The stopping criterion is: $T_{(i)} = \emptyset$, or there is no T_j such that

$$v(F | (T_{(1)}, \dots, T_{(i-1)}) \rightarrow T_j) > v(F | T_{(1)}, \dots, T_{(i-1)}).$$

The output is the test-sequence $(T_{(1)}, \dots, T_{(i-1)})$.

The optimal test-schedule based on the output of *Greedy-Test before Sequence* is

$$\Phi_G = [\dots [\emptyset \rightarrow T_{(1)}] \rightarrow \dots] \rightarrow T_{(i-1)}.$$

In contrast to the sequence before test algorithm, the test before sequence algorithm sequences a subset out of the n available tests. Comparing the outputs of these two algorithms in terms of expected value is to compare $v(F | \Phi_G)$ to $v(F | \Phi_{Gr})$. To find general conditions in which one of the algorithms perform better than the other is difficult. One can think of problems where the *Greedy-Test before Sequence* sequences a few tests such that this subset always performs better than the sequence given by *Greedy-Sequence before Test*.

Recall that it is important here to come up with some algorithms that sequence the tests in shorter time than the $O(n!)$ it takes to try out all test-sequences. Both *Greedy-Sequence before Test* and *Greedy-Test before Sequence* are algorithms that take shorter than $O(n^2)$ time. Combining the two algorithms seems like a good idea: First sequence a test according to the *Greedy-Test before Sequence* algorithm. If a test is not returned i.e. $T_{(i)} = \emptyset$ then the test is sequenced by the *Greedy-Sequence before Test* algorithm.

Note that $T_{(i)} = \emptyset$ in the *Greedy-Test before Sequence* when there is no T_j , $j \in N \setminus \{(1), \dots, (i-1)\}$ such that

$$v(F | (T_{(1)}, \dots, T_{(i-1)}) \rightarrow T_j) > v(F | T_{(1)}, \dots, T_{(i-1)}).$$

Given the condition above the enhanced algorithm sequences the test according to the strategy in the *Greedy-Sequence before Test* instead of the *Greedy-Test before Sequence*.

Algorithm Greedy-Enhanced Test before Sequence:

$$\ddot{T}_{(1)} = \underset{j \in N}{\text{Argmax}} v(F | T_j)$$

$$\ddot{T}_{(2)} = \underset{j \in N \setminus \{(1)\}}{\text{Argmax}} v(F | \ddot{T}_{(1)} \rightarrow T_j)$$

$$\text{If } \ddot{T}_{(2)} = \emptyset \text{ then reassign } \ddot{T}_{(2)} := \underset{j \in N \setminus \{(1)\}}{\text{Argmax}} v(F | \ddot{T}_{(1)}, T_j)$$

If $v(F | [\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dot{T}_{(2)}) \geq v(F | [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \ddot{T}_{(2)})$ *then reassign*

$$\dot{T}_{(1)} := \dot{T}_{(1)}, \ddot{T}_{(2)} := \dot{T}_{(2)}$$

$\vdots$

$$\ddot{T}_{(n)} = \underset{j \in N \setminus \{(1), \dots, (n-1)\}}{\text{Argmax}} v(F | (\ddot{T}_{(1)}, \dots, \ddot{T}_{(n-1)}) \rightarrow T_j)$$

$$\text{If } \ddot{T}_{(n)} = \emptyset \text{ then reassign } \ddot{T}_{(n)} := \underset{i \in N \setminus \{(1), \dots, (n-1)\}}{\text{Argmax}} v(F | \ddot{T}_{(1)}, \dots, \ddot{T}_{(n-1)}, T_i)$$

If $v(F | [\dots [\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dots] \rightarrow \dot{T}_{(n)}) \geq v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)})$ *then reassign*

$$\dot{T}_{(1)} := \dot{T}_{(1)}, \ddot{T}_{(2)} := \dot{T}_{(2)}, \dots, \ddot{T}_{(n)} := \dot{T}_{(n)}$$

The output is the test-sequence $(\dot{T}_{(1)}, \ddot{T}_{(2)}, \dots, \ddot{T}_{(n)})$.

Based on this test-sequence the optimal test-schedule is

$$\Phi_{EG} = [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)}.$$

The time spent by *Greedy-Enhanced Test before Sequence* to compute the test sequence in worst case is important. The worst case computation at each step is dominated by the time it takes to compute $v(F | [\dots [\emptyset \rightarrow T_i] \rightarrow \dots] \rightarrow T_j)$. This takes less than $O(k \cdot 2^{i+1})$ comparisons (the maximisations), where k is the size of outcome space τ_i such that $k = \max_{i \in N} |\tau_i|$. Note that if all the tests has only two test outcomes then $k = 2$. Running the *Greedy-Enhanced Test before Sequence* with all its comparisons takes an additional $O(k \cdot 2^{n+2})$ time which bounds the problem asymptotically to $O(k \cdot 2^n)$.

The test sequence from *Greedy-Test before Sequence* can differ from the *Greedy-Enhanced Test before Sequence* when more than two tests are sequenced.

LEMMA 14 (Sequence before Test vs. Enhanced Test before Sequence)

The output of the Greedy-Sequence before Test and the output of the Greedy-Enhanced Test before Sequence algorithm are not necessarily equal from the second test onwards.

PROOF: To see that the two algorithms can provide different results assume that *Greedy-Sequence before Test* sequences a test $\dot{T}_{(2)}$ such that $\forall i_{(1)}$ we have that

$$v(F|i_{(1)}, \dot{T}_{(2)}) < v(F|i_{(1)}).$$

Assume that *Greedy-Enhanced Test before Sequence* sequences a test such that

$$\ddot{T}_{(2)} = \underset{j \in N \setminus \{(1)\}}{\text{Argmax}} v(F|\ddot{T}_{(1)} \rightarrow T_j).$$

This gives

$$\max_{j \in N \setminus \{(1)\}} v(F|\ddot{T}_{(1)} \rightarrow T_j) > v(F|\ddot{T}_{(1)})$$

for some $\ddot{T}_{(1)}$ and $j \in N \setminus \{(1)\}$. Applying Jensen's inequality to the sum over $\ddot{T}_{(1)}$ gives

$$\max_{j \in N \setminus \{(1)\}} v(F|\ddot{T}_{(1)} \rightarrow T_j) \geq \max_{j \in N \setminus \{(1)\}} \left\{ v(F|\ddot{T}_{(1)}) \right. \\ \left. v(F|\ddot{T}_{(1)}, T_j) \right\}.$$

Note that both algorithms are the same in the first test so that $\ddot{T}_{(1)} \equiv \dot{T}_{(1)}$. This gives

$$\max_{j \in N \setminus \{(1)\}} \left\{ v(F|\dot{T}_{(1)}) \right. \\ \left. v(F|\dot{T}_{(1)}, T_j) \right\} = \max_{j \in N \setminus \{(1)\}} \left\{ v(F|\dot{T}_{(1)}) \right. \\ \left. \max_{j \in N \setminus \{(1)\}} v(F|\dot{T}_{(1)}, T_j) \right\}.$$

Note that the lower term is sequencing the same test as the *Greedy-Sequence before Test*. Hence $T_j \equiv \dot{T}_{(2)}$ and $\forall i_{(1)}$ we have that

$$v(F|i_{(1)}, \dot{T}_{(2)}) < v(F|i_{(1)}).$$

This contradicts the assumption that $\ddot{T}_{(2)} = \underset{j \in N \setminus \{(1)\}}{\text{Argmax}} v(F|\ddot{T}_{(1)} \rightarrow T_j)$. Hence,

$$\ddot{T}_{(2)} \neq \dot{T}_{(2)}.$$

■

The condition

If $v(F|[\dots[\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dots] \rightarrow \dot{T}_{(i)}) \geq v(F|[\dots[\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(i)})$ then

$$\ddot{T}_{(1)} = \dot{T}_{(1)}, \ddot{T}_{(2)} = \dot{T}_{(2)}, \dots, \ddot{T}_{(i)} = \dot{T}_{(i)}$$

in the *Greedy-Enhanced Test before Sequence* algorithm ensures that the algorithm can not do worse than *Greedy-Sequence before Test*. One should however expect the

Greedy-Enhanced Test before Sequence to quickly perform better than *Greedy-Sequence before Test* since it first attempts to sequence test $\ddot{T}_{(i)}$ such that

$$\ddot{T}_{(i)} = \underset{j \in N \setminus \{1, \dots, (i-1)\}}{\text{Argmax}} \ v(F | (\ddot{T}_{(1)}, \dots, \ddot{T}_{(i-1)}) \rightarrow T_j),$$

which according to Theorem 9 satisfies

$$v(F | (\ddot{T}_{(1)}, \dots, \ddot{T}_{(i-1)}) \rightarrow T_j) \geq v(F | \ddot{T}_{(1)}, \dots, \ddot{T}_{(i-1)}, T_j).$$

When the optimal solution is obtained by Lemma 13 it will also be obtained for the same conditions for the *Greedy-Enhanced Test before Sequence* algorithm.

In search for a better sequencing algorithm, the strategy is improved by including and excluding tests. This is used in the *Greedy-Sequence by Inclusion & Exclusion* algorithm below. The first two steps of this algorithm chooses the same tests as the algorithms above, however the latter steps can be different as shown below.

Algorithm Greedy-Sequence by Inclusion & Exclusion:

$$\ddot{T}_{(1)} = \underset{j \in N}{\text{Argmax}} \ v(F | T_j).$$

$$\ddot{T}_{(2)} = \underset{j \in N \setminus \{1\}}{\text{Argmax}} \ v(F | [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow T_j)$$

$$\text{If } \ddot{T}_{(2)} = \emptyset \text{ then } \ddot{T}_{(2)} = \underset{j \in N \setminus \{1\}}{\text{Argmax}} \ v(F | \ddot{T}_{(1)}, T_j)$$

$$\text{If } v(F | [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \ddot{T}_{(2)}) \geq v(F | [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \ddot{T}_{(2)}) \text{ then}$$

$$\ddot{T}_{(1)} = \ddot{T}_{(1)}, \ddot{T}_{(2)} = \ddot{T}_{(2)}$$

⋮

$$\ddot{T}_{(n)} = \underset{j \in N \setminus \{1, \dots, (n-1)\}}{\text{Argmax}} \ v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n-1)}) \rightarrow T_j)$$

$$\text{If } \ddot{T}_{(n)} = \emptyset \text{ then } \ddot{T}_{(n)} = \underset{j \in N \setminus \{1, \dots, (n-1)\}}{\text{Argmax}} \ v(F | \ddot{T}_{(1)}, \dots, \ddot{T}_{(n-1)}, T_j)$$

$$\text{If } v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)}) \geq v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)}) \text{ then}$$

$$\ddot{T}_{(1)} = \ddot{T}_{(1)}, \ddot{T}_{(2)} = \ddot{T}_{(2)}, \dots, \ddot{T}_{(n)} = \ddot{T}_{(n)}$$

The output is the tests-sequence $(\ddot{T}_{(1)}, \dots, \ddot{T}_{(n)})$.

The optimal test-schedule based on the test-sequence is

$$\Phi_{1,E} = [\dots[\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)}.$$

The *Greedy-Enhanced Test before Sequence* is first 'inspecting' the test results of the selected tests $\ddot{T}_{(1)}, \dots, \ddot{T}_{(i-1)}$ before deciding whether to do the next test T_i . The *Greedy-Sequence by Inclusion & Exclusion* has a richer choice and can give a different test sequence than *Greedy-Enhanced Test before Sequence*. For each step the computation time is dominated by the time it takes to compute $v(F | [\dots[\emptyset \rightarrow T_i] \rightarrow \dots] \rightarrow T_i)$. So that the worst case computation time would also asymptotically be bounded by $O(k \cdot 2^n)$ which is the same as the worst case of *Greedy-Enhanced Test before Sequence*.

LEMMA 15 (Incl. & Exc. vs. Enhanced Test before Sequence algorithm)

The output of the Greedy-Sequence by Inclusion & Exclusion and the output of the Greedy-Enhanced Test before Sequence are not necessarily the same from the third test onwards.

PROOF: It is sufficient to consider a situation where $\psi = \{T_1, T_2, T_3, T_4\}$. Assume that $\ddot{T}_{(1)} = \ddot{T}_{(1)} = T_1$ and

$$\ddot{T}_{(2)} = \ddot{T}_{(2)} = T_2 = \underset{j \in N \setminus \{(1)\}}{\operatorname{Argmax}} v(F | [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow T_j).$$

Inspect the third step of *Greedy-Sequence by Inclusion & Exclusion*. This gives

$$\begin{aligned} v(F | ([\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \ddot{T}_{(2)}) \rightarrow \ddot{T}_{(3)}) &= \max_{j \in N \setminus \{(1), (2)\}} v(F | ([\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \ddot{T}_{(2)}) \rightarrow T_j) \\ &= \max_{j \in N \setminus \{(1), (2)\}} \begin{cases} v(F) \\ v(F | T_j) \\ v(F | \ddot{T}_{(2)} \rightarrow T_j) \\ v(F | [\ddot{T}_{(1)} \rightarrow \ddot{T}_{(2)}] \rightarrow T_j) \end{cases} \end{aligned}$$

We know that for all $k \in N$ that

$$\max \left\{ \begin{matrix} v(F) \\ v(F | T_k) \end{matrix} \right\} \leq \max_{j \in N \setminus \{(1), (2)\}} v(F | [\ddot{T}_{(1)} \rightarrow \ddot{T}_{(2)}] \rightarrow T_j)$$

which gives

$$= \max_{j \in N \setminus \{(1), (2)\}} \begin{cases} v(F | \ddot{T}_{(2)} \rightarrow T_j) \\ v(F | [\ddot{T}_{(1)} \rightarrow \ddot{T}_{(2)}] \rightarrow T_j) \end{cases}.$$

Assume that there is a T_3 and T_4 such that

$$\begin{aligned}
& \sum_{\ddot{i}_{(1)}} p(\ddot{i}_{(1)}) \max \left\{ v(F | \ddot{i}_{(1)}) \right. \\
& \quad \left. \sum_{\ddot{i}_{(2)}} p(\ddot{i}_{(2)} | \ddot{i}_{(1)}) \max \left\{ \begin{array}{l} v(F | \ddot{i}_{(1)}, \ddot{i}_{(2)}) \\ v(F | \ddot{i}_{(1)}, \ddot{i}_{(2)}, T_3) \end{array} \right. \right\} \\
& \leq \sum_{\ddot{i}_{(1)}} p(\ddot{i}_{(1)}) \max \left\{ v(F | \ddot{i}_{(1)}) \right. \\
& \quad \left. \sum_{\ddot{i}_{(2)}} p(\ddot{i}_{(2)} | \ddot{i}_{(1)}) \max \left\{ \begin{array}{l} v(F | \ddot{i}_{(1)}, \ddot{i}_{(2)}) \\ v(F | \ddot{i}_{(1)}, \ddot{i}_{(2)}, T_4) \end{array} \right. \right\} \\
& < \sum_{\ddot{i}_{(2)}} p(\ddot{i}_{(2)}) \max \left\{ \begin{array}{l} v(F | \ddot{i}_{(2)}) \\ v(F | \ddot{i}_{(2)}, T_3) \end{array} \right\}
\end{aligned}$$

This would cause *Greedy-Sequence by Inclusion & Exclusion* to have $\ddot{T}_{(3)} = T_3$ while *Greedy-Enhanced Test before Sequence* would in compliance with the central term sequence $\ddot{T}_{(3)} = T_4$ since

$$v(F | \ddot{T}_{(1)}, \ddot{T}_{(2)} \rightarrow T_4) > v(F | \ddot{T}_{(1)}, \ddot{T}_{(2)} \rightarrow T_3).$$

Hence the sequence of *Greedy-Sequence by Inclusion & Exclusion* is (T_1, T_2, T_3, T_4) while *Greedy-Enhanced Test before Sequence* gives (T_1, T_2, T_4, T_3) .

■

The three algorithms proposed above have properties as shown in Theorem 16, and when the conditions are the same as the ones in Lemma 13 all three algorithms give optimal solutions.

THEOREM 16

$$v(F | \Phi) \stackrel{I.E.}{\geq} v(F | \Phi) \stackrel{EG}{\geq} v(F | \Phi) \stackrel{Gr}{\geq}$$

with equality holding if the tests $T_1, \dots, T_n$ have the same conditions as in Theorem 15.

PROOF: First note that for $n = 1$ the three algorithms sequences the same test so that

$$\dot{T}_{(1)} = \ddot{T}_{(1)} = \ddot{T}_{(1)}.$$

Recall that one dot refers to the sequence in *Greedy-Sequence before Test*, two dots refers to *Greedy-Enhanced Test before Sequence* and three dots to the *Greedy-Sequence by Inclusion & Exclusion*. Induction is used to establish the inequality

$$v(F | \Phi) \stackrel{EG}{\geq} v(F | \Phi) \stackrel{Gr}{\geq}.$$

The first step is for $n = 2$ where *Greedy-Enhanced Test before Sequence* algorithm gives

$$v(F | [\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dot{T}_{(2)}) \leq v(F | [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \ddot{T}_{(2)}).$$

Let $n = j$ and assume that

$$v(F | [\dots [\emptyset \rightarrow \dot{T}_{(j)}] \rightarrow \dots] \rightarrow \dot{T}_{(j)}) \leq v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(j)}] \rightarrow \dots] \rightarrow \ddot{T}_{(j)})$$

Again the *Greedy-Enhanced Test before Sequence* algorithm sequences the $(j + 1)$ 'st test so that

$$v(F | [\dots [\emptyset \rightarrow \dot{T}_{(j)}] \rightarrow \dots] \rightarrow \dot{T}_{(j+1)}) \leq v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(j)}] \rightarrow \dots] \rightarrow \ddot{T}_{(j+1)})$$

Hence

$$v(F | [\dots [\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dots] \rightarrow \dot{T}_{(n)}) \leq v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)}).$$

To verify the second inequality $v(F | \dot{\Phi}) \geq v(F | \ddot{\Phi})$ is similar to proving $v(F | \ddot{\Phi}) \geq v(F | \overset{Gr}{\Phi})$ above.

The last to be proven is that equality holds under the conditions of Theorem 15. Lemma 13 gives

$$v(F | \overset{Gr}{\Phi}) = \max_{\pi \in Per(n)} v(F | [\dots [T_{\pi_1} \rightarrow T_{\pi_2}] \rightarrow \dots] \rightarrow T_{\pi_n}),$$

Hence $v(F | \overset{IE}{\Phi}) = v(F | \overset{Gr}{\Phi})$ so that equality holds.

■

Note that the reason why Theorem 16 holds is because of the comparison in the *If* statement performed by the *Greedy-Enhanced Test before Sequence* and the *Greedy-Sequence by Inclusion & Exclusion*. Since the *Greedy-Enhanced Test before Sequence* contains statements as

$$If \ v(F | [\dots [\emptyset \rightarrow \dot{T}_{(1)}] \rightarrow \dots] \rightarrow \dot{T}_{(n)}) \geq v(F | [\dots [\emptyset \rightarrow \ddot{T}_{(1)}] \rightarrow \dots] \rightarrow \ddot{T}_{(n)}) \text{ then}$$

$$\ddot{T}_{(1)} = \dot{T}_{(1)}, \ddot{T}_{(2)} = \dot{T}_{(2)}, \dots, \ddot{T}_{(n)} = \dot{T}_{(n)}$$

then it can not do worse than the Greedy-Test before Sequence algorithm. Similar comparisons are also taken by the Greedy-Sequence by Inclusion & Exclusion algorithm. Both of Greedy-Sequence by Inclusion & Exclusion and Greedy-Enhanced Test before Sequence can be executed less adequately without the *If* statements discussed above, however Theorem 16 will not hold. This can be seen by the same kind of argument that is used in the proof of Lemma 15. Recall that it was shown that the sequence of *Greedy-Sequence by Inclusion & Exclusion* is (T_1, T_2, T_3, T_4) while *Greedy-Enhanced Test before Sequence* gives (T_1, T_2, T_4, T_3) . If the *If* statement in the *Greedy-Sequence by Inclusion & Exclusion* algorithm is dropped then the algorithm can not prevent

$$v(F | [[[\emptyset \rightarrow T_1] \rightarrow T_2] \rightarrow T_3] \rightarrow T_4) \leq v(F | [[[\emptyset \rightarrow T_1] \rightarrow T_2] \rightarrow T_4] \rightarrow T_3).$$

Greedy - Counter Example

This example was used by Norström et al [2] to show that the Greedy algorithms are not optimal. The construction of this example is shown in detail in the Construction of Greedy Example on page 136. The example shows that the greedy algorithms are not

always optimal. Let $S = \{X_1, X_2 \mid 1 \geq X_i \geq 0, i \in (1, 2)\}$ and let the event B be defined by $B = 0$ if $x_1 < \frac{1}{2}$. Let us consider two acts f_1 and f_2 . Defining the consequences over $F \times S$ gives the following utilities:

$$c_1 : f_1, B = 0; U(c_1) = a + 2000$$

$$c_2 : f_1, B = 1; U(c_2) = a$$

$$c_3 : f_2, B = 0; U(c_3) = a + 1150$$

$$c_4 : f_2, B = 1; U(c_4) = a + 2150$$

where $a \geq 0$.

X_1 and X_2 are independent $U[0,1]^2$ variables. Let T_1 and T_2 be given by:

$$T_1 = \begin{cases} 0 & \text{if } x_2 < \varepsilon_1 x_1 + \frac{1}{2}(1 - \varepsilon_1) \\ 1 & \text{otherwise} \end{cases}; T_2 = \begin{cases} 0 & \text{if } x_2 < \varepsilon_2 x_1 + \frac{1}{2}(1 - \varepsilon_2) \\ 1 & \text{otherwise} \end{cases}$$

where $0 \leq \varepsilon_1 \leq \varepsilon_2 \leq 1$. The model is shown in Figure 25 with the test outcomes marked.

Figure 25 Test outcomes.

The lines $x_2 = \varepsilon_1 x_1 + \frac{1}{2}(1 - \varepsilon_1)$ and $x_2 = \varepsilon_2 x_1 + \frac{1}{2}(1 - \varepsilon_2)$ in Figure 25 show how the tests split the region S given by x_1 and x_2 . This gives the following probabilities:

$$p\{T_1 = 0\} = p\{T_2 = 0\} = \frac{1}{2}$$

$$p\{T_1 = 0, T_2 = 0\} = p\{T_1 = 1, T_2 = 1\} = \frac{4 - \varepsilon_2 + \varepsilon_1}{8}$$

$$p\{T_1 = 1, T_2 = 0\} = p\{T_1 = 0, T_2 = 1\} = \frac{\varepsilon_2 - \varepsilon_1}{8}$$

$$p\{B = 0 \mid T_1 = 0, T_2 = 0\} = p\{B = 1 \mid T_1 = 1, T_2 = 1\} = \frac{2 - \varepsilon_2}{4 - \varepsilon_2 + \varepsilon_1}$$

$$p\{B = 0 \mid T_1 = 1, T_2 = 0\} = p\{B = 1 \mid T_1 = 0, T_2 = 1\} = 0$$

$$p\{B = 0 \mid T_1 = 0, T_2 = 1\} = p\{B = 1 \mid T_1 = 1, T_2 = 0\} = 1$$

$$p\{B = 0 \mid T_1 = 1, T_2 = 1\} = p\{B = 1 \mid T_1 = 0, T_2 = 0\} = \frac{2 + \varepsilon_1}{4 - \varepsilon_2 + \varepsilon_1}$$

Assigning the values $\varepsilon_1 = 0.5$; $\varepsilon_2 = 1$; $\kappa_1 = 5$; $\kappa_2 = 48$ and calculating $v(F \mid [\emptyset \rightarrow T_1] \rightarrow T_2)$ and $v(F \mid [\emptyset \rightarrow T_2] \rightarrow T_1)$ with DPL (Ada Decision Systems 1992) give the expected values of information:

$$v(F \mid [\emptyset \rightarrow T_1] \rightarrow T_2) = v(F \mid T_1 \rightarrow T_2) = a + 1674.1$$

$$> v(F \mid [\emptyset \rightarrow T_2] \rightarrow T_1)$$

$$= v(F \mid T_2 \rightarrow T_1) = a + 1652.6 > v(F \mid T_2) = a + 1652$$

$$> v(F) = a + 1650 > v(F \mid T_1) = a + 1645.$$

Note that $v(F \mid T_1, T_2) = a + 1650.1$. It can be seen that this example serves as a counter example to both greedy algorithms since the greedy algorithms above will choose a sequence where T_2 is scheduled before T_1 . In Figure 27 and Figure 28 below the optimal test strategy is shown.

Figure 26: The optimal schedule of Figure 27.

By following the optimal 'bolded' lines Figure 27 the optimal schedule can be drawn as a control software flowchart as shown in Figure 26.

Construction of Greedy Example

Let $S = \{X_1, X_2 \mid 1 \geq X_i \geq 0, i \in (1, 2)\}$ and let the event B be defined as

$$B = \begin{cases} 0 & \text{if } x_1 < \frac{1}{2} \\ 1 & \text{otherwise} \end{cases}$$

Consider two acts f_1 and f_2 . The consequences are defined over $F \times S$, which gives the following consequences with the respective utilities:

$$c_1 : f_1, B = 0; U(c_1) = a + \xi$$

$$c_2 : f_1, B = 1; U(c_2) = a$$

$$c_3 : f_2, B = 0; U(c_3) = a + \Delta$$

$$c_4 : f_2, B = 1; U(c_4) = a + \Delta + \eta$$

where $a, \xi, \Delta, \eta \geq 0$.

X_1 and X_2 are independent with a $U[0,1]^2$ distribution. Let T_1 and T_2 be given by:

$$T_1 = \begin{cases} 0 & \text{if } x_2 < \varepsilon_1 x_1 + \frac{1}{2}(1 - \varepsilon_1) \\ 1 & \text{otherwise} \end{cases}; T_2 = \begin{cases} 0 & \text{if } x_2 < \varepsilon_2 x_1 + \frac{1}{2}(1 - \varepsilon_2) \\ 1 & \text{otherwise} \end{cases}$$

where $0 \leq \varepsilon_1 \leq \varepsilon_2 \leq 1$. This is shown in Figure 25 with the probabilities shown above.

Calculating $v(F)$:

$$v(f_1) = a + \xi \frac{1}{2}; v(f_2) = a + \Delta + \eta \frac{1}{2}$$

f_2 is required to be optimal which implies that $v(f_1) < v(f_2)$. This gives

$$\xi < 2\Delta + \eta \quad (90)$$

and the expected utility

$$v(F) = a + \Delta + \frac{\eta}{2}.$$

Calculating $v(F|T_1)$ and $v(F|\emptyset \rightarrow T_1)$:

$$v(f_1|T_1=0) = a + \xi \frac{2-\varepsilon_1}{4} - \kappa_1; v(f_2|T_1=0) = a + \Delta + \eta \frac{2+\varepsilon_1}{4} - \kappa_1$$

$$v(f_1|T_1=1) = a + \xi \frac{2+\varepsilon_1}{4} - \kappa_1; v(f_2|T_1=1) = a + \Delta + \eta \frac{2-\varepsilon_1}{4} - \kappa_1$$

f_2 is required optimal for all t_1 . This holds if $v(f_2|T_1=0) > v(f_1|T_1=0)$ and $v(f_2|T_1=1) > v(f_1|T_1=1)$. This gives the condition

$$\Delta > \xi \frac{2+\varepsilon_1}{4} - \eta \frac{2-\varepsilon_1}{4} \quad (91)$$

and the expected utility

$$v(F|T_1) = a + \Delta + \frac{\eta}{2} - \kappa_1.$$

Condition (91) implies that $v(F|T_1) = v(F) - \kappa_1$. This means no increase in the expected utility by doing T_1 .

It is required that

$$v(F|T_1) = a + \Delta + \frac{\eta}{2} - \kappa_1 < v(F) = a + \Delta + \frac{\eta}{2}$$

which gives the condition

$$\kappa_1 > 0 \quad (92)$$

and the expected utility

$$v(F|\emptyset \rightarrow T_1) = a + \Delta + \frac{\eta}{2}.$$

The consequence is that only doing T_1 can not increase the expected utility even if it test T_1 has no cost.

Calculating $v(F|T_2)$ and $v(F|\emptyset \rightarrow T_2)$:

$$v(f_1|T_2=0) = a + \xi \frac{2-\varepsilon_2}{4} - \kappa_2; \quad v(f_2|T_2=0) = a + \Delta + \eta \frac{2+\varepsilon_2}{4} - \kappa_2$$

$$v(f_1|T_2=1) = a + \xi \frac{2+\varepsilon_2}{4} - \kappa_2; \quad v(f_2|T_2=1) = a + \Delta + \eta \frac{2-\varepsilon_2}{4} - \kappa_2$$

Here f_2 is required optimal for $T_2 = 0$ and f_1 is required optimal for $T_2 = 1$. This holds if $v(f_2|T_2=0) > v(f_1|T_2=0)$ and $v(f_2|T_2=1) < v(f_1|T_2=1)$. This gives the condition

$$\xi \frac{2-\varepsilon_2}{4} - \eta \frac{2+\varepsilon_2}{4} < \Delta < \xi \frac{2+\varepsilon_2}{4} - \eta \frac{2-\varepsilon_2}{4} \quad (93)$$

and the expected utility

$$v(F|T_2) = a + \frac{\Delta}{2} + \eta \frac{2+\varepsilon_2}{8} + \xi \frac{2+\varepsilon_2}{8} - \kappa_2.$$

Condition (93) implies that $v(F|T_2) \geq v(F) - \kappa_2$. This means that an increase in the expected utility can be expected by doing T_2 if the cost κ_2 is sufficiently small.

It is required that

$$v(F|T_2) = a + \frac{\Delta}{2} + \eta \frac{2+\varepsilon_2}{8} + \xi \frac{2+\varepsilon_2}{8} - \kappa_2 > v(F) = a + \Delta + \frac{\eta}{2}.$$

This gives the condition

$$\Delta < \xi \frac{2+\varepsilon_2}{4} - \eta \frac{2-\varepsilon_2}{4} - 2\kappa_2 \quad (94)$$

and the expected utility

$$v(F|\emptyset \rightarrow T_2) = a + \frac{\Delta}{2} + \eta \frac{2+\varepsilon_2}{8} + \xi \frac{2+\varepsilon_2}{8} - \kappa_2$$

This means that T_2 is chosen if (93) and (94) is satisfied which gives

$$\xi \frac{2-\varepsilon_2}{4} - \eta \frac{2+\varepsilon_2}{4} < \Delta < \xi \frac{2+\varepsilon_2}{4} - \eta \frac{2-\varepsilon_2}{4} - 2\kappa_2 \quad (95)$$

Calculating $v(F|T_1, T_2)$, $v(F|T_1 = t_1, T_2)$ and $v(F|T_1, T_2 = t_2)$:

The following relations are required:

$$v(f_1|T_1=0, T_2=0) = a + \xi \frac{2-\varepsilon_2}{4-\varepsilon_2+\varepsilon_1} - \kappa_1 - \kappa_2 <$$

$$v(f_2|T_1=0, T_2=0) = a + \Delta + \eta \frac{2+\varepsilon_1}{4-\varepsilon_2+\varepsilon_1} - \kappa_1 - \kappa_2$$

$$v(f_1|T_1=1, T_2=0) = a - \kappa_1 - \kappa_2 < v(f_2|T_1=1, T_2=0) = a + \Delta + \eta - \kappa_1 - \kappa_2$$

$$v(f_1|T_1=0, T_2=1) = a + \xi - \kappa_1 - \kappa_2 > v(f_2|T_1=0, T_2=1) = a + \Delta - \kappa_1 - \kappa_2$$

$$v(f_1|T_1=1, T_2=1) = a + \xi \frac{2+\varepsilon_1}{4-\varepsilon_2+\varepsilon_1} - \kappa_1 - \kappa_2 <$$

$$v(f_2|T_1=1, T_2=1) = a + \Delta + \eta \frac{2-\varepsilon_2}{4-\varepsilon_2+\varepsilon_1} - \kappa_1 - \kappa_2$$

These inequalities give the condition:

$$\xi > \Delta > \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1}{4-\varepsilon_2+\varepsilon_1} \quad (96)$$

and the expected utilities

$$v(F|T_1, T_2) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_1 - \kappa_2 \quad (97)$$

$$v(F|T_1 = 0, T_2) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{4} \right) + \eta \frac{2 + \varepsilon_1}{4} + \xi \frac{\varepsilon_2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2 \quad (98)$$

$$v(F|T_1 = 1, T_2) = a + \Delta + \eta \frac{2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2 \quad (99)$$

$$v(F|T_1, T_2 = 0) = a + \Delta + \eta \frac{2 + \varepsilon_2}{4} - \kappa_1 - \kappa_2 \quad (100)$$

$$v(F|T_1, T_2 = 1) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{4} \right) + \eta \frac{2 - \varepsilon_2}{4} + \xi \frac{\varepsilon_2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2. \quad (101)$$

Condition (96) implies that $v(F|T_1, T_2) \geq v(F) - \kappa_1 - \kappa_2$. This means that there is an increase in the expected utility by doing T_1 and T_2 if the cost $\kappa_1 + \kappa_2$ is sufficiently small.

Calculating $v(F|T_1 \rightarrow T_2)$:

The required relations for $v(F|T_1 \rightarrow T_2)$ are given by:

$$v(F|T_1 = 0) = a + \Delta + \eta \frac{2 + \varepsilon_1}{4} - \kappa_1 <$$

$$v(F|T_1 = 0, T_2) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{4} \right) + \eta \frac{2 + \varepsilon_1}{4} + \xi \frac{\varepsilon_2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2$$

and

$$v(F|T_1 = 1) = a + \Delta + \eta \frac{2 - \varepsilon_1}{4} - \kappa_1 > v(F|T_1 = 1, T_2) = a + \Delta + \eta \frac{2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2$$

These inequalities give the condition

$$\Delta < \xi - \frac{4\kappa_2}{\varepsilon_2 - \varepsilon_1}; \kappa_2 > 0 \quad (102)$$

and the expected utility

$$v(F|T_1 \rightarrow T_2) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_1 - \frac{\kappa_2}{2}.$$

Condition (102) implies that $v(F|T_1 \rightarrow T_2) \geq v(F|T_1)$. Hence, there is an increase in the expected utility by doing T_1 and then having the choice to do T_2 .

Calculating $v(F|[\emptyset \rightarrow T_1] \rightarrow T_2)$:

The required relation for $v(F|[\emptyset \rightarrow T_1] \rightarrow T_2)$ is given by:

$$v(F|T_1 \rightarrow T_2) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_1 - \frac{\kappa_2}{2} >$$

$$v(F|\emptyset \rightarrow T_2) = a + \frac{\Delta}{2} + \eta \frac{2 + \varepsilon_2}{8} + \xi \frac{2 + \varepsilon_2}{8} - \kappa_2$$

This inequality gives the condition

$$\Delta > \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1}{4 - \varepsilon_2 + \varepsilon_1} - \frac{4\kappa_2 - 8\kappa_1}{4 - \varepsilon_2 + \varepsilon_1} \quad (103)$$

and the expected utility

$$v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_1 - \frac{\kappa_2}{2}. \quad (104)$$

The conditions above ensure that it is optimal to always do test T_1 , and given test T_1 it is optimal to do T_2 when $T_1 = 0$.

Calculating $v(F|T_2 \rightarrow T_1)$ and $v(F|[\emptyset \rightarrow T_2] \rightarrow T_1)$, Option - 1:

Permuting the test order so that T_2 is ordered before T_1 ; conditions that make it preferable to do T_1 given T_2 are searched so that T_2 always is chosen; T_1 is chosen for some t_2 ; the expected utility by ordering the two tests T_1, T_2 is higher than T_2, T_1 .

The relations for $v(F|T_2 \rightarrow T_1)$ here must conform with the conditions above and are given by:

$$v(F|T_2 = 0) = a + \Delta + \eta \frac{2 + \varepsilon_2}{4} - \kappa_2 >$$

$$v(F|T_1, T_2 = 0) = a + \Delta + \eta \frac{2 + \varepsilon_2}{4} - \kappa_1 - \kappa_2$$

and

$$v(F|T_2 = 1) = a + \xi \frac{2 + \varepsilon_2}{4} - \kappa_2 <$$

$$v(F|T_1, T_2 = 1) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{4} \right) + \eta \frac{2 - \varepsilon_2}{4} + \xi \frac{\varepsilon_2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2$$

These inequalities lead to the following condition

$$\Delta > \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1}{4 - \varepsilon_2 + \varepsilon_1} + \frac{4\kappa_1}{4 - \varepsilon_2 + \varepsilon_1}; \kappa_1 > 0 \quad (105)$$

and the expected utility

$$v(F|T_2 \rightarrow T_1) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_2 - \frac{\kappa_1}{2}.$$

Note that for $\kappa_1 = 0$, (105) conforms with (96). Condition (105) implies that $v(F|T_2 \rightarrow T_1) \geq v(F|T_2)$. If the cost κ_1 is sufficiently small an increase in the expected utility can be expected by doing T_2 and then having the choice to do T_1 .

The required relation for $v(F|[\emptyset \rightarrow T_2] \rightarrow T_1)$ is given by:

$$v(F|T_2 \rightarrow T_1) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_2 - \frac{\kappa_1}{2} >$$

$$v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) = a + \Delta + \frac{\eta}{2}$$

This inequality gives the condition

$$\Delta < \xi - \frac{8\kappa_2 + 4\kappa_1}{\varepsilon_2 - \varepsilon_1} \quad (106)$$

and the expected utility

$$v(F|[\emptyset \rightarrow T_2] \rightarrow T_1) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{8} \right) + \frac{\eta}{2} + \xi \frac{\varepsilon_2 - \varepsilon_1}{8} - \kappa_2 - \frac{\kappa_1}{2}$$

Finally it is required that

$$v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1)$$

which gives the last condition

$$\kappa_1 < \kappa_2 \quad (107)$$

This condition is required to prevent the greedy algorithms to give an optimal solution of the test-sequence.

Summary Conditions, Option - 1 and The Counter Example of the Greedy

To perform T_2 given T_1 it is sufficient to consider the conditions (90), (91), (92), (95), (96), (102) and (103) that ensures that T_1 does not provide any information alone. However, while in combination with T_2 , T_1 should always be executed. In addition it is required that $v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1)$ for the case that T_1 is performed given T_2 . This gives the additional conditions (105), (106) and (107). Simplifying the conditions above to the conditions in (108).

$$\frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1 + 4\kappa_1}{4 - \varepsilon_2 + \varepsilon_1} < \Delta < \min \left\{ \xi - \frac{8\kappa_2 + 4\kappa_1}{\varepsilon_2 - \varepsilon_1}, \xi \frac{2 + \varepsilon_2}{4} - \eta \frac{2 - \varepsilon_2}{4} - 2\kappa_2 \right\}; \quad (108)$$

$$0 < \kappa_1 < \kappa_2; 0 \leq \varepsilon_1 \leq \varepsilon_2 \leq 1$$

The conditions in (108) give $v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) = v(F|T_1 \rightarrow T_2) > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1) = v(F|T_2 \rightarrow T_1) > v(F|T_2) > v(F) > v(F|T_1)$.

Some numbers that satisfy (108) are

$$\varepsilon_1 = 0.5; \varepsilon_2 = 1; \xi = 2000; \eta = 1000; \Delta = 1150; \kappa_1 = 5; \kappa_2 = 48$$

which gives the expected utilities

$$\begin{aligned}
v(F | [\emptyset \rightarrow T_1] \rightarrow T_2) &= v(F | T_1 \rightarrow T_2) = a + 1674.1 > v(F | [\emptyset \rightarrow T_2] \rightarrow T_1) = \\
v(F | T_2 \rightarrow T_1) &= a + 1652.6 > v(F | T_2) = a + 1652 > v(F) = a + 1650 > \\
v(F | T_1) &= a + 1645
\end{aligned}$$

note that $v(F | T_1, T_2) = a + 1650.1$. It can be seen that this example serves as a counter example to the greedy algorithm since the greedy algorithm (both *Test before Schedule* and *Schedule before Testing*) will select the order so T_2 is done before T_1 . In Figure 27 and Figure 28 below the optimal test strategy is shown. The optimal decisions are the ones with bold lines.

Figure 27 $v(F | [\emptyset \rightarrow T_1] \rightarrow T_2)$.

Figure 28 $v(F | [\emptyset \rightarrow T_2] \rightarrow T_1)$.

Calculating $v(F | T_2 \rightarrow T_1)$ and $v(F | [\emptyset \rightarrow T_2] \rightarrow T_1)$, Option - 2:

Permuting the test order so that T_2 is ordered before T_1 ; conditions that make it preferable to do T_2 are searched so that T_2 always is chosen; T_1 is not chosen when sequenced after T_2 ; the expected utility by ordering the two tests T_1, T_2 is higher than T_2, T_1 .

The relations for $v(F | T_2 \rightarrow T_1)$ must conform with the conditions above and are given by:

$$v(F | T_2 = 0) = a + \Delta + \eta \frac{2 + \varepsilon_2}{4} - \kappa_2 >$$

$$v(F | T_1, T_2 = 0) = a + \Delta + \eta \frac{2 + \varepsilon_2}{4} - \kappa_1 - \kappa_2$$

and

$$v(F | T_2 = 1) = a + \xi \frac{2 + \varepsilon_2}{4} - \kappa_2 >$$

$$v(F | T_1, T_2 = 1) = a + \Delta \left(1 - \frac{\varepsilon_2 - \varepsilon_1}{4} \right) + \eta \frac{2 - \varepsilon_2}{4} + \xi \frac{\varepsilon_2 - \varepsilon_1}{4} - \kappa_1 - \kappa_2$$

These inequalities lead to the following condition

$$\Delta < \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1}{4 - \varepsilon_2 + \varepsilon_1} + \frac{4\kappa_1}{4 - \varepsilon_2 + \varepsilon_1}; \kappa_1 > 0 \quad (109)$$

and the expected utility

$$v(F|T_2 \rightarrow T_1) = a + \frac{\Delta}{2} + \eta \frac{2+\varepsilon_2}{8} + \xi \frac{2+\varepsilon_2}{8} - \kappa_2.$$

Condition (109) implies that $v(F|T_2 \rightarrow T_1) = v(F|T_2)$. This means that there is no increase in the expected utility when doing T_2 and having the choice to do T_1 if the cost κ_1 is sufficiently big.

The required relation for $v(F|[\emptyset \rightarrow T_2] \rightarrow T_1)$ is given by:

$$v(F|T_2 \rightarrow T_1) = a + \frac{\Delta}{2} + \eta \frac{2+\varepsilon_2}{8} + \xi \frac{2+\varepsilon_2}{8} - \kappa_2 > v(F|\emptyset \rightarrow T_1) = a + \Delta + \frac{\eta}{2}$$

This inequality gives the condition

$$\Delta < \xi \frac{2+\varepsilon_2}{4} - \eta \frac{2-\varepsilon_2}{4} - 2\kappa_2 \quad (110)$$

It conforms with (95), and the expected utility

$$v(F|[\emptyset \rightarrow T_2] \rightarrow T_1) = a + \frac{\Delta}{2} + \eta \frac{2+\varepsilon_2}{8} + \xi \frac{2+\varepsilon_2}{8} - \kappa_2$$

Condition (110) means that only T_2 will be chosen when sequencing T_2 before T_1 .

Finally it is required that

$$v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1)$$

which gives the last condition

$$\Delta > \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1}{4-\varepsilon_2 + \varepsilon_1} - \frac{4\kappa_2 - 8\kappa_1}{4-\varepsilon_2 + \varepsilon_1}. \quad (111)$$

Hence, the greedy algorithm can not give the optimal solution. Condition (111) conforms to condition (103).

Summary Conditions, Option - 2 and The Counter Example of the Greedy

To perform T_2 given T_1 it is sufficient to consider the conditions (90), (91), (92), (95), (96), (102) and (103) that ensures that T_1 does not provide any information alone. However, while in combination with T_2 , T_1 should always be executed. In addition it is required that $v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1) = v(F|T_2)$ for the case that T_1 is performed given T_2 . This gives the additional conditions (109), (110) and (111). Simplifying the conditions above give (112).

$$\max \left\{ \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1}{4 - \varepsilon_2 + \varepsilon_1}, \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1 + 8\kappa_1 - 4\kappa_2}{4 - \varepsilon_2 + \varepsilon_1} \right\} < \Delta < \min \left\{ \xi - \frac{4\kappa_2}{\varepsilon_2 - \varepsilon_1}, \xi \frac{2 + \varepsilon_2}{4} - \eta \frac{2 - \varepsilon_2}{4} - 2\kappa_2, \frac{-2\eta + \varepsilon_2\eta + 2\xi + \xi\varepsilon_1 + 4\kappa_1}{4 - \varepsilon_2 + \varepsilon_1} \right\}; \quad (112)$$

$$0 \leq \varepsilon_1 \leq \varepsilon_2 \leq 1$$

It is shown above that the inequalities in (112) are sufficient to give $v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) = v(F|T_1 \rightarrow T_2) > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1) = v(F|T_2 \rightarrow T_1) = v(F|T_2) > v(F) > v(F|T_1)$.

Some numbers that satisfy (112) are

$$\varepsilon_1 = 0.5; \varepsilon_2 = 1; \xi = 2000; \eta = 1000; \Delta = 1150; \kappa_1 = 20; \kappa_2 = 48.$$

They give the expected utilities

$$\begin{aligned} v(F|[\emptyset \rightarrow T_1] \rightarrow T_2) &= v(F|T_1 \rightarrow T_2) = a + 1659.1 > v(F|[\emptyset \rightarrow T_2] \rightarrow T_1) = \\ v(F|T_2 \rightarrow T_1) &= v(F|T_2) = a + 1652 > v(F) = a + 1650 > v(F|T_1) = a + 1630 \end{aligned}$$

note that $v(F|T_1, T_2) = a + 1635.1$. In Figure 29 and Figure 30 the decision trees with the figures above are shown. The optimal test strategy is marked with bold lines.

Figure 29 $v(F | [\emptyset \rightarrow T_1] \rightarrow T_2)$.

Figure 30 $v(F | [\emptyset \rightarrow T_2] \rightarrow T_1)$.

It can be seen that this example serves as a counter example to the greedy algorithm since the greedy algorithm (both *Test before Schedule* and *Schedule before Testing*) will select a test-sequence so T_2 is done before T_1 .

Note that the example shows a case where it is not optimal to always do the two tests. However having the choice to do the next test given the results of the previous test gives a higher expected utility.

Notation

The need to understand the test-schedules of $v(F | \emptyset \rightarrow [T_1 \rightarrow [\dots \rightarrow T_n] \dots])$ and $v(F | [\dots [\emptyset \rightarrow T_1] \rightarrow \dots] \rightarrow T_n)$ is important. Below they are expanded to show their context.

$$\begin{aligned}
 v(F | \emptyset \rightarrow [T_1 \rightarrow [\dots \rightarrow T_n] \dots]) &= \max \begin{cases} v(F) \\ v(F | T_1 \rightarrow [T_2 \rightarrow [\dots \rightarrow T_n] \dots]) \end{cases} \\
 &= \max \begin{cases} v(F) \\ \sum_{t_1} p(t_1) \max \begin{cases} v(F | t_1), \\ \sum_{t_k \in T_k} p(t_k | t_1, \dots, t_{k-1}) \max \begin{cases} v(F | t_1, \dots, t_k) \\ v(F | t_1, \dots, t_k, T_{k+1} \rightarrow [T_{k+2} \rightarrow [\dots \rightarrow T_n] \dots]) \end{cases} \end{cases} \end{cases}
 \end{aligned}$$

Expanding $v(F | [\dots [\emptyset \rightarrow T_1] \rightarrow \dots] \rightarrow T_n)$ gives:

$$v(F | [\dots [\emptyset \rightarrow T_1] \dots] \rightarrow T_n) = \max \begin{cases} v(F | [\dots [\emptyset \rightarrow T_2] \dots] \rightarrow T_n) \\ v(F | [\dots [T_1 \rightarrow T_2] \dots] \rightarrow T_n) \end{cases}$$

$$\begin{aligned}
&= \max \left\{ \begin{array}{l} \max \left\{ \begin{array}{l} v(F | [\dots [\emptyset \rightarrow T_3] \rightarrow \dots] \rightarrow T_n) \\ v(F | [\dots [T_2 \rightarrow T_3] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \\ \sum_{t_1} p(t_1) \max \left\{ \begin{array}{l} v(F | t_1; [\dots [\emptyset \rightarrow T_3] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_1; [\dots [T_2 \rightarrow T_3] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \end{array} \right. \\
&= \max \left\{ \begin{array}{l} \max \left\{ \begin{array}{l} v(F | [\dots [\emptyset \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \\ v(F | [\dots [T_3 \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \\ \sum_{t_2} p(t_2) \max \left\{ \begin{array}{l} v(F | t_2; [\dots [\emptyset \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_2; [\dots [T_3 \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \\ \sum_{t_1} p(t_1) \max \left\{ \begin{array}{l} \max \left\{ \begin{array}{l} v(F | t_1; [\dots [\emptyset \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_1; [\dots [T_3 \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \\ \sum_{t_2} p(t_2 | t_1) \max \left\{ \begin{array}{l} v(F | t_1, t_2; [\dots [\emptyset \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_1, t_2; [\dots [T_3 \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \end{array} \right. \end{array} \right. \\
&= \max \left\{ \begin{array}{l} \max \left\{ \begin{array}{l} v(F | [\dots [\emptyset \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \\ v(F | [\dots [T_3 \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \\ \sum_{t_2} p(t_2) \max \left\{ \begin{array}{l} v(F | t_2; [\dots [\emptyset \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_2; [\dots [T_3 \rightarrow T_4] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \\ \dots \max \left\{ \begin{array}{l} v(F | t_1, \dots, t_{k-1}; [\dots [\emptyset \rightarrow T_{k+1}] \dots] \rightarrow T_n) \\ \sum_{t_k} p(t_k | t_1, \dots, t_{k-1}) \max \left\{ \begin{array}{l} v(F | t_1, \dots, t_k; [\dots [\emptyset \rightarrow T_{k+2}] \rightarrow \dots] \rightarrow T_n) \\ v(F | t_1, \dots, t_k; [\dots [T_{k+1} \rightarrow T_{k+2}] \rightarrow \dots] \rightarrow T_n) \end{array} \right. \end{array} \right. \end{array} \right.
\end{aligned}$$

4.4 References

- [1] L. Savage: "The Foundations of Statistics". Dover, New York, 1972.
- [2] J.G. Norstrøm, R.M. Cooke & T. Bedford: "Statistical methods in design of safety critical software". In Lydersen, Hansen & Sandtorv (eds), "Safety and Reliability", Rotterdam, Balkema, 1998, pp. 1061-1068.
- [3] J. G. Norstrøm, R. M. Cooke, T. Bedford: "Value of Information based inspection strategy of a fault-tree". In Kafka & Schüeller (eds), Safety and Reliability, 621-626. Rotterdam: Balkema, 1999.
- [4] Ada Decision Systems: "DPL Advanced Version User Guide", 2710 Sand Hill Road, Menlo Park, CA94025, USA, 1992.
- [5] R. M. Dudley: "Real Analysis and Probability", Chapman & Hall, 1989.
- [6] M. H. De Groot: "Optimal Statistical Decisions", McGraw-Hill, New York, 1970.
- [7] De Finetti: "Theory of Probability", Wiley & Sons, 1990 (Note first published 1975), pp. 217-218.
- [8] R. M. Cooke: "Experts in Uncertainty", Oxford University Press, New York, 1991.
- [9] S. French, 'Decision Theory: An introduction to the Mathematics of Rationality', Ellis Horwood Limited, 1993.

Chapter 5

Optimal Inspection of Fault-Trees

Control software is software that receives observations from sensors built into a system to timely activate actuators to trigger required functions. How the software monitors its sensors determines the expected number of inspections before one can make a decision. This problem is similar to minimal cut-set search in a fault-tree. Previous approaches in Balkey [3] and Xiaozhong [5] are based on Fussel-Vesely importance measures. We will now show how we can do better than an inspection strategy purely based on the Fussel-Vesely importance.

Norstrøm et al [1] developed statistical methods for the design of safety critical software that uses value of information. It has been hard to validate the approach since the mathematics involved is rather complex. During the PSAM 4 conference located in New York in 1998, Dr. Tim Bedford got the idea of using the methods in Norstrøm et al [1] to optimally search out the minimal cut-set that has failed in a fault-tree given the top event. I researched his ideas and the results were presented at the ESREL'99 conference in Munich (see Norstrøm et al [2]).

5.1 Introduction

We will first illustrate the problem of finding the minimal cut set with a small example. Note that we will not go into detail in the calculations used. Such details are saved for later. We use $X_i = 1$ to denote the failure of Component i and $X_i = 0$ to denote that it is not failed and so on. Let us consider the fault tree in Figure 31 with the minimal cut-sets $s_1 = \{1, 2, 4\}$, $s_2 = \{1, 2, 5\}$, $s_3 = \{1, 3, 6\}$ and $s_4 = \{3, 4, 5\}$. The probabilities of failure are shown in Figure 31.

Figure 31. Fault-tree

Assuming that the events in the fault-tree are independent, the calculation of the probability of failure for each minimal cut-set gives

$$p(s_1) = p(X_1 = 1 \cap X_2 = 1 \cap X_4 = 1) = 5.00 \cdot 10^{-11}$$

$$p(s_2) = p(X_1 = 1 \cap X_2 = 1 \cap X_5 = 1) = 1.00 \cdot 10^{-8}$$

$$p(s_3) = p(X_1 = 1 \cap X_3 = 1 \cap X_6 = 1) = 2.00 \cdot 10^{-7}$$

$$p(s_4) = p(X_3 = 1 \cap X_4 = 1 \cap X_5 = 1) = 1.00 \cdot 10^{-8}.$$

Since each basic event has a small probability of failure we can use the rare-event approximation of the top event which gives

$$p(\text{top}) = \sum_i p(s_i) = 2.201 \cdot 10^{-7}.$$

The rare event approximation makes the assumption that minimal cut-sets are exclusive events, that is that precisely one minimal cut-set fails.

One inspection strategy is to inspect the Components (6, 5, 4) as shown in Figure 32. In this strategy we stop inspections when we are almost certain about which cut-set that has failed. For example the probability that cut s_3 has failed given the top event and the failure of Component 6 is $p(s_3 | X_6 = 1, \text{top}) = .999499 \approx 1$.

Figure 32. Inspection strategy.

Figure 32 is similar to an event tree. The number in a circular node shows the component that is being inspected. An inspection result can either be that the component is *failed* or *not failed* as indicated on the two branches attached to the node. For each branch it is also indicated the conditional probability of the state of the component. For example the *not fail* branch of Component 5 has the conditional probability

$$p(X_5 = 0 | X_6 = 0, \text{top}) = .00246889.$$

This is the probability that Component 5 is *not failed* given the top event and that Component 6 is *not failed*. The *not fail* branch of Component 5 is not followed by another circular node which means that inspection has stopped. We have then given the probability of the most likely cut set causing the top event. For example the *not fail* branch of Component 5 gives the conditional probability

$$p(s_1 | X_5 = 0, X_6 = 0, \text{top}) = 1.$$

This is the conditional probability of minimal cut-set s_1 given that Components 5 and 6 are not failed and the top event. The other branch of Component 5 denoted *fail* leads to the node 4 that shows that Component 4 will be inspected. After inspecting Component 4 we become almost certain which minimal cut-set is failed and we stop testing after a maximum of 3 inspections.

Another inspection strategy is (5, 2, 6) as shown in Figure 33.

Figure 33 Inspection strategy.

The expected number of steps required to find the minimal cut-set that has failed for Figure 32 and Figure 33 are

$$E(N) = p(X_6 = 1 | \text{top}) \cdot 1 + p(X_6 = 0 | \text{top}) \cdot p(X_5 = 0 | X_6 = 0, \text{top}) \cdot 2 + p(X_6 = 0 | \text{top}) \cdot p(X_5 = 1 | X_6 = 0, \text{top}) \cdot 3 = 1.185,$$

and

$$E(N) = p(X_5 = 1 | \text{top}) \cdot 2 + p(X_5 = 0 | \text{top}) \cdot 2 = 2.000$$

respectively. From this small example several questions arise.

1. How should one choose the components to inspect in order to find the failed minimal cut-set, and in which order should one inspect them?
2. Does the component importance such as Fussel-Vesely give any good indications to answer question (1)?
3. How should one compute the probabilities that are needed to compute the expectations in Figure 32 and Figure 33? Are there good approximations and when do the approximations work?
4. When should one stop inspecting? In the example above inspections were stopped when one had become almost certain of being in a particular minimal cut-set.

5.2 Useful Approximations, Probabilities and Error-Bounds

Let the set of cut-sets be denoted by S and let the set of components be $K = \{1, 2, \dots, n\}$. Let the component indicators be the random variables $\psi = \{X_1, X_2, \dots, X_n\}$, where the index identifies the component. Each indicator X_i , $i \in K$ has two possible outcomes

$$X_i = \begin{cases} 1 & \text{if component } i \text{ is failed,} \\ 0 & \text{otherwise.} \end{cases}$$

We assume that an inspection always correctly identifies the state of the component. The realisation x_i of X_i describes the states that Component i can have so that if $x_i = 1$ Component i is failed and if $x_i = 0$ then Component i is not failed. The probability of failure of Component i is p_i , and the components fail mutually independently of each other.

The state of the system can be described by the structure function

$$h(x_1, \dots, x_n) = \begin{cases} 1 & \text{if the top event is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Høyland & Rausand [7] describes how the structure function can be obtained and it is shown how the probability of the top-event can be obtained from the structure function in standard form so that

$$p(\text{top}) = h(p_1, \dots, p_n). \quad (113)$$

According to Høyland & Rausand [7] one can use the rare event approximation,

$$p(\text{top}) \approx \sum_i p(s_i),$$

if the basic-events in the fault-tree have probabilities less than about 10^{-2} . Let the error by using the rare-event approximation be defined by

$$p(\text{top}) = \sum_i p(s_i) - \varepsilon, \quad \varepsilon \geq 0. \quad (114)$$

Since the basic-events are independent the probability that cut-set s_i is failed is

$$p(s_i) = \prod_{k \in s_i} p(X_k = 1) = \prod_{k \in s_i} p_k$$

Some useful quantities that we need when we want to calculate the expected number of inspections are

- i) The conditional probability of cut-set s_i given top, $p(s_i | \text{top})$, and the conditional probability of cut-set s_i given the top-event and the state of Components $1, \dots, j$ which is written $p(s_i | \text{top}, x_1, \dots, x_j)$.
- ii) The conditional probability of the state of Component k given the top event, $p(x_k | \text{top})$.
- iii) For any $j, k_1, \dots, k_i \in K$ such that $j \neq k_l$ then $p(x_j | \text{top}, x_{k_1}, \dots, x_{k_i})$ is the conditional probability of the state of Component j given the top and the states of Components $k_1, \dots, k_i$.

In the following we will show how one can obtain these probabilities. Before we answer i) to iii) above let us consider some results that we will need. A useful expression is

$$p(\text{top} \cap (X_k = 1)) = p\left(\bigcup_{i \in S} ((X_k = 1) \cap s_i)\right) = \sum_{i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k), \quad (115)$$

where (115) defines $\varepsilon'(k)$. To find upper bounds for ε and $\varepsilon'(k)$ we will use the inclusion exclusion formula to calculate the probability of the union of events.

LEMMA 16

Let $k \in K$, $s_i \in S$. Upper bounds for ε and ε' are

$$\varepsilon \leq \sum_{i < j} p(s_i \cap s_j) \text{ and}$$

$$\varepsilon'(k) \leq \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j).$$

PROOF: The inclusion exclusion formula is

$$\begin{aligned} p\left(\bigcup_{i \in S} ((X_k = 1) \cap s_i)\right) &= \sum_{i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) \\ &+ \sum_{i < j < l} p((X_k = 1) \cap s_i \cap s_j \cap s_l) - \dots + (-1)^{n+1} p((X_k = 1) \cap s_1 \cap \dots \cap s_n). \end{aligned}$$

Define Δ by

$$\begin{aligned} \Delta &= \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) \\ &- \sum_{i < j < l} p((X_k = 1) \cap s_i \cap s_j \cap s_l) + \dots - (-1)^{n+1} p((X_k = 1) \cap s_1 \cap \dots \cap s_n). \end{aligned}$$

We will now use an inequality that is established in the proof of Leibniz's Theorem in Spivak [8]. Since Δ equals a finite decreasing alternating series then

$$\begin{aligned} \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) - \sum_{i < j < l} p((X_k = 1) \cap s_i \cap s_j \cap s_l) \leq \\ \Delta \leq \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j). \end{aligned}$$

This gives the upper bound. The upper bound for ε can be obtained similarly by just removing X_k from the equations above. ■

Intuitively by looking at Lemma 16 one should expect that $\varepsilon \geq \varepsilon'(k) \geq 0$.

LEMMA 17

$$\varepsilon \geq \varepsilon'(k) \geq 0$$

PROOF: Above in equations (114) and (115) ε and $\varepsilon'(k)$ were defined by

$$\varepsilon'(k) = \sum_{s_i \in \mathbf{S}} p((X_k = 1) \cap s_i) - p\left(\bigcup_{s_i \in \mathbf{S}} ((X_k = 1) \cap s_i)\right)$$

$$\varepsilon = \sum_{s_i \in \mathbf{S}} p(s_i) - p\left(\bigcup_{s_i \in \mathbf{S}} s_i\right).$$

Note that ε can be written

$$\begin{aligned} \varepsilon = \sum_{s_i \in \mathbf{S}} p((X_k = 1) \cap s_i) + \sum_{s_i \in \mathbf{S}} p((X_k = 0) \cap s_i) \\ - p\left(\bigcup_{s_i \in \mathbf{S}} ((X_k = 1) \cap s_i)\right) - p\left(\bigcup_{s_i \in \mathbf{S}} ((X_k = 0) \cap s_i)\right) \end{aligned}$$

Subtracting $\varepsilon'(k)$ from ε gives

$$\varepsilon - \varepsilon'(k) = \sum_{s_i \in \mathbf{S}} p((X_k = 0) \cap s_i) - p\left(\bigcup_{s_i \in \mathbf{S}} ((X_k = 0) \cap s_i)\right).$$

Applying Boole's inequality (see Tucker [10]) gives

$$\sum_{s_i \in \mathbf{S}} p((X_k = 0) \cap s_i) \geq p\left(\bigcup_{s_i \in \mathbf{S}} ((X_k = 0) \cap s_i)\right)$$

with equality if $(X_k = 0) \cap s_i$ are disjoint for all i . Hence it follows that $\varepsilon \geq \varepsilon'(k)$ with equality if $(X_k = 0) \cap s_i$ are disjoint for all i . It remains to show that $\varepsilon'(k) \geq 0$. Applying Boole's inequality,

$$p\left(\bigcup_{s_i \in \mathbf{S}} ((X_k = 1) \cap s_i)\right) \leq \sum_{s_i \in \mathbf{S}} p((X_k = 1) \cap s_i),$$

to (115) gives $\varepsilon'(k) \geq 0$. ■

We now return to the estimation of the quantities described in (i), (ii), (iii) above, starting with (i). Bayes' formula gives

$$p(s_i | top) = \frac{p(top | s_i) p(s_i)}{p(top)} = \frac{p(top | s_i) \prod_{k \in s_i} p_k}{p(top)}.$$

Since $p(top | s_i) = 1$, this simplifies to

$$p(s_i | top) = \frac{\prod_{k \in s_i} p_k}{p(top)} = \frac{\prod_{k \in s_i} p_k}{h(p_1, \dots, p_n)} \quad (116)$$

where we have used the structure function in the last equality. One can approximate (116) by using the rare event approximation. This gives

$$p(s_i | top) = \frac{\prod_{k \in s_i} p(k)}{p(top)} \approx \frac{\prod_{k \in s_i} p_k}{\sum_{s \in S} p(s)} = \frac{\prod_{k \in s_i} p_k}{\sum_{s \in S} \prod_{k \in s} p_k} \quad (117)$$

By using (114) one can obtain error bounds for the approximation in (117).

THEOREM 17

The approximation

$$p(s_i | top) \approx \frac{\prod_{k \in s_i} p_k}{\sum_{s_i \in S} p(s_i)}$$

has absolute error

$$\Delta = \left| p(s_i | top) - \frac{\prod_{k \in s_i} p_k}{\sum_{s_i \in S} p(s_i)} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j) \cdot \prod_{k \in s_i} p_k}{\sum_{s_i \in S} p(s_i) \left(\sum_{s_i \in S} p(s_i) - \sum_{i < j} p(s_i \cap s_j) \right)}$$

and relative error

$$\left| \frac{\Delta}{p(s_i | top)} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j)}{\sum_{s_i \in S} p(s_i)}.$$

PROOF: Using (117) and (114) gives the absolute error

$$\Delta = \left| p(s_i | top) - \frac{\prod_{k \in s_i} p_k}{\sum_{i \in S} p(s_i)} \right| = \frac{\varepsilon \prod_{k \in s_i} p_k}{\sum_{i \in S} p(s_i) (\sum_{i \in S} p(s_i) - \varepsilon)} \quad (118)$$

Using Lemma 16 to substitute for an upper bound for ε gives

$$\Delta \leq \frac{\sum_{i < j} p(s_i \cap s_j) \cdot \prod_{k \in s_i} p_k}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \sum_{i < j} p(s_i \cap s_j))}$$

as an upper bound for the absolute error. The relative error is obtained by dividing (118) by $p(s_i | top)$ in (117) which gives

$$\left| \frac{\Delta}{p(s_i | top)} \right| = \frac{\varepsilon}{\sum_{i \in S} p(s_i)}.$$

Using Lemma 16 to substitute for a upper bound for ε gives

$$\left| \frac{\Delta}{p(s_i | top)} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j)}{\sum_{s_i \in S} p(s_i)}$$

which proves the Theorem. ■

To illustrate the use of the Theorem 17 we will continue to use the fault-tree example in Figure 31 above.

EXAMPLE 1

The structure function of the fault-tree in Figure 31 is

$$\begin{aligned} h(x_1, \dots, x_n) = & x_1 x_2 x_4 + x_1 x_2 x_5 + x_1 x_3 x_6 + x_3 x_4 x_5 - x_1 x_2 x_4 x_5 - x_1 x_3 x_4 x_5 x_6 - x_1 x_2 x_3 x_4 x_6 \\ & - x_1 x_2 x_3 x_4 x_5 - x_1 x_2 x_3 x_5 x_6 + 2x_1 x_2 x_3 x_4 x_5 x_6 \end{aligned}$$

From (113) the probability of the top event can be obtained from the structure function. This gives

$$p(top) = h(p_1, \dots, p_n) = 2.200483851 \cdot 10^{-7}.$$

The conditional probability $p(s_3 | top)$ is obtained from (116). This gives

$$p(s_3 | top) = .9088910146.$$

The approximation in (117) gives

$$p(s_3|top) \approx .9088843445.$$

By comparing it can be seen that the approximation has 4 significant digits and a relative error of $7.3 \cdot 10^{-6}$. To calculate the relative error in Theorem 17 we need

$$\begin{aligned} \sum_{i < j} p(s_i \cap s_j) &= p(X_1 = 1, X_2 = 1, X_4 = 1, X_5 = 1) \\ &\quad + p(X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 1, X_6 = 1) \\ &\quad + p(X_1 = 1, X_2 = 1, X_3 = 1, X_5 = 1, X_6 = 1) \\ &\quad + 2p(X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 1, X_5 = 1) \\ &\quad + p(X_1 = 1, X_3 = 1, X_4 = 1, X_5 = 1, X_6 = 1) \\ &= 1.625 \cdot 10^{-12}. \end{aligned}$$

This gives an upper bound of the relative error of

$$\left| \frac{\Delta}{p(s_3|top)} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j)}{\sum_{s_i \in S} p(s_i)} = \frac{2 \cdot 10^{-12}}{2.200500 \cdot 10^{-7}} = 7.4 \cdot 10^{-6}.$$

■

We have so far only considered a simplified solution to the problem described in (i). We will now look at the more general conditional probability of cut-set s_i given the top-event and the state of Components $k_1, \dots, k_j \in \mathbf{K}$ which is written $p(s_i | top, x_{k_1}, \dots, x_{k_j})$. Using Bayes' formula gives

$$\begin{aligned} p(s_i | top, x_{k_1}, \dots, x_{k_j}) &= \frac{p(s_i, top, x_{k_1}, \dots, x_{k_j})}{p(top, x_{k_1}, \dots, x_{k_j})} \\ &= \frac{p(top | s_i, x_{k_1}, \dots, x_{k_j}) p(s_i, x_{k_1}, \dots, x_{k_j})}{p(top | x_{k_1}, \dots, x_{k_j}) p(x_{k_1}, \dots, x_{k_j})}, \end{aligned}$$

where $\{x_{k_1}, \dots, x_{k_j}\}$ denotes the event $\{X_{k_1} = x_{k_1}, \dots, X_{k_j} = x_{k_j}\}$. Note that if we are in a state such that $top \cap \{x_{k_1}, \dots, x_{k_j}\} = \emptyset$ then the conditional probability does not exist. Since $s_i \subset top$ we have

$$p(top | s_i, x_{k_1}, \dots, x_{k_j}) = 1,$$

and since that the components fail independently,

$$p(s_i|top, x_{k_1}, \dots, x_{k_j}) = \begin{cases} 0, & \text{if } \exists g \text{ with } k_g \in s_i \text{ and } x_{k_g} = 0 \\ 1, & \text{if } s_i \subset \{k_1, \dots, k_j\} \text{ and } x_{k_g} = 1 \forall k_g \in s_i \\ \frac{\prod_{l \in s_i \setminus \{k_1, \dots, k_j\}} p_l}{p(top|x_{k_1}, \dots, x_{k_j})}, & \text{otherwise.} \end{cases}$$

For $l \in \mathbf{K}$ use

$$y_l = \begin{cases} x_l & \text{for } l \in \{k_1, \dots, k_j\} \\ p_l & \text{otherwise} \end{cases}$$

to obtain the conditional probability via the structure function. This gives

$$p(s_i|top, x_{k_1}, \dots, x_{k_j}) = \begin{cases} 0, & \text{if } \exists g \text{ with } k_g \in s_i \text{ and } x_{k_g} = 0 \\ 1, & \text{if } s_i \subset \{k_1, \dots, k_j\} \text{ and } x_{k_g} = 1 \forall k_g \in s_i \\ \frac{\prod_{l \in s_i \setminus \{k_1, \dots, k_j\}} p_l}{h(y_1, \dots, y_n)}, & \text{otherwise.} \end{cases} \quad (119)$$

From (119) one can compute exact numerical values via the structure function. Since (119) is an exact formula it does not assume that the minimal cut-sets are exclusive. This is shown in Example 2.

EXAMPLE 2

Consider the fault-tree in Figure 31 and use (119) to calculate the conditional probability of the cut-sets given the top event and

$$X_1 = 1, X_3 = 0, X_5 = 1, X_6 = 1.$$

We find that

$$p(s_1|top, X_1 = 1, X_3 = 0, X_5 = 1, X_6 = 1) = \frac{p_2 p_4}{h(1, p_2, 0, p_4, 1, 1)} = .00005$$

and

$$p(s_2|top, X_1 = 1, X_3 = 0, X_5 = 1, X_6 = 1) = \frac{p_2}{h(1, p_2, 0, p_4, 1, 1)} = 1$$

Since $X_3 = 0$ then

$$p(s_3|top, X_1 = 1, X_3 = 0, X_5 = 1, X_6 = 1) = 0$$

and

$$p(s_4|top, X_1 = 1, X_3 = 0, X_5 = 1, X_6 = 1) = 0.$$

Note that later in Section 5.3, when we search for the minimal cut-set which causes the top-event in a fault-tree, we have assumed that the minimal cut-sets fail exclusively. Hence, by assumption it follows that when one minimal cut-set has failed then no other cut-sets can have failed. Example 2 shows that although minimal cut-set s_2 has failed given the top event and $\{X_1 = 1, X_3 = 0, X_5 = 1, X_6 = 1\}$ there is also a small probability that minimal cut-set s_1 is also failed. In our method later in Section 5.3 the exclusivity assumption affects the use of (119) such that if we have that there is one minimal cut-set s_i such that $p(s_i | \text{top}, x_{k_1}, \dots, x_{k_j}) = 1$ then the other minimal cut-sets are assumed not to have failed.

Let us see how the conditional probability of the state of Component k given the top event can be obtained e.g. looking at (ii) above. Since

$$p(X_k = 0 | \text{top}) + p(X_k = 1 | \text{top}) = 1$$

it suffices to consider $p(X_k = 1 | \text{top})$. Using Bayes' formula $p(X_k = 1 | \text{top})$ can be written

$$\begin{aligned} p(X_k = 1 | \text{top}) &= \frac{p(\text{top} \cap (X_k = 1))}{p(\text{top})} \\ &= \frac{p(\text{top} | (X_k = 1))p(X_k = 1)}{p(\text{top})}. \end{aligned}$$

Using the structure function gives

$$p(X_k = 1 | \text{top}) = \frac{h(p_1, \dots, p_{k-1}, 1, p_{k+1}, \dots, p_n)p(X_k = 1)}{h(p_1, \dots, p_n)}. \quad (120)$$

From (120) one can obtain the exact value of $p(X_k = 1 | \text{top})$. Calculation of the structure function every time is demanding. Therefore we will now look at two possible approximations. Note that we can write

$$p(X_k = 1 | \text{top}) = \frac{p(\text{top} \cap (X_k = 1))}{p(\text{top})} = \frac{p\left(\bigcup_{s_i \in S} ((X_k = 1) \cap s_i)\right)}{p(\text{top})}.$$

We want to approximate $p(X_k = 1 | \text{top})$ by

$$p(X_k = 1 | \text{top}) \approx \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)}.$$

From (114) and (115) it follows that we can write

$$p(X_k = 1 | top) = \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)}{\sum_{s_i \in S} p(s_i) - \varepsilon}. \quad (121)$$

where $\varepsilon \geq \varepsilon'(k) \geq 0$.

LEMMA 18

The approximation

$$p(X_k = 1 | top) \approx \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)}$$

has absolute error

$$\Delta = \left| p(X_k = 1 | top) - \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)} \right| = \frac{|\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)|}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}$$

and relative error

$$\left| \frac{\Delta}{p((X_k = 1) | top)} \right| = \frac{|\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)|}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}.$$

PROOF: Multiplying each side of (121) by $\sum_{s_i \in S} p(s_i) - \varepsilon$ and solving for

$p((X_k = 1) | top) - \sum_{s_i \in S} p((X_k = 1) \cap s_i) / \sum_{s_i \in S} p(s_i)$ gives

$$\begin{aligned} \Delta &= \left| p((X_k = 1) | top) - \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)} \right| \\ &= \frac{|\varepsilon \cdot p((X_k = 1) | top) - \varepsilon'(k)|}{\sum_{s_i \in S} p(s_i)}. \end{aligned} \quad (122)$$

To calculate the absolute error we substitute (121) for $p(X_k = 1 | top)$. This gives

$$\begin{aligned}
\frac{\varepsilon \cdot p((X_k = 1) | top) - \varepsilon'(k)}{\sum_{s_i \in S} p(s_i)} &= \frac{\varepsilon \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)}{\sum_{s_i \in S} p(s_i) - \varepsilon} - \varepsilon'(k)}{\sum_{s_i \in S} p(s_i)} \\
&= \frac{\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}
\end{aligned}$$

Hence, the absolute error is

$$\Delta = \frac{\left| \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i) \right|}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}.$$

The relative error is obtained by dividing by $p(X_k = 1 | top)$. Using (121), (122) and solving for $\Delta / p(X_k = 1 | top)$ gives

$$\left| \frac{\Delta}{p(X_k = 1 | top)} \right| = \frac{\left| \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i) \right|}{\left(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \right) \sum_{s_i \in S} p(s_i)}.$$

This completes the proof of the Lemma. ■

The problem with the absolute error and the relative error in Lemma 18 appears when we want to insert the upper bound for ε and $\varepsilon'(k)$. It is not possible to see how that affects the error. This problem is fixed in the upper bounds given in Theorem 18.

THEOREM 18

The approximation

$$p(X_k = 1 | \text{top}) \approx \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)}$$

has absolute error bounded by

$$\Delta \leq \frac{\max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}$$

and relative error

$$\left| \frac{\Delta}{p((X_k = 1) | \text{top})} \right| \leq \frac{\max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}.$$

PROOF: Since $\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) \geq 0$ and $\varepsilon'(k) \sum_{s_i \in S} p(s_i) \geq 0$ then

$$\left| \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i) \right| \leq \max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}$$

Substituting this into the absolute error and the relative error in Lemma 18 gives the Theorem. ■

We will now replace the epsilons in the relative error of Theorem 18 with their upper bounds in Lemma 16. This gives an error bound that can be obtained numerically.

COROLLARY 11

The relative error of the approximation

$$p(X_k = 1 | top) \approx \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)}$$

in Theorem 18 is bounded by

$$\left| \frac{\Delta}{p((X_k = 1) | top)} \right| \leq \max \left\{ \frac{\sum_{i < j} p(s_i \cap s_j) \cdot \sum_{s_j \in S} p((X_k = 1) \cap s_j)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)}, \frac{\sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) \cdot \sum_{s_j \in S} p(s_j)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)} \right\}$$

PROOF: Consider the relative error in Theorem 18 and use Lemma 16 to substitute for ε . This gives

$$\left| \frac{\Delta}{p((X_k = 1) | top)} \right| \leq \frac{\max \{ \sum_{i < j} p(s_i \cap s_j) \cdot \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}.$$

If we consider the term in the denominator it can be seen that

$$\begin{aligned} \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) & \\ & \geq \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j). \end{aligned} \quad (123)$$

Substituting (123) into the relative error gives

$$\left| \frac{\Delta}{p((X_k = 1) | top)} \right| \leq \frac{\max \{ \sum_{i < j} p(s_i \cap s_j) \cdot \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)}.$$

By using Lemma 16 again and substituting for $\varepsilon'(k)$ we have

$$\left| \frac{\Delta}{p((X_k = 1) | \text{top})} \right| = \max \left\{ \frac{\sum_{i < j} p(s_i \cap s_j) \cdot \sum_{s_j \in S} p((X_k = 1) \cap s_j)}{(\sum_{i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)}, \frac{\sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) \cdot \sum_{s_j \in S} p(s_j)}{(\sum_{s_j \in S} p((X_k = 1) \cap s_j) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)} \right\}.$$

which proves the Corollary. ■

To calculate the relative error in Corollary 11 one needs to compute the upper bounds of both ε and $\varepsilon'(k)$. In Corollary 12 we will give a rougher error bound than in Corollary 11 that is simpler to compute.

COROLLARY 12

The relative error of the approximation

$$p(X_k = 1 | \text{top}) \approx \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)}$$

in Theorem 18 is bounded by

$$\left| \frac{\Delta}{p((X_k = 1) | \text{top})} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j)}{\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)}$$

PROOF: Recall the relative error in Theorem 18

$$\left| \frac{\Delta}{p((X_k = 1) | \text{top})} \right| \leq \frac{\max_{i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{i \in S} p(s_i)}.$$

Consider the numerator on the right hand-side. Lemma 17 gives

$$\begin{aligned} & \max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \} \\ & \leq \max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon \sum_{s_i \in S} p(s_i) \} \leq \varepsilon \sum_{s_i \in S} p(s_i), \end{aligned} \tag{124}$$

since $p((X_k = 1) \cap s_i) \leq p(s_i)$ for each i . Consider the denominator of the relative error. Apply the inequality (123) from the proof of Corollary 11 and cancel $\sum_{s_i \in S} p(s_i)$. ■

Let us continue Example 1 of the fault-tree in Figure 31 to illustrate the approximation in Theorem 18 with the error bounds in Corollary 11 and Corollary 12.

EXAMPLE 3

Find $p(X_3 = 1 \mid \text{top})$ and $p(X_4 = 1 \mid \text{top})$ exact and by using the approximations above. Give the relative-error bounds for the approximations. From (120) one can obtain the exact values

$$p(X_3 = 1 \mid \text{top}) = \frac{h(p_1, p_2, 1, p_4, p_5, p_6)p(X_3 = 1)}{h(p_1, \dots, p_6)} = 0.9552438892$$

$$p(X_4 = 1 \mid \text{top}) = \frac{h(p_1, p_2, p_3, 1, p_5, p_6)p(X_4 = 1)}{h(p_1, \dots, p_6)} = 0.04571669565$$

Consider the approximation in Theorem 18. Note that the approximation in Theorem 18 can be written

$$\begin{aligned} p(X_k = 1 \mid \text{top}) &= \frac{\sum_{s_i \in S} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)} \\ &= \frac{\sum_{s_i: k \in s_i} p(s_i) + p(X_k = 1) \sum_{s_i: k \notin s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} \end{aligned}$$

Using this gives

$$\begin{aligned} p(X_3 = 1 \mid \text{top}) &\approx \frac{\sum_{s_i: 3 \in s_i} p(s_i) + p(X_3 = 1) \sum_{s_i: 3 \notin s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} = 0.9552419905 \\ p(X_4 = 1 \mid \text{top}) &\approx \frac{\sum_{s_i: 4 \in s_i} p(s_i) + p(X_4 = 1) \sum_{s_i: 4 \notin s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} = 0.04571915474 \end{aligned}$$

When we compare the approximations above with the actual value we find that $p(X_3 = 1 \mid \text{top})$ has 5 significant digits with an relative error of

$$\left| \frac{0.9552419905 - 0.9552438892}{0.9552438892} \right| \approx 2.0 \cdot 10^{-6}$$

and $p(X_4 = 1 \mid \text{top})$ has 4 significant digits with an relative error of

$$\left| \frac{0.04571915474 - 0.04571669565}{0.04571669565} \right| \approx 5.4 \cdot 10^{-5}.$$

Now we will calculate the upper relative error bounds by using Corollary 11. From Example 1 we have that

$$\varepsilon \leq \sum_{i < j} p(s_i \cap s_j) = 1.625 \cdot 10^{-12}.$$

Similarly we need

$$\begin{aligned} \varepsilon'(k) \leq \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) = & p(X_k=1, X_1 = 1, X_2 = 1, X_4 = 1, X_5 = 1) \\ & + p(X_k=1, X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 1, X_6 = 1) \\ & + p(X_k=1, X_1 = 1, X_2 = 1, X_3 = 1, X_5 = 1, X_6 = 1) \\ & + 2p(X_k=1, X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 1, X_5 = 1) \\ & + p(X_k=1, X_1 = 1, X_3 = 1, X_4 = 1, X_5 = 1, X_6 = 1). \end{aligned}$$

This gives

$$\sum_{i < j} p((X_3 = 1) \cap s_i \cap s_j) = 1.135 \cdot 10^{-12}$$

and

$$\sum_{i < j} p((X_4 = 1) \cap s_i \cap s_j) = 6.251 \cdot 10^{-13}.$$

Applying this in Corollary 11 we obtain the upper bound for the relative error as

$$\left| \frac{\Delta}{p(X_3 = 1 | \text{top})} \right| \leq 7.4 \cdot 10^{-6}$$

and

$$\left| \frac{\Delta}{p(X_4 = 1 | \text{top})} \right| \leq 6.2 \cdot 10^{-5}$$

which suggests 4 and 3 significant digits respectively which seems reasonable. We will now calculate the relative error in Corollary 12. Corollary 12 gives

$$\left| \frac{\Delta}{p((X_3 = 1) | \text{top})} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j)}{\sum_{s_i \in S} p((X_3 = 1) \cap s_i) - \sum_{i < j} p((X_3 = 1) \cap s_i \cap s_j)} = 7.7 \cdot 10^{-6}$$

and

$$\left| \frac{\Delta}{p((X_4 = 1) | top)} \right| \leq \frac{\sum_{i < j} p(s_i \cap s_j)}{\sum_{s_i \in S} p((X_4 = 1) \cap s_i) - \sum_{i < j} p((X_4 = 1) \cap s_i \cap s_j)} = 1.6 \cdot 10^{-4} .$$

It can be seen that error bounds are larger than the ones obtained from Corollary 11. However we would still conclude that there is 4 and 3 significant digits respectively.

■

We will now look at the absolute error and the relative error when we use the approximation

$$p(X_k = 1 | top) \approx \frac{\sum_{s_i: k \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} . \quad (125)$$

Let ε and $\varepsilon'(k)$ be as above. Note that $s_i: k \in s_i$ are the cut-sets that contain Component k and $s_i: k \notin s_i$ are the cut-sets that do not contain Component k .

LEMMA 19

The approximation

$$p(X_k = 1 | top) = \frac{\sum_{s_i: k \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)}$$

has absolute error

$$\begin{aligned} \Delta &= \left| p(X_k = 1 | top) - \frac{\sum_{s_i: k \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} \right| \\ &= \left| \frac{\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)} + p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} \right| \end{aligned}$$

and relative error

$$\begin{aligned} \left| \frac{\Delta}{p(X_k = 1 | top)} \right| &= \left| p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} \right. \\ &\quad \left. + \frac{\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} \right| \end{aligned}$$

PROOF: Substitute

$$\sum_{s_i \in S} p((X_k = 1) \cap s_i) = \sum_{s_i: k \in s_i} p(s_i) + \sum_{s_i: k \notin s_i} p((X_k = 1) \cap s_i)$$

into (121) gives

$$p(X_k = 1 | top) = \frac{\sum_{s_i: k \notin s_i} p((X_k = 1) \cap s_i) + \sum_{s_i: k \in s_i} p(s_i) - \varepsilon'(k)}{\sum_{s_i \in S} p(s_i) - \varepsilon}.$$

Multiplying each side by $(\sum_{s_i \in S} p(s_i) - \varepsilon) / \sum_{s_i \in S} p(s_i)$ gives

$$p(X_k = 1 | top) - \frac{\sum_{s_i \in S} p(s_i)}{\sum_{s_i: k \in s_i} p(s_i)} = \frac{\sum_{s_i: k \notin s_i} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)} + \frac{\varepsilon \cdot p((X_k = 1) | top) - \varepsilon'(k)}{\sum_{s_i \in S} p(s_i)}.$$

Substitute (121) for $p(X_k = 1 | top)$ on the right-hand side gives

$$\begin{aligned} p(X_k = 1 | top) - \frac{\sum_{s_i \in S} p(s_i)}{\sum_{s_i: k \in s_i} p(s_i)} &= \frac{\sum_{s_i: k \notin s_i} p((X_k = 1) \cap s_i)}{\sum_{s_i \in S} p(s_i)} \\ &\quad + \frac{\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}. \end{aligned}$$

Note that independence gives

$$\sum_{s_i: k \notin s_i} p((X_k = 1) \cap s_i) = p(X_k = 1) \sum_{s_i: k \notin s_i} p(s_i),$$

so that

$$\begin{aligned} p(X_k = 1 | top) - \frac{\sum_{s_i \in S} p(s_i)}{\sum_{s_i: k \in s_i} p(s_i)} &= p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} \\ &\quad + \frac{\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}. \end{aligned}$$

Taking the absolute value gives the absolute error. The relative error is obtained by dividing Δ by $p(X_k = 1 | top)$, which gives (using (121))

$$\begin{aligned} \left| \frac{\Delta}{p(X_k = 1 | top)} \right| &= \left| p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} \right. \\ &\quad \left. + \frac{\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} \right|. \end{aligned}$$

■

Comparing the absolute error in Lemma 18 and Lemma 19 we can see that the first term of Lemma 19 is the same as the absolute error in Lemma 18. The second term in Lemma 19 is the additional error caused by the rougher estimate. A second way to get

an error-bound for the approximation in Lemma 19 is to establish an upper bound. Similar to Theorem 18 we can establish upper bounds for the approximation in (125).

THEOREM 19

The approximation

$$p(X_k = 1 | \text{top}) \approx \frac{\sum_{s_i: k \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)}$$

has absolute error

$$\Delta \leq \frac{\max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{\sum_{s_i \in S} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)} + p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)}$$

and relative error

$$\left| \frac{\Delta}{p(X_k = 1 | \text{top})} \right| \leq p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} + \frac{\max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}.$$

PROOF: The proof is direct from Lemma 19. Note that since $\varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) \geq 0$

and $\varepsilon'(k) \sum_{s_i \in S} p(s_i) \geq 0$ then

$$| \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k) \sum_{s_i \in S} p(s_i) | \leq \max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}.$$

Applying this inequality to Lemma 19 proves the Theorem. ■

We will now replace the epsilons in Theorem 19 to get an upper relative error bound that can be estimated. This is obtained in Corollary 13.

COROLLARY 13

The relative error of the approximation

$$p(X_k = 1 | \text{top}) \approx \frac{\sum_{s_i: k \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)}$$

is bounded by

$$\left| \frac{\Delta}{p(X_k = 1 | \text{top})} \right| \leq p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i)}{\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)} + \frac{\max_{i < j} \{ \sum_{s_i \in S} p(s_i \cap s_j) \cdot \sum_{s_i \in S} p((X_k = 1) \cap s_i), \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) \} \sum_{s_i \in S} p(s_i)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)}.$$

PROOF: Recall the relative error in Theorem 19

$$\left| \frac{\Delta}{p(X_k = 1 | \text{top})} \right| \leq p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} + \frac{\max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}.$$

Consider the first term and note that

$$\sum_{s_i: k \notin s_i} p(s_i) (\sum_{s_i \in S} p(s_i) - \varepsilon) \leq \sum_{s_i: k \notin s_i} p(s_i) \cdot \sum_{s_i \in S} p(s_i)$$

which gives

$$\left| \frac{\Delta}{p(X_k = 1 | \text{top})} \right| \leq p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i) \cdot \sum_{s_i \in S} p(s_i)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} + \frac{\max_{s_i \in S} \{ \varepsilon \sum_{s_i \in S} p((X_k = 1) \cap s_i), \varepsilon'(k) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}.$$

Replacing $\varepsilon'(k)$ with its upper term in Lemma 16 proves the first term in Corollary 13. Consider the last term. It can be seen that it equals the relative error in Theorem 18. Hence, Corollary 11 gives

$$\begin{aligned}
& \frac{\max_{s_i \in S} \{ \varepsilon \sum p((X_k = 1) \cap s_i), \varepsilon'(k) \sum p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)} \\
& \leq \frac{\max_{i < j} \{ \sum_{s_i \in S} p(s_i \cap s_j) \cdot \sum_{s_i \in S} p((X_k = 1) \cap s_i), \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j) \sum_{s_i \in S} p(s_i) \}}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \sum_{i < j} p((X_k = 1) \cap s_i \cap s_j)) \sum_{s_i \in S} p(s_i)}.
\end{aligned}$$

which proves the Corollary. ■

Note that where we used Corollary 11 to establish the last term of the relative error in Corollary 13 we could also have used Corollary 12 to provide a rougher estimate. Let us continue Example 1 and Example 3 of the fault-tree in Figure 31 to illustrate the approximation in Theorem 19.

EXAMPLE 4

Find $p(X_3 = 1 \mid \text{top})$ and $p(X_4 = 1 \mid \text{top})$ by using the approximation in Theorem 19 and calculate the relative error by using Corollary 13. The approximation in Theorem 19 gives

$$p(X_3 = 1 \mid \text{top}) \approx \frac{\sum_{s_i: 3 \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} = 0.9543285617$$

and

$$p(X_4 = 1 \mid \text{top}) \approx \frac{\sum_{s_i: 4 \in s_i} p(s_i)}{\sum_{s_i \in S} p(s_i)} = 0.04567143831.$$

Comparing with the exact values above we find that $p(X_3 = 1 \mid \text{top})$ has 2 significant digits with a relative error of

$$\left| \frac{0.9543285617 - 0.9552438892}{0.9552438892} \right| \approx 9.58 \cdot 10^{-4}$$

and $p(X_4 = 1 \mid \text{top})$ has 2 significant digits with an relative error of

$$\left| \frac{0.04567143831 - 0.04571669565}{0.04571669565} \right| \approx 9.90 \cdot 10^{-4}.$$

Now we will calculate the upper relative error bounds by using Corollary 13.

It suffices to calculate

$$p(X_k = 1) \frac{\sum_{s_i: k \notin s_i} p(s_i) \cdot \sum_{s_i \in S} p(s_i)}{(\sum_{s_i \in S} p((X_k = 1) \cap s_i) - \varepsilon'(k)) \sum_{s_i \in S} p(s_i)}$$

and add it to the relative error obtained by using Corollary 11 in Example 3 above. This gives

$$\left| \frac{\Delta}{p(X_3 = 1 | top)} \right| \leq 7.4 \cdot 10^{-6} + 9.56 \cdot 10^{-4} = 9.64 \cdot 10^{-4}$$

and

$$\left| \frac{\Delta}{p(X_4 = 1 | top)} \right| \leq 6.2 \cdot 10^{-5} + 1.044 \cdot 10^{-3} \approx 1.1 \cdot 10^{-3}$$

which suggests two significant digits as above. ■

Now we will look at the conditional probability of the state of Component j given the top and the states of Components $k_1, \dots, k_i$. This is the probability in 3) above on page 154. Let $j, k_1, \dots, k_i \in \mathbf{K}$ such that $j \neq k_l$ for all $l \in \{1, \dots, i\}$. Using Bayes' formula gives

$$p(x_j | top, x_{k_1}, \dots, x_{k_i}) = \frac{p(x_j, top, x_{k_1}, \dots, x_{k_i})}{p(top, x_{k_1}, \dots, x_{k_i})}.$$

Note that we need to assume that $p(top, x_{k_1}, \dots, x_{k_i}) \neq 0$. When this is not the case the state of the Components $k_1, \dots, k_i$ are such that the top-event is impossible. Use of Bayes' formula and the independence of the component failure gives

$$p(x_j | top, x_{k_1}, \dots, x_{k_i}) = \frac{p(top | x_{k_1}, \dots, x_{k_i}, x_j) p(x_j)}{p(top | x_{k_1}, \dots, x_{k_i})} \quad (126)$$

Similar to (113) the structure function can be used to obtain the conditional probabilities. Let

$$y_l = \begin{cases} x_l & \text{for } l \in \{k_1, \dots, k_i, j\} \\ p_l & \text{otherwise} \end{cases}$$

so that

$$p(top | x_{k_1}, \dots, x_{k_i}, x_j) = h(y_1, \dots, y_n),$$

and let

$$y'_l = \begin{cases} x_l & \text{for } l \in \{k_1, \dots, k_i\} \\ p_l & \text{otherwise.} \end{cases}$$

so that

$$p(top | x_{k_1}, \dots, x_{k_i}) = h(y'_1, \dots, y'_n)$$

Now (126) can be written

$$p(x_j | top, x_{k_1}, \dots, x_{k_i}) = \frac{h(y'_1, \dots, y'_n) p(x_j)}{h(y'_1, \dots, y'_n)}. \quad (127)$$

When $h(y'_1, \dots, y'_n) = 0$ the conditional probability does not exist. From (127) one can obtain the exact numerical values of the probabilities.

We will now illustrate approximations for (127). Similarly for any $k_1, k_2 \in \mathbf{K}$ such that $k_1 \neq k_2$ then via Bayes' Theorem $p(X_{k_1} = 1 | top, X_{k_2} = 1)$ can be written

$$p(X_{k_1} = 1 | top, X_{k_2} = 1) = \frac{p(X_{k_1} = 1 \cap X_{k_2} = 1 | top)}{p(X_{k_2} = 1 | top)}.$$

Consider the denominator and recall the approximation in Theorem 19. This gives

$$p(X_{k_2} = 1 | top) \approx \frac{\sum_{s: k_2 \in s} p(s)}{\sum_{s \in S} p(s)}.$$

Similarly one can approximate the numerator by

$$p(X_{k_1} = 1 \cap X_{k_2} = 1 | top) \approx \frac{\sum_{s: k_1, k_2 \in s} p(s)}{\sum_{s \in S} p(s)}$$

which suggests the approximation

$$p(X_{k_1} = 1 | top, X_{k_2} = 1) \approx \frac{\sum_{s: k_1 \in s, k_2 \in s} p(s | top)}{\sum_{s: k_2 \in s} p(s | top)}. \quad (128)$$

EXAMPLE 5

We will use (127) and the approximation (128) to calculate $p(X_3 = 1 | top, X_4 = 1)$ and $p(X_3 = 1 | top, X_4 = 0)$ in the fault-tree in Figure 31. We will then compute the relative error. The exact computation (127) gives

$$p(X_3 = 1 | top, X_4 = 1) = .9951291690$$

and

$$p(X_3 = 1 | top, X_4 = 0) = .9533331111.$$

The approximation in (128) gives

$$p(X_3 = 1 | top, X_4 = 1) = \frac{p(s_4 | top)}{p(s_4 | top) + p(s_1 | top)} = .9950248755$$

and

$$p(X_3 = 1 | top, X_4 = 0) = \frac{p(s_3 | top)}{p(s_2 | top) + p(s_3 | top)} = .9523809524.$$

The relative error of $p(X_3 = 1 | top, X_4 = 1)$ is

$$|.9951291690 - .9950248755| / .9951291690 = 1.05 \cdot 10^{-4}$$

and the relative error of $p(X_3 = 1 | top, X_4 = 0)$ is

$$|.9533331111 - .9523809524| / .9533331111 = 9.99 \cdot 10^{-4}$$

This suggests that the approximation work well.

■

5.3 Value of Information based Inspections Strategy

Let the set of minimal cut-sets be denoted by S and let the set of Components be $K = \{1, 2, \dots, n\}$. Let the inspections be denoted by $\psi = \{X_1, X_2, \dots, X_n\}$, where the index identifies the component. Each inspection X_i , $i \in K$ has two possible outcomes

$$X_i = \begin{cases} 1 & \text{if component } i \text{ is failed,} \\ 0 & \text{otherwise.} \end{cases}$$

An act is to identify which of the minimal cut-sets that has caused the system to fail. The set of available acts, F , is indexed by the collection of minimal cut-sets. Let $S = \{s_1, \dots, s_m\}$ and let $F = \{f_1, \dots, f_m\}$. The act f_i , $i \in \{1, \dots, m\}$ is to decide that minimal cut set s_i has failed.

The utility function is generally defined over $F \times S$. Let π be a subset of K containing the elements $\{\pi_1, \dots, \pi_k\}$. We will apply the utility function

$$U_{\pi_1, \dots, \pi_k}(f_i, s_j) = \begin{cases} -\sum_{l=1}^k \kappa_l & \text{if } i = j \\ -\sum_{l=1}^k \kappa_l - u & \text{otherwise} \end{cases} \quad (129)$$

here. The cost of inspecting Component i is κ_i . The inspection strategy is computed by considering test-schedules of the kind

$$[\dots [\emptyset \rightarrow X_1] \rightarrow \dots] \rightarrow X_n.$$

The problem is that there are $n!$ test-sequences corresponding to the permutations of K .

Clearly if we choose to inspect all the components then we will know which cut-sets that have failed. In this model parameters u and κ regulate the tests that we would like to use in order to optimise the expected value of the problem. For example if we take the cost $\kappa_i = 1$ for all i and $u = \infty$ then any finite price will be paid to avoid choosing the wrong minimal cut-set. Minus the expected value of the problem is then equal to the expected number of steps required to find a failed minimal cut-set. On the other hand if all inspections were free we could inspect the entire system to determine out the components causing the system failure.

Lemma 20 is based on the assumptions above and is useful later.

LEMMA 20

Define $F = \{f_1, \dots, f_m\}$, $S = \{s_1, \dots, s_m\}$ and let $\pi = \{\pi_1, \dots, \pi_k\}$ be a subset of Components **K**. Take the utility function

$$U_{\pi_1, \dots, \pi_k}(f_i, s_j) = \begin{cases} -\sum_{l=1}^k \kappa_{\pi_l} & \text{if } i = j \\ -\sum_{l=1}^k \kappa_{\pi_l} - u & \text{otherwise,} \end{cases} \quad (130)$$

then

$$v(F | x_{\pi_1}, \dots, x_{\pi_k}) = u \cdot \max_{s \in S} p(s | x_{\pi_1}, \dots, x_{\pi_k}) - u - \sum_{l=1}^k \kappa_{\pi_l}$$

holds.

PROOF: By definition we have that

$$v(F | x_{\pi_1}, \dots, x_{\pi_k}) = \max_{f \in F} \sum_{s \in S} p(s | x_{\pi_1}, \dots, x_{\pi_k}) U_{\pi_1, \dots, \pi_k}(s, f).$$

Substituting (130) for the utility function gives

$$\max_{f_i \in F} \left(-p(s_i | x_{\pi_1}, \dots, x_{\pi_k}) \sum_{l=1}^k \kappa_{\pi_l} - \sum_{s_j \in S: i \neq j} p(s_j | x_{\pi_1}, \dots, x_{\pi_k}) \left(\sum_{l=1}^k \kappa_{\pi_l} + u \right) \right).$$

Applying

$$\sum_{s_j \in S: i \neq j} p(s_j | x_{\pi_1}, \dots, x_{\pi_k}) = 1 - p(s_i | x_{\pi_1}, \dots, x_{\pi_k})$$

gives

$$\max_{f_i \in F} \left([p(s_i | x_{\pi_1}, \dots, x_{\pi_k}) - 1] \left(\sum_{l=1}^k \kappa_{\pi_l} + u \right) - p(s_i | x_{\pi_1}, \dots, x_{\pi_k}) \sum_{l=1}^k \kappa_{\pi_l} \right).$$

Simplifying gives

$$u \cdot \max_{s \in S} p(s | x_{\pi_1}, \dots, x_{\pi_k}) - \sum_{l=1}^k \kappa_{\pi_l} - u,$$

which proves the Lemma. ■

Let us use the decision model above to optimise the inspection strategy of the fault-tree in Figure 31. We will calculate the probabilities by using the exact numerical values as shown in the examples in Section 5.2

EXAMPLE 6

Find the optimal inspection sequence of the fault-tree in Figure 31 based on

$$\hat{\Phi} = \text{Argmax}_{\pi \in \text{Per}(n)} v(F | [\dots [\emptyset \rightarrow X_{\pi_1}] \rightarrow \dots] \rightarrow X_{\pi_n}),$$

and draw the decision tree for the optimal solution. We will use cost $\kappa_i = 1$ for all i and $u = \infty$ in (129) so that the expected value of information is equal to the expected number of inspections. The expected number of inspection is the negative of the expected value of information that is

$$\max_{\pi \in \text{Per}(6)} v(F | [\dots [\emptyset \rightarrow X_{\pi_1}] \rightarrow \dots] \rightarrow X_{\pi_6}) = v(F | [[X_4 \rightarrow X_1] \rightarrow X_2] \rightarrow X_5) \approx -2.05.$$

The inspection strategy is shown in Figure 34.

Figure 34. Inspection strategy

Note that the solution above is not unique. The test-sequence $(X_4, X_1, X_2, X_6, X_5)$ gives the same expected value and has a diagram similar to the one in Figure 34. ■

The number of computation to obtain the result in Figure 35 is 6! times the computations to calculate the choice-tree for each permutation. Therefore some heuristic approaches are needed.

5.3.1 Inspections Sequenced by a Greedy Algorithm

We will first consider the Greedy-Sequence before Test algorithm from Section 4.3. To distinguish the different sequences we use the notation $\dot{X}$ to denote a test sequenced by the Greedy-Sequence before Test while a test denoted by two dots is sequenced by the Greedy-Test before sequence algorithm. Let us use the assumptions we made in the beginning of Section 5.3.

THEOREM 20

Given the conditions in Lemma 20. Then the i 'th step of the Greedy-Sequence before Test is to sequence test $\dot{T}_{(i)}$ such that

$$\dot{X}_{(i)} = \underset{j \in N \setminus \{(1), \dots, (i-1)\}}{\text{Argmax}} \sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, \dot{x}_j) \max_{s \in S} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, \dot{x}_j) - \frac{\kappa_j}{u}.$$

PROOF: The i 'th step of Greedy-Sequence before Test is

$$v(F | \dot{X}_{(1)}, \dots, \dot{X}_{(i-1)}, \dot{X}_{(i)}) = \max_{j \in N \setminus \{(1), \dots, (i-1)\}} \{v(F | \dot{X}_{(1)}, \dots, \dot{X}_{(i-1)}, X_j)\}.$$

Expanding gives

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \left(\sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) v(F | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) \right).$$

Applying Lemma 20 and using that $\kappa_k = \kappa$ for all k gives

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \left(\sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) (u \cdot \max_{s \in S} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) - \kappa_j - \sum_{l=1}^{i-1} \kappa_{(l)} - u) \right).$$

Since u , i and κ_k are constant when taking the max over j it suffices to maximise

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \left(\sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) (u \cdot \max_{s \in S} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) - \kappa_j) \right)$$

Hence, it suffices to maximise

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \left(\sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) \max_{s \in S} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) - \frac{\kappa_j}{u} \right).$$

This completes the proof of the Theorem. ■

Theorem 20 shows that the expected probability of deciding the correct minimal cut-set given the inspections $(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_i)$ and the test cost of test X_i are important when the Greedy-Sequence before Test algorithm orders the tests.

COROLLARY 14

Given the conditions in Lemma 20. If

i) for all $k \in \mathbf{K}$ the inspection cost for test X_k is $\kappa_k = \kappa$ or

ii) if for all $k \in \mathbf{K}$ we have that $u \gg \kappa_k$ so that $\frac{\kappa_k}{u} \approx 0$

then the i 'th step of the Greedy-Sequence before Test is to take test $\dot{X}_{(i)}$ to be

$$\dot{X}_{(i)} = \underset{j \in \mathbf{K} \setminus \{(1), \dots, (i-1)\}}{\text{Argmax}} \sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) \max_{s \in \mathbf{S}} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j).$$

PROOF: From Theorem 20 the i 'th step of the Greedy-Sequence before Test is

$$\dot{X}_{(i)} = \underset{j \in \mathbf{K} \setminus \{(1), \dots, (i-1)\}}{\text{Argmax}} \sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) \max_{s \in \mathbf{S}} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) - \frac{\kappa_j}{u}.$$

It can be seen that if $\kappa_k = \kappa$ for all k then it suffices to consider

$$\dot{X}_{(i)} = \underset{j \in \mathbf{K} \setminus \{(1), \dots, (i-1)\}}{\text{Argmax}} \sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) \max_{s \in \mathbf{S}} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j).$$

This proves (i) above. If $u \gg \kappa_k$ so that $\frac{\kappa_k}{u} \approx 0$ it can be seen that (ii) of the Corollary holds. ■

Corollary 14 gives conditions for which it is sufficient to only consider the expected probability of deciding the correct minimal cut-set given the inspections $(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_i)$. Note that when $u \rightarrow \infty$ so that $\frac{\kappa_k}{u} \rightarrow 0$ then (ii) of the corollary applies. Just as Fussell-Vesely measure of importance one can create the value of information measure of importance based on Corollary 14. The value of information importance of Component j given $(\dot{X}_{(1)}, \dots, \dot{X}_{(i-1)})$ is

$$\text{VI}(j | \dot{X}_{(1)}, \dots, \dot{X}_{(i-1)}) = \sum_{\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j} p(\dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j) \max_{s \in \mathbf{S}} p(s | \dot{x}_{(1)}, \dots, \dot{x}_{(i-1)}, x_j).$$

We will now continue the example in Figure 31 and Example 6 and use the Greedy-Sequence before test algorithm to determine the test-sequence.

EXAMPLE 7

Recall Example 6 and find the optimal inspection sequence of the fault-tree in Figure 31 based on the Greedy-Sequence before test algorithm. Draw the decision tree for the optimal solution. We will use cost $\kappa_i = 1$ for all i and $u = \infty$ in (129) so that Corollary 14 holds. Calculating the value of information importance gives

$$\begin{aligned} VI(1) &= .9542376896, VI(2) = .9538739082, VI(3) = .9534195951, \\ VI(4) &= .9542830153, VI(5) = .9452433074, VI(6) = .9541021620. \end{aligned}$$

It can be seen that Component 4 has the highest value of information importance so that $\dot{X}_{(1)} = X_4$. The value of information importance given X_4 is

$$\begin{aligned} VI(1|X_4) &= .9544203910, VI(2|X_4) = .9994754158, VI(3|X_4) = .9990411253, \\ VI(5|X_4) &= .9908623168, VI(6|X_4) = .9994979450. \end{aligned}$$

It can be seen that Component 6 has the highest value of information importance given X_4 so that $\dot{X}_{(2)} = X_6$. Calculating the value of information importance given X_4 and X_6 gives

$$\begin{aligned} VI(1|X_4, X_6) &= .9996791796, VI(2|X_4, X_6) = .9997041725, \\ VI(3|X_4, X_6) &= .9999500452, VI(5|X_4, X_6) = .9997667776. \end{aligned}$$

Here Component 3 has the highest value of information importance so that $\dot{X}_{(3)} = X_3$. Calculating the value of information importance given X_3, X_4 and X_6 gives

$$\begin{aligned} VI(1|X_3, X_4, X_6) &= .9999954375, VI(2|X_3, X_4, X_6) = .9999500631, \\ VI(5|X_3, X_4, X_6) &= 1.0000000000. \end{aligned}$$

Component 5 has the highest value of information importance so that $\dot{X}_{(4)} = X_5$. Since one can always determine which minimal cut-set that has failed after inspecting X_3, X_4, X_5 and X_6 the two last components have value of information importance given X_3, X_4, X_5 and X_6 equal to one. Hence the test-sequence from the Greedy-Sequence before test algorithm is $(X_4, X_6, X_3, X_5, X_1, X_2)$ or $(X_4, X_6, X_3, X_5, X_2, X_1)$. Calculating the expected value of information gives

$$v(F|[[[[[\emptyset \rightarrow X_4] \rightarrow X_6] \rightarrow X_3] \rightarrow X_5] \rightarrow X_1] \rightarrow X_2)) \approx -2.10007$$

$$v(F|[[[[[\emptyset \rightarrow X_4] \rightarrow X_6] \rightarrow X_3] \rightarrow X_5] \rightarrow X_2] \rightarrow X_1)) \approx -2.10007.$$

Recall that the expected number of inspections is the negative of the expected value of information. The inspection strategy for both test-sequences is the same and is given in Figure 36.

Figure 36. Inspection strategy

This result can be compared to the result in Example 6. It can be seen that the expectation is 0.05 worse. The Greedy-Sequence before Test did pretty well here considering the amount of computation required to achieve the optimal sequence in Example 6.

We will now consider the Greedy-Test before Sequence algorithm under the same conditions as in Theorem 20.

THEOREM 21

Given the conditions in Lemma 20. Then the i 'th step of the Greedy-Test before Sequence is take test $\ddot{X}_{(i)}$ to be

$$\ddot{X}_{(i)} = \underset{j \in N \setminus \{(1), \dots, (i-1)\}}{\text{Argmax}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \cdot \max \left\{ \begin{array}{l} \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{\ddot{x}_j} p(t_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) - \frac{\kappa_j}{u} \end{array} \right.$$

PROOF: The i 'th step of the Greedy-Test before Sequence is

$$v(F \mid (\ddot{X}_{(1)}, \dots, \ddot{X}_{(i)})) = \max_{j \in N \setminus \{(1), \dots, (i-1)\}} v(F \mid (\ddot{X}_{(1)}, \dots, \ddot{X}_{(i-1)}) \rightarrow X_j).$$

Expanding the expression gives

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max \left\{ \begin{array}{l} v(F \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) v(F \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) \end{array} \right.$$

Applying Lemma 20 to upper and lower term in the max gives

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \cdot \max \left\{ \begin{array}{l} u \cdot \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) - u - \sum_{l=1}^{i-1} \kappa_l \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) (u \cdot \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) - u - \sum_{l=1}^{i-1} \kappa_l - \kappa_j) \end{array} \right.$$

When we take the max over j then κ_k and u are constant for all $k \neq j$ so that it suffices to consider

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max \left\{ \begin{array}{l} u \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ u \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) - \kappa_j \end{array} \right.$$

Simplifying gives

$$u \max_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max \left\{ \begin{array}{l} \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) - \frac{\kappa_j}{u} \end{array} \right.$$

Hence it suffices to maximise

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max \left\{ \begin{array}{l} \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) - \frac{\kappa_j}{u} \end{array} \right.$$

Taking the argument of the maximum completes the proof of the Theorem. ■

Theorem 21 shows that the next test is chosen such it maximises the expected probability given $(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)})$ with the option to maybe do test $\ddot{X}_{(i)}$. Let us consider an interesting case of Theorem 21.

COROLLARY 15

Given the conditions in Lemma 20. If for all $k \in \mathbf{K}$ we have that $u \gg \kappa_k$ so that $\frac{\kappa_k}{u} \approx 0$ then the i 'th step of the Greedy-Test before Sequence is to sequence test $\ddot{X}_{(i)}$ such that

$$\ddot{X}_{(i)} = \text{Argmax}_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max \left\{ \begin{array}{l} \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) \end{array} \right.$$

PROOF: According to Theorem 21 we should consider

$$\max_{j \in N \setminus \{(1), \dots, (i-1)\}} \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max \left\{ \begin{array}{l} \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) - \frac{\kappa_j}{u} \end{array} \right.$$

The example in Figure 31 will be continued. For convenience we define the conditional value of information importance for the Greedy-Test before Sequence algorithm as

$$\text{VCI}(j \mid \ddot{X}_{(1)}, \dots, \ddot{X}_{(i-1)}) = \sum_{\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}} p(\ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \cdot \max \left\{ \begin{array}{l} \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \\ \sum_{x_j} p(x_j \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}) \max_{s \in S} p(s \mid \ddot{x}_{(1)}, \dots, \ddot{x}_{(i-1)}, x_j) \end{array} \right.$$

EXAMPLE 8

We will use the Greedy-Test before Sequence algorithm to calculate the test-sequence. Since for all $k \in \mathbf{K}$ we have that $u \gg \kappa_k$ so that $\frac{\kappa_k}{u} \approx 0$ then Corollary 15 holds. Calculating the conditional value of information importance gives

$$\begin{aligned} \text{VCI}(1) &= .9542376896, \text{VCI}(2) = .9538739082, \text{VCI}(3) = .9534195951, \\ \text{VCI}(4) &= .9542830153, \text{VCI}(5) = .9452433074, \text{VCI}(6) = .9541021620. \end{aligned}$$

It can be seen that Component 4 has the highest value of information importance so that $\ddot{X}_{(1)} = X_4$. The conditional value of information importance given X_4 is

$$\begin{aligned} \text{VCI}(1|X_4) &= .9544203910, \text{VCI}(2|X_4) = .9994754158, \\ \text{VCI}(3|X_4) &= .9990411253, \text{VCI}(5|X_4) = .9908623168, \\ \text{VCI}(6|X_4) &= .9994979450. \end{aligned}$$

It can be seen that Component 6 has the highest value of information importance given X_4 so that $\ddot{X}_{(2)} = X_6$. Calculating the conditional value of information importance given X_4 and X_6 gives

$$\begin{aligned} \text{VCI}(1|X_4, X_6) &= .9996791796, \text{VCI}(2|X_4, X_6) = .9997041725, \\ \text{VCI}(3|X_4, X_6) &= .9999500452, \text{VCI}(5|X_4, X_6) = .9997667776. \end{aligned}$$

Here Component 3 has the highest value of information importance so that $\ddot{X}_{(3)} = X_3$. Calculating the conditional value of information importance given X_3, X_4 and X_6 gives

$$\begin{aligned} \text{VCI}(1|X_3, X_4, X_6) &= .9999954375, \text{VCI}(2|X_3, X_4, X_6) = .9999500631, \\ \text{VCI}(5|X_3, X_4, X_6) &= 1.000000000. \end{aligned}$$

Component 5 has the highest value of information importance so that $\ddot{X}_{(4)} = X_5$. When compared to Example 7 we see that the sequence provided here is the same, hence giving the same result as Example 7.

■

When we compare the results of the greedy algorithms in Example 7 and Example 8 to the exhaustive algorithm in Example 6 above it seems that both greedy algorithms gives good results. Note that for this fault-tree they give exactly the same result.

5.3.2 Test-Schedule Sorting

In the previous Section 5.3.1 we considered greedy algorithms that obtained a test-sequence in $O(n^2)$ time. We will now look at an algorithm that is similar to an insertion-sort algorithm. This algorithm takes $O(n^2)$ times the time to compute the choice-tree (see Theorem 14). Note that $O(n^2)$ is much better than the $O(n!)$ time it takes to try all possible test-sequences e.g. see Example 6.

Let Φ be the initial test-schedule

$$\Phi = [\dots [\emptyset \rightarrow X_{I_1}] \rightarrow \dots] \rightarrow X_{I_n}$$

where I is the initial permutation of the Components $\mathbf{K}$.

Algorithm Test-Schedule Sort:

$$X_{(1)} = \underset{i \in K}{\text{Argmax}} v(F | [\emptyset \rightarrow X_i] \rightarrow (\Phi - X_i))$$

$$X_{(2)} = \underset{i \in K \setminus \{1\}}{\text{Argmax}} v(F | [[\emptyset \rightarrow X_{(1)}] \rightarrow X_i] \rightarrow ((\Phi - X_{(1)}) - X_i))$$

$\vdots$

$$X_{(n)} = \underset{i \in K \setminus \{1, \dots, (n-1)\}}{\text{Argmax}} \{ \\ v(F | [\dots [\emptyset \rightarrow X_{(1)}] \rightarrow \dots] \rightarrow X_{(n-1)}] \rightarrow X_i] \rightarrow (\dots (\Phi - X_{(1)}) \dots) - X_{(n-1)}) - X_i) \}$$

The output is the test-sequence $(X_{(1)}, \dots, X_{(n)})$.

Each $X_{(i)}$, $i \in \{1, \dots, n\}$ in the procedure can have more than one candidate test in K which would give the same expected value of information at level i . Therefore the outcome of the Test-Schedule Sort algorithm depends on how the candidate test is chosen by the algorithm. Therefore we will first give a pseudo-code implementation of the algorithm that we will use in later in Example 9. The pseudo-code conventions we use can be found in Cormen et al [9]. The Test-Schedule-Sort below takes as a parameter an array $\Phi[1..n+1]$ which stores the initial test-schedule-sequence $(\emptyset, X_{i_1}, X_{i_2}, \dots, X_{i_n})$. The output is a sorted test-schedule-sequence, an array *SortedTestSchedule*[1..n+1] stores the sorted sequence. The algorithm uses an array *TestChosen*[1..n] to indicate which of the tests of $\Phi[1..n+1]$ that has been sorted such that

$$\text{TestChosen}[j] = \begin{cases} 1 & \text{if } \Phi[j+1] \text{ is sorted} \\ 0 & \text{otherwise.} \end{cases}$$

The pseudo code is:

TEST-SCHEDULE-SORT(Φ)

```

1  ▷ Initially no tests are sorted.
2  for j ← 1 to n
3    do
4      TestChosen[j] ← 0
5  SortedTestSchedule[1] ←  $\emptyset$ 
6  for j ← 2 to n+1
7    do
8      Key ←  $-\infty$ 
9      for i ← 1 to n
10       do
11         ▷ Insert  $\Phi[i]$  into SortedTestSchedule[j].
12         if TestChosen[i] < 1
13           then SortedTestSchedule[j] ←  $\Phi[i+1]$ 
14           TestChosen[i] ← 1
15         ▷ Insert non sorted tests into end of SortedTestSchedule.
16         Position ← j

```

```

17      for  $k \leftarrow 1$  to  $n$ 
18      do
19          if  $TestChosen[k] < 1$ 
20          then  $Position \leftarrow Position + 1$ 
21               $SortedTestSchedule[Position] \leftarrow \Phi[k+1]$ 
22       $TestChosen[i] \leftarrow 0$ 
23      if  $v(F | SortedTestSchedule) \geq Key$ 
24      then  $Chosentest \leftarrow i$ 
25           $Key \leftarrow v(F | SortedTestSchedule)$ 
26      ▷ Insert  $Chosentest$  into  $SortedTestSchedule[j]$ .
27       $TestChosen[Chosentest] \leftarrow 1$ 
28       $SortedTestSchedule[j] \leftarrow \Phi[Chosentest+1]$ 
29  return  $SortedTestSchedule$ 

```

The *if* statement in Lines 23 to 25 chooses which test that is the next to be. For example if one changes the less or equality sign, $\geq$, in Line 23 to strict inequality $>$ one would select the first test that gives $v(F | SortedTestSchedule) > Key$. Less or equality sign, $\geq$, selects the last test. Let us continue the example of the fault-tree in Figure 31 and use Test-Schedule-Sort to provide a test-sequence.

EXAMPLE 9

Calculate an inspection order by using the test-sequence from the Greedy-Sequence before Test in Example 7 as the initial sequence for Test-Schedule-Sort. The initial sequence is $(X_4, X_6, X_3, X_5, X_1, X_2)$. The initial expected value of information is

$$v(F | [\emptyset \rightarrow X_4] \rightarrow X_6 \rightarrow X_3 \rightarrow X_5 \rightarrow X_1 \rightarrow X_2) \approx -2.10007.$$

Test-Schedule-Sort is iterated until two successive runs give the same expected value. The first run of the algorithm gives

$$v(F | [\emptyset \rightarrow X_2] \rightarrow X_3 \rightarrow X_6 \rightarrow X_4 \rightarrow X_1 \rightarrow X_5) \approx -2.09161.$$

The second run gives

$$v(F | [\emptyset \rightarrow X_4] \rightarrow X_1 \rightarrow X_6 \rightarrow X_3 \rightarrow X_2 \rightarrow X_5) \approx -2.04639.$$

The third run gives

$$v(F | [\emptyset \rightarrow X_3] \rightarrow X_6 \rightarrow X_4 \rightarrow X_1 \rightarrow X_2 \rightarrow X_5) \approx -2.04639.$$

The inspection strategy obtained from the choice-tree is shown in Figure 34 above. We can see that the expected number of inspections is the same as obtained in Example 6.

■

The cost of obtaining the result in Example 9 is 3 times $O(6^2)$ times the time to compute the choice-tree. This is to be compared to $O(6!)$ times the time to compute the choice-tree in Example 6.

5.4 Wang's Algorithm and Fussell-Vesely Importance Approaches.

Recall the notation in Section 5.2. The Fussell-Vesely importance measure for Component i is the probability that Component i is in a failed minimal cut-set given the top event. This is written

$$I^{VF}(i) = p\left(\bigcup_{j \in S} (i \cap s_j) \mid top\right) = \frac{p\left(\bigcup_{j: i \in s_j} s_j\right)}{p(top)}. \quad (131)$$

An approximation for this is

$$I^{VF}(i) \approx \frac{\sum_{j: i \in s_j} p(s_j)}{p(top)} \approx \frac{\sum_{j: i \in s_j} p(s_j)}{\sum_{s_j \in S} p(s_j)} \quad (132)$$

which is the same as the approximation in Theorem 19 of $p(X_k = 1 \mid top)$. The Fussell-Vesely importance is strongly emphasised by Balkey et al [3] to provide important information about a system with respect to component inspection policies. Here we will study implications of the Fussell-Vesely importance on inspection strategies for a failed system.

5.4.1 Traditional Inspection Strategy

Many authors (Balkey et al [3]) have suggested using the Fussell-Vesely importance measure to identify the failed minimal cut-set given the top-event. The traditional way of inspecting components is to inspect the most important components first.

EXAMPLE 10

Apply (131), calculate the Fussell-Vesely importance measures for the components in Figure 31 and arrange them in descending order give

$$I^{VF}(1) = .9545559492, I^{VF}(3) = .9543351109, I^{VF}(6) = .9088910146, \\ I^{VF}(5) = .0908890560, I^{VF}(4) = .0456717280, I^{VF}(2) = .0456695013.$$

Thus we have the test-sequence (1, 3, 6, 5, 4, 2). The expected number of inspections is the negative of

$$v(F \mid \emptyset \rightarrow [X_1 \rightarrow [X_3 \rightarrow [X_6 \rightarrow [X_5 \rightarrow [X_4 \rightarrow X_2]]]]]) \approx -2.95517$$

with the decision strategy in Figure 37.

Figure 37.

When comparing the result in Example 10 to Example 6 it is clear that basing the inspection strategy on the Fussell-Vesely importance measure is far from optimal. Indeed the Fussell-Vesely requires almost 50% more inspections on average. The Approaches shown in Example 7, Example 8 and Example 9 also shows that other ways of sequencing the inspections can do a lot better.

5.4.2 Wang's Algorithm

Xiaozhong & Cooke [6] argue that the optimal inspection strategy is to first inspect the component with Fussell-Vesely importance closest to 0.5 in absolute value. In Xiaozhong [5] a method based on a conditional Fussell-Vesely importance is used to choose which components to inspect. We call this method for Wang's algorithm. The conditional Fussell-Vesely importance is given by

$$I^{VF}(i | x_{\phi_1}, \dots, x_{\phi_k}) = p(\bigcup_{j:i \in s_j} s_j | top, x_{\phi_1}, \dots, x_{\phi_k}). \quad (133)$$

The inspection strategy is constructed by calculating

$$h(i | x_{\phi_1}, \dots, x_{\phi_k}) = |I^{VF}(i | x_{\phi_1}, \dots, x_{\phi_k}) - .5| \quad (134)$$

for $i \in K \setminus \{\phi_1, \dots, \phi_k\}$. Then one inspects the component that gives the smallest value in (134).

EXAMPLE 11

Applying Wang's algorithm above to find an inspection strategy of the fault-tree in Figure 31 gives

$$h(1) = .4545559492, h(2) = .4543304987, h(3) = .4543351109, \\ h(4) = .4543282720, h(5) = .4091109440, h(6) = .4088910146.$$

We can see that we should inspect Component 6 first.

$$\begin{aligned}h(1|X_6 = 1) &= .4997506236, h(2|X_6 = 1) = .4997488870, \\h(3|X_6 = 1) &= .4997539092, h(4|X_6 = 1) = .4997488747, \\h(5|X_6 = 1) &= .4995002480.\end{aligned}$$

We can see that given $X_6 = 1$ one should inspect Component 5.

$$\begin{aligned}h(1|X_6 = 1, X_5 = 1) &= .4762256420, h(2|X_6 = 1, X_5 = 1) = .4761779980, \\h(3|X_6 = 1, X_5 = 1) &= .4766544380, h(4|X_6 = 1, X_5 = 1) = .4761768307.\end{aligned}$$

We can see that given $X_6 = 1$ and $X_5 = 1$ one should inspect Component 4.

$$\begin{aligned}h(1|X_6 = 1, X_5 = 1, X_4 = 1) &= .4979511004, h(2|X_6 = 1, X_5 = 1, X_4 = 1) = \\&= .4999500024, h(3|X_6 = 1, X_5 = 1, X_4 = 1) = .4999510024.\end{aligned}$$

We can see that given $X_6 = 1, X_5 = 1$ and $X_4 = 1$ one should inspect Component 1.

$$\begin{aligned}h(2|X_6 = 1, X_5 = 1, X_4 = 1, X_1 = 1) &= .4755978526, \\h(3|X_6 = 1, X_5 = 1, X_4 = 1, X_1 = 1) &= .4760858956.\end{aligned}$$

We can see that given $X_6 = 1, X_5 = 1, X_4 = 1$ and $X_1 = 1$ one should inspect Component 2.

$$\begin{aligned}h(1|X_6 = 1, X_5 = 1, X_4 = 0) &= .5000000000, h(2|X_6 = 1, X_5 = 1, X_4 = 0) = \\&= .4755978526, h(3|X_6 = 1, X_5 = 1, X_4 = 0) = .4760858956.\end{aligned}$$

We can see that given $X_6 = 1, X_5 = 1$ and $X_4 = 0$ one should inspect Component 2.

$$\begin{aligned}h(1|X_6 = 1, X_5 = 0) &= .5000000000, h(2|X_6 = 1, X_5 = 0) = .4999987500, \\h(3|X_6 = 1, X_5 = 0) &= .4999987750, h(4|X_6 = 1, X_5 = 0) = .4999987500.\end{aligned}$$

We can see that given $X_6 = 1$ and $X_5 = 0$ one should inspect Component 4 or 2 and we will use Component 4.

$$\begin{aligned}h(1|X_6 = 1, X_5 = 0, X_4 = 1) &= .5000000000, h(2|X_6 = 1, X_5 = 0, X_4 = 1) = \\&= .4755978526, h(3|X_6 = 1, X_5 = 0, X_4 = 1) = .4760858956.\end{aligned}$$

We can see that given $X_6 = 1, X_5 = 0$ and $X_4 = 1$ one should inspect Component 2.

$$\begin{aligned}h(1|X_6 = 0) &= .0012346947, h(2|X_6 = 0) = .0012346947, h(3|X_6 = 0) = \\&= .0012341960, h(4|X_6 = 0) = .0012591343, h(5|X_6 = 0) = .4975311093.\end{aligned}$$

One can see that given $X_6 = 0$ one should inspect Component 3.

$$\begin{aligned}h(1|X_6 = 0, X_3 = 1) &= .4802969911, h(2|X_6 = 0, X_3 = 1) = .4802969911, \\h(4|X_6 = 0, X_3 = 1) &= .4803950209, h(5|X_6 = 0, X_3 = 1) = .4999029505.\end{aligned}$$

Given $X_6 = 0$ and $X_3 = 1$ one should inspect Component 1 or 2 and we will use Component 1.

$$\begin{aligned}h(2|X_6 = 0, X_3 = 1, X_1 = 1) &= .4095393248, h(4|X_6 = 0, X_3 = 1, X_1 = 1) = \\&= .4050140284, h(5|X_6 = 0, X_3 = 1, X_1 = 1) = .4955199566.\end{aligned}$$

Given $X_6 = 0, X_3 = 1$ and $X_1 = 1$ one should inspect Component 4.

$$\begin{aligned}h(2|X_6 = 0, X_3 = 1, X_1 = 1, X_4 = 1) &= .4523582658, \\h(5|X_6 = 0, X_3 = 1, X_1 = 1, X_4 = 1) &= .4528346832.\end{aligned}$$

Given $X_6 = 0, X_3 = 1, X_1 = 1$ and $X_4 = 1$ one should inspect Component 2.

Given $X_6 = 0$ and $X_3 = 0$ one should inspect Component 4. The inspection strategy is summarised in Figure 38.

Figure 38.

■

5.5 Conclusion

We have shown an example where by applying decision theory and value of information we can find a minimal cut-set in an expected number of inspections that is less than two existing methods based on Fussell-Vesely importance measure. We gave an alternative *value of information importance measure*.

The model gives an understanding of the computational complexity involved to find a good inspection strategy. The approximate method, Test-Schedule Sort, presented here gives the best result and has a computational complexity of $O(n^2[3^n-1])$ which is much better than the $O(n![3^n-1])$ time (see Theorem 14 in Chapter 4) it takes to do an exhaustive search covering all test-sequences.

It is my feeling that the complexity of the inspection problem that we have shown above is largely underestimated in literature and that the existing approaches that has been used not give a good understanding of the problem. We have shown that Value of Information gives a better insight to explain why the existing approaches are not optimal.

Note, one should even be able to do better, that is find an inspection strategy that gives a lower expected number of inspections, than the one we presented here. Our method where limited since it only considers fixed inspection sequences i.e. it does not re-sequence the tests using the results of previous inspections.

5.6 References

- [1] J. G. Norstrøm, R. M. Cooke, T. Bedford: "Statistical methods in design of safety critical software". In Lydersen, Hansen & Sandtorv (eds), *Safety and Reliability*, 1061-1068. Rotterdam: Balkema, 1998.
- [2] J. G. Norstrøm, R. M. Cooke, T. Bedford: "Value of Information based inspection strategy of a fault-tree". In Kafka & Schüeller (eds), *Safety and Reliability*, 621-626. Rotterdam: Balkema, 1999.
- [3] K. R. Balkey et al: "ASME Risk-Based Inservice Inspection and Testing: An Outlook to the Future". *Risk Analysis*: 18:407-421, 1998.
- [4] W. E. Vesely: "Reservations on Risk-Based Inservice Inspection and Testing: An Outlook to the Future". *Risk Analysis*: 18:423-425, 1998.
- [5] W. Xiaozhong: "Fault Tree Diagnosis Based on Shannon Entropy", *Reliability and System Safety*, 34:143-167, England: Elsevier, 1991.
- [6] W. Xiaozhong, R. M. Cooke: "Optimal inspection sequence in fault diagnosis", *Reliability and System Safety*, 37:207-210. England: Elsevier, 1991.
- [7] A. Høyland, M. Rausand: "System Reliability Theory, Models and Statistical Methods", Wiley, Series in Probability and Mathematical Statistics, 1994.
- [8] M. Spivak: "Calculus", W. A. Benjamin, 1967.
- [9] T. H. Cormen et al: "Introduction to Algorithms", London, MIT Press, 1990.

- [10] H. G. Tucker: "A Graduate Course in Probability", Academic Press Inc., New York, 1967.
- [11] Ada Decision Systems 1992: "DPL Advanced Version User Guide". 2710 Sand Hill Road, Menlo Park, CA94025, USA.

Chapter 6

Satellite Control-Software

We will design the control software to activate the deployment of an antenna boom for a satellite after it has been launched into orbit. Our main goal is to present an example where value of information is used to help design the control software architecture. We will use Figure 39 to describe the problem.

Figure 39. Rocket main-stage and cargo container.

On top of the Ariane 5 rocket [3] there is one cylindrical cargo container with an upper and a lower compartment. In each compartment there are two scientific satellites. Between Satellite 1 and 2 there are 5 break-wires. Similarly there are 5 break wires for satellites 3 and 4. Inside each satellite the same Separation Sequence control-software as shown in Figure 40 is running. We will now describe how a satellite is launched.

The rocket takes off from ground and the cargo container is on a trajectory to its destination in space. The solid booster stages (not shown in Figure 39 since they are dropped off at a lower altitude) and the rocket main stage, the Vulcain engine, carries the cargo container. Once the launcher has left the atmosphere, the upper compartment cover, the fairing, no longer serves any purpose and is jettisoned by means of a pyrotechnic device. Later the Vulcain engine is separated from the cargo container. The upper stage engine, the Aestus engine, is activated to propel the cargo container to its final destination. This typically takes 25-40 minutes depending on the launch window and mission specific goals.

The upper stage engine stops and the cargo container is in separation mode. Once in separation mode satellite 1 and 2 is jettisoned into space by means of a pyrotechnic

device between satellite 1 and 2 and the Lower compartment cover. Note that satellite 1 and 2 are still mounted together until another set of pyrotechnic devices separates satellite 1 and 2. The 5 break-wire sensors between Satellites 1 and 2 are broken and the Separation Sequence software in Figure 40 is required to start deployment of the upper antenna so ground contact can be established for Satellite 1 and Satellite 2. Note that each satellite is equipped with the same software and that the software in each Satellite runs without exchanging information with software in other Satellites. Once Satellites 1 and 2 have left, the Lower compartment cover no longer serves any purpose and is jettisoned by means of a pyrotechnic device. Then the two remaining Satellites 3 and 4 are separated in the same way as Satellites 1 and 2. Note that we are uncertain whether the separation is successful. For example assume that if Satellite 1 is stuck on top of Satellite 2. Then one can separate the Lower compartment cover (with Satellites 1 and 2) and safely try to separate Satellites 3 and 4.

The operation of a satellite is required to ensure mission success, which only can be achieved if ground can communicate with the launched satellites. Note that only a separated satellite with a deployed antenna can communicate.

We are uncertain about when the rocket engines and particularly the upper stage engine stops. This can be caused by uncertainties related to the launch window that is used as well as technical uncertainties that arise during launch. For example if there is a problem with the upper engine one should be able to start separation mode even if the cargo container has failed to reach its final destination. Problems could cause the upper stage engine to burn for longer or shorter periods causing the launch time to vary. If the engines did not give sufficient thrust it might be necessary to extend the launch time. On the other hand if the engine stops too early and the Cargo Container is in a too low orbit it is important that separation mode is entered to attempt to separate the satellites. In this case operation is often required sooner after separation than otherwise because the satellite is in the wrong orbit.

6.1 Detailed Control Software Description

In Norström [4], software to control the release mechanisms of the antenna booms of a satellite was analysed in detail. During launch the antenna booms on the top and bottom side of each satellite are stowed. In this configuration tele-command access to and telemetry from the satellite cannot be guaranteed as it depends on the actual distance and spacecraft altitude relative to the ground station. Sometimes a fixed antenna is mounted and provides lower hemisphere coverage.

Early command access, however, may be required in failure cases, e.g. wrong separation altitude with unfavourable solar aspect angle.

Therefore the application-software 'Separation Sequence' is introduced which initiates the antenna boom deployment after separation from launcher to ensure communication capability with the satellite (see Figure 40).

In addition the application software provides the capability of executing additional commands at pre-programmable times, to cope with potentially time critical requirements, e.g. thermal.

During the final pre-launch operations, some hours before launch, all parameters for the 'Separation Sequence' and 'Pyro Firing' program are loaded to protected-memory

and verified. Shortly before lift-off the program 'Separation Sequence' will be scheduled under specified operational modes on the computer.

We will now consider the software inside one of the satellites. The program waits for time t_L , which corresponds approximately to the launch duration. Then it continuously monitors the break wire status (for example the Separation Sequence software monitors the break-wires between Satellites 1 and 2) until separation is detected. Separation is assumed when 3 of 5 break wires are showing broken status. No time-out for the break wire monitoring is foreseen. So in any case the software will wait, until the separation is detected (i.e. 3 of 5 break wires are showing broken status).

Figure 40. The required control flow of the separation sequence program.

At a defined delay after detection of separation the separation sequence program performs the following actions:

- Send Cmd "Pyro Power On"
- Send Cmd "Arm Pyro"

- Schedule the "Pyro Firing" that fires the 'pyros' program.

The "Pyro Firing" program releases the upper antenna boom by firing the corresponding pyros and records the firing current at very high sampling rate.

The "Separation Sequence Program" checks the correct execution of the "Pyro Firing" program by its return status. In case of incorrect execution, the program returns to the cyclical monitoring of the separation status and after detection of separation it starts the whole sequence again.

After the antenna boom deployment, three free programmable command sequences stored in protected-memory, are executed. These sequences configure for example temperature management. In principle the following actions are to be performed:

- Heater/Dumper reconfiguration as required.
- Any other On/Off commands, as required.
- Establish ground contact, which includes the switch "On" of the appropriate transmitter/HPA and the change of computer mode at the expected begin of contact.
- Enable Time Tag command execution.

Note that these three free programmable sequences are not executed if less than 3 out of 5 break-wires show broken status. The command sequences are performed in the time range between 1 minute and 11 hours (the first orbit) after antenna deployment.

The timing and the command contents are free-programmable, because they depend on launch date and time and the final outcome of the thermal analysis.

In case of OBDH (On Board Data Handling) switch over to the redundant computer the "Separation" program will be restarted and it will properly continue execution, because the execution status and all necessary parameters are stored in protected memory.

6.1.1 Required Control-Software Decisions

In the required control-flow shown in Figure 40 the control software obtains information about the state of the satellite by verifying the pre-programmed launch time and monitoring the 5 break-wire sensors. It also monitors the current and voltage output when the pyrotechnic device is fired.

If more than 3 out of the 5 break-wires show broken and the launch time has passed then the control software decides that the satellite is separated and that the upper antenna boom should be deployed. Otherwise, the antenna boom will never be deployed by the control-software. At the first glance this seems like a good strategy. Let us however analyse what can go wrong by drawing the event tree in Figure 41 of the major incidents that can be caused by the deploy antenna mechanism. Note that we consider the software in only one satellite. Imagine that we for example consider Satellite 1 and the break-wires between Satellites 1 and 2.

Figure 41. Event tree illustration of the antenna deployment.

If the engine is running during launch and 3 out of 5 (3 oo 5) break wires show broken then the control software in Figure 40 will activate the deploy mechanism that can cause a loss of up to 4 satellites. If the engine has stopped, no separation and 3 oo 5 break-wires show broken then the control software will activate the deploy mechanism which can lead to the loss of 4 satellites. The break wire can be broken due to stress while the engines are running which can trick the software to activate the deploy mechanism too early. Note that the control software can only monitor the voltage of the break wire and does not see whether the break wire actually is broken. So if separation has occurred then the control software has a non-zero probability of not detecting the separation correctly via the break-wires since the wrong break wire status is shown which again might lead to the loss of a satellite. If the antenna is not deployed and we are separated we have a non-zero probability to retry the deploy mechanism from ground via a **small** fixed antenna that is a backup. However it should be noted that connection via the fixed antenna depends on satellite position and orbit and should therefore not be relied on.

If the satellite is separated without a successfully deployed antenna and the retry option via the fixed antenna is exhausted then it might be possible to retrieve a satellite with the shuttle. This is not shown in the figure since most satellites are not built for retrieval. Note that the space shuttle can not just pick up any satellite in space. The satellite must be designed for interaction with the shuttle.

6.2 Formal Decision Problem Formulation

We define "launch" of a satellite as the process, carried out by the Ariane 5, that takes the satellite to an orbit in space. The launch is characterised by the acts carried out by the launcher whose main function is to run the launcher engines. Launch is completed if all launcher engines have ceased to run (booster burn out time). When launcher

engines stop running they do not start again which we define as "launch finished". The destination orbit is uncertain since the launch can end in the wrong orbit depending on the performance of the rocket.

Separation occurs when the satellite ceases to be physically attached to other satellites or to the rocket. We define "separated" to be that the satellite is not physically attached to the rocket or other satellites.

In Figure 42 T_L is the time the launch is finished, T_S is the time separation occurs. In addition we use failure time T_F which is the time the satellite can not become operational after. Hence, define the state space S by

$$S = \{ (T_L, T_S, T_F) \mid T_L < T_S, T_S \leq T_F, (T_L, T_S, T_F) = (\text{engines stop at } T_L, \text{Separation at } T_S, \text{Failure at } T_F) \}.$$

Figure 42. The launch events schedule.

We will assume that $T_L = \tau_L + \varepsilon_L$ where $\varepsilon_L \sim N(0, \sigma_L^2)$ and τ_L is the pre-programmed expected time of launch termination. If $t < T_L$ then the launch is not finished. T_S is the time when the separation of the satellite takes place. In Figure 42 we can see that the satellite is separated if $t \geq T_S$ and not separated if $t < T_S$. We will assume that $T_S = \tau_L + \Delta_S + \varepsilon_S + \varepsilon_L$ where Δ_S is the time we wait after the rocket engine has stopped to the separation occurs and we use that ε_S has the distribution $\varepsilon_S \sim N(0, \sigma_S^2)$. Note, we required $T_L < T_S$ in S which gives

$$\varepsilon_S > -\Delta_S.$$

If $t \leq T_F$ the satellite is not failed. We will use $T_F = \tau_L + \Delta_S + \Delta_F + \varepsilon_S + \varepsilon_L + \varepsilon_F$ where Δ_F is the expected time we wait after separation to the satellite fail if we do not deploy the antenna. For simplicity we will assume that $\varepsilon_F \sim N(0, \sigma_F^2)$ mostly because it is easy to work with a joint normal distribution numerically. We have that

$$\varepsilon_F \geq -\Delta_F$$

since we required $T_S \leq T_F$ in S .

Note that we have chosen the normal distribution only for the purpose of illustration. The normal distribution however can be justified for the launch time since it represents launch time as an expected time of launch plus the uncertainty in time that the launch represents. Similarly we have used the Normal distribution to model the separation time. The major factors that influence the launch time uncertainty are rocket engine performance, weather conditions and the launch window. A study of these factors can be used to give a distribution of the launch time. To obtain the

separation time uncertainty we would consider the separation process physically and also look at real data from previous launches. For the failure time T_F we could however have used a survival distribution. We did also try to use the exponential distribution and found that it did not ensure a quick enough failure of a satellite with a non-deployed antenna to give a realistic example. Another distribution that we could use is the Weibull distribution, however the failure rate must quickly increase since a separated satellite without deployed antenna quickly fails. Appropriately chosen parameters in the Normal distribution that we have used will also ensure quick failure if the antenna is not deployed. Further it is important that the failure time is correlated to the separation time and launch time. For example a scenario where the launch ends too early will cause a failure much earlier than a launch which for some reason require the engines to run significantly longer. This is easy to model with a normal distribution.

We will now obtain the joint distribution of (t_L, t_S, t_F) given $\varepsilon_S \geq -\Delta_S$ and $\varepsilon_F \geq -\Delta_F$. Since we can not explain any direct linkage between the errors we will assume that ε_S , ε_L and ε_F are mutually independent. However, note although ε_S , ε_L and ε_F are mutually independent, the times T_L , T_S and T_F are not independent. The launch termination time T_L is uncertain. We discussed some reasons for this uncertainty in the introduction. If the launch duration lasts longer we would for example also expect that the separation occurs later so that the two properties are positively correlated. Similarly the separation time, T_S , should be positively correlated to the failure time T_F . In our notation we use $\Phi(x) = p(X \leq x)$ for the standard normal distribution.

To obtain the conditional joint distribution of (t_L, t_S, t_F) we shall first find the joint-distribution of

$$\mathbf{T} = \begin{bmatrix} T_L \\ T_S \\ T_F \\ \varepsilon_S \\ \varepsilon_F \end{bmatrix} = \begin{bmatrix} \tau_L + \varepsilon_L \\ \tau_L + \Delta_S + \Delta_F + \varepsilon_L + \varepsilon_S \\ \tau_L + \Delta_S + \Delta_F + \varepsilon_L + \varepsilon_S + \varepsilon_F \\ \varepsilon_S \\ \varepsilon_F \end{bmatrix}. \quad (135)$$

Since each of the variables are normally distributed the expectation is

$$\mathbf{E}(\mathbf{T}) = \begin{bmatrix} \tau_L \\ \tau_L + \Delta_S + \Delta_F \\ \tau_L + \Delta_S + \Delta_F \\ 0 \\ 0 \end{bmatrix}. \quad (136)$$

Using the mutually independence of ε_S , ε_L and ε_F we obtain the covariance matrix

$$\Sigma = \begin{bmatrix} \sigma_L^2 & & & & \\ \sigma_L^2 & \sigma_S^2 + \sigma_L^2 & & & \\ \sigma_L^2 & \sigma_S^2 + \sigma_L^2 & \sigma_S^2 + \sigma_L^2 + \sigma_F^2 & & \\ 0 & \sigma_S^2 & \sigma_S^2 & \sigma_S^2 & \\ 0 & 0 & \sigma_F^2 & 0 & \sigma_F^2 \end{bmatrix}. \quad (137)$$

Conditional on $\varepsilon_S > -\Delta_S$ and $\varepsilon_F \geq -\Delta_F$ the joint distribution of (t_L, t_S, t_F) is

$$f(t_L, t_S, t_F | \varepsilon_S > -\Delta_S, \varepsilon_F \geq -\Delta_F) = \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(\mathbf{T}-\mathbf{ET})^T \Sigma^{-1} (\mathbf{T}-\mathbf{ET})/2} d\varepsilon_S d\varepsilon_F, \quad (138)$$

$$(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))$$

where $\mathbf{T}$ is given by (135) above. Since we always are given $\varepsilon_S > -\Delta_S$ and $\varepsilon_F \geq -\Delta_F$, we will simply write $f(t_L, t_S, t_F)$ instead of $f(t_L, t_S, t_F | \varepsilon_S > -\Delta_S, \varepsilon_F \geq -\Delta_F)$.

6.2.1 Making observations

We will assume that we can use information from the onboard computer time and the break-wire test. The following three additional assumptions are made:

1. The break wire test $X(\tau)$ only shows the number of broken wires at the time we inspect, $\tau \geq 0$, and there is a total of five break wires so that $X(\tau) \in \{0, 1, \dots, 5\}$.
2. The onboard computer time is known.

We can in principle monitor the break wire status continuously at the cost of draining computer-resources like CPU time and memory. If we have already decided that memory is limited so we would not be able to store results from previous inspections we can however select the optimal time to inspect the break wires. If we decide that we inspect the break wires twice then we can optimise to find out when it is optimal to make those two observations. In the real system that was considered in Norström [4] no memory was used to store past results see Figure 40. We will now show how we can use value of information to redesign this control-software and find optimal deployment times given various test-outcomes.

ONE BREAK WIRE INSPECTION

The number of broken break wires is described by the random variable $X(\tau) \in \{0, 1, \dots, 5\}$. Since we have assumed that the software is without memory we do not know when a wire was broken when $X(\tau)$ is inspected at time τ . Thus at inspection time τ it is only possible to monitor how many break-wires show broken. Hence, the software is ignoring the fact that a particular wire was broken at a particular time during launch.

Conditional on the time we inspect, the break-wire test result will have different distributions. We will now define the probability distribution of the break-wire test given the four different inspection intervals (see Figure 42). Given $\tau < t_L$ or $t_L \leq \tau < t_S$ we assume that $X(\tau)$ has the binomial distribution

$$p(x(\tau) | \tau < t_L \cup t_L \leq \tau < t_S) = \binom{5}{x(\tau)} p(\tau)^{x(\tau)} (1 - p(\tau))^{5-x(\tau)}.$$

We use the binomial model to describe that the break wires are broken due to shock during launch and separation. The model assumes that each wire breaks independently with probability $p(\tau)$. The probability $p(\tau)$ is a function of the accumulated stress over time and is believed to increase as a function of τ . We will however for simplicity assume that $p(\tau)$ is constant and use $p = 0.05$.

Given separation we will assume that the break-wire test can return the wrong number of broken wires due to either bit flip or transient failures in the electronic circuits. The probability of bit flip or transient errors is assumed to be $\varepsilon = 0.001$. We will use the distribution

$$p(x(\tau) | t_S \leq \tau < t_F) = \begin{cases} \varepsilon & \text{for } x(\tau) \in \{0, 1, \dots, 4\} \\ 1 - 5\varepsilon & \text{otherwise.} \end{cases}$$

Given failure we will assume that the break-wires do not give relevant information which we will model by using the uniform distribution

$$p(x(\tau) | t_F < \tau) = 1/6, \text{ for } x(\tau) \in \{0, 1, \dots, 5\}.$$

From the conditional distributions one can obtain the joint distribution of the test result $x(\tau)$ and the vector (t_L, t_S, t_F) simply by multiplying with the distribution $f(t_L, t_S, t_F)$ in (138). This gives the joint distribution:

$$\begin{aligned} f(t_L, t_F, t_S, x(\tau)) &= f(t_L, t_F, t_S) \binom{5}{x(\tau)} p^{x(\tau)} (1 - p)^{5-x(\tau)} & \text{if } \tau < T_S \\ f(t_L, t_F, t_S, x(\tau)) &= f(t_L, t_F, t_S) \begin{cases} \varepsilon & \text{for } x(\tau) \in \{0, 1, \dots, 4\} \\ 1 - 5\varepsilon & \text{otherwise} \end{cases} & \text{if } T_S \leq \tau < T_F \quad (139) \\ f(t_L, t_F, t_S, x(\tau)) &= f(t_L, t_F, t_S) 1/6 & \text{if } T_F \leq \tau \end{aligned}$$

6.2.2 The Set of Available Acts

The main goal of the control software is to prevent the antenna boom being deployed inside the rocket and to deploy the antenna boom when the satellite is separated. The boom is required for communication between the satellite and the ground station. Therefore the deployed antenna is necessary when contact with the satellite is required.

We will now formulate the decision to deploy the antenna boom as a decision theoretic problem. Our acts for all $t > 0$ are

$$f(t) = \text{Activate deploy antenna boom mechanism at time } t.$$

We will now compare these acts to the control flow diagram in Figure 40. $f(t)$ is the act that the control-software in Figure 40 takes when at least 3 break-wires consistently show broken status after it has waited for the launch time to end. If less than 3 of 5 break-wires consistently show broken the antennas will never be deployed automatically. There is a certain risk that the 3 out of 5 wires never will show broken

status even if the satellite is separated. To never deploy the antenna can depending on the mission be the same as mission failure. Hence, we can see that the acts $F = \{f(t) | t > 0\}$ are the same decisions used in the control-software in Figure 40. Note that $t = \infty$ is the act to never deploy the antenna which the current architecture decides if less than 3 out of 5 wires show broken.

6.2.3 Consequence Assessment & Utility Definition

We will briefly define utility over the consequences over $S \times F$ and describe the consequences and allocate values. Here we have made estimates of the costs involved based on the authors reasoning. The numbers are only example of figures to illustrate the optimisation problem. Estimating the real costs would require detailed studies of the consequences involved.

	$t < T_L$	$T_L \leq t < T_S$	$T_S \leq t < T_F$	$T_F \leq t$
$f(t)$	$-500 \cdot 10^6$	$-250 \cdot 10^6$	0	$-125 \cdot 10^6$

Table 1: The utility over $F(t) \times S$.

The reasoning behind here is that one satellite costs 125 000 000 \$. If we deploy during launch we risk losing 4 satellites by attempting to deploy the antenna. Therefore $U(f(t), t < T_L) = -4 \cdot 125 \cdot 10^6$. Similar after the launch ends before separation we risk losing 2 satellites by deploying the antenna boom. Therefore we use $U(f(t), T_L \leq t < T_S) = -2 \cdot 125 \cdot 10^6$. If $T_F \leq t$ we loose the satellite anyway. Therefore we use $U(f(t), T_F \leq t) = -125 \cdot 10^6$.

6.2.4 Parameter Specification

We will now specify the parameters in the model. This is a field where for example expert judgement can be considered. We will however use some typical values for the Ariane 5 launcher fit the model so it matches this information. The expected launch time is

$$\tau_L = 26.49 \text{ min.}$$

We will assume that the variance of the parameter ϵ_L is

$$\text{Var}(\epsilon_L) = \sigma_L^2 = (8 \text{ min})^2.$$

The expected time for separation to be finished is $\tau_S = 33.30 \text{ min}$. This gives the expected time between launch finished and separation as

$$\Delta_S = 33.30 \text{ min} - 26.49 \text{ min} = 6.81 \text{ min.}$$

The variance of the parameter ϵ_S is which is the uncertainty of the separation has to cover that satellites 1 and 2 are separated at 28.10 min and satellites 3 and 4 at 33.30 which suggest that it is at least $(5.20 \text{ min})^2$. Hence, we will use

$$\text{Var}(\epsilon_S) = \sigma_S^2 = (6 \text{ min})^2.$$

The expected time from separation to failure is

$$\Delta_F = 7.70 \text{ min.}$$

Since this time is rather uncertain we will use a rather large variance

$$\text{Var}(\varepsilon_F) = \sigma_F^2 = (10 \text{ min})^2.$$

6.2.5 The Value of The Break Wire Test

We will now find the optimal deployment time if no inspection is carried out. The expected utility if no inspection is carried out is given by

$$\begin{aligned} v(f(t)) = & (-500 \cdot 10^6 p(t < T_L) - 250 \cdot 10^6 p(T_L \leq t < T_S) \\ & - 0 p(T_S \leq t < T_F) - 125 \cdot 10^6 p(T_F \leq t)). \end{aligned} \quad (140)$$

To obtain the relevant probabilities from our probability density in (138) is just to integrate over the respective regions by using the method in Genz [2]. The optimal act is obtained by taking the best of all possible acts. This gives

$$v(F) = \max_{t \geq 0} v(f(t)). \quad (141)$$

REMARK: In the following we use time in minutes if nothing else is specified.

Figure 43. The expected value $v(f(t))$ as function of time t .

Figure 43 indicates that the optimal deployment time is 48 minutes with an expected utility of $-90 \cdot 10^6$. Optimisation with 3 significant digits gives

$$v(F) = v(f(t = 48.3)) = -94.8 \cdot 10^6 \text{ \$}. \quad (142)$$

In general we have that $v(F|X(\tau)) \geq v(F)$. We will now obtain the conditional expectation if we make one observation at test time τ . Note that since we must observe the break-wire before we can take the conditional expected value we have

$$v(f(t) | \text{Breakwires}) = \begin{cases} v(f(t) | X(t)) & \text{if } t \geq \tau \\ v(f(t)) & \text{otherwise.} \end{cases}$$

So for $t \geq \tau$ we need to compute the expected value

$$v(f(t) | x(\tau)) = (-500 \cdot 10^6 p(t < T_L | x(\tau)) - 250 \cdot 10^6 p(T_L \leq t < T_S | x(\tau)) - 0 p(T_S \leq t < T_F | x(\tau)) - 125 \cdot 10^6 p(T_F \leq t | x(\tau))). \quad (143)$$

The probabilities are obtained from the distribution in (139). This is however a rather complex process which we discuss in Section 6.2.7. Optimisation over all accessible acts gives

$$v(F | x(\tau)) = \max_{t > 0} v(f(t) | x(\tau)).$$

For each test outcome this optimisation produces various optimal deployment times if we inspect the wires at test time τ . The conditional expectation is given by taking the expectation over the test-outcomes that gives

$$v(F | X(\tau)) = \sum_{x(\tau)} p(x(\tau)) \max_{t > 0} v(f(t) | x(\tau)). \quad (144)$$

The hardest problem is to find the optimal test time.

$$\tau = \operatorname{argmax}_{\tau > 0} v(F | X(\tau)).$$

A rough overview by limiting the search between 20 minutes and 55 minutes produces the graph in Figure 44.

Figure 44. The conditional expected utility $v(F | X(\tau))$ as function of test time τ .

From Figure 44 we see that the optimal inspection time is between 37 and 38 minutes. An optimisation with 3 significant digits gives.

$$\max_{\tau > 0} v(F | X(\tau)) = v(F | X(\tau = 37.5)) = -59.9 \cdot 10^6 \$.$$

If we compare the expected value in Figure 44 to the one in Figure 43 we can see that we are better off by inspecting. The value of information is

$$v(F | X(\tau = 37.5)) - v(F) = 59.9 \cdot 10^6 \$ - 94.8 \cdot 10^6 \$ = 34.9 \cdot 10^6 \$ \quad (145)$$

6.2.6 The Optimal Control-Flow

We will now look at how the architecture of the control software should implement the break wire inspection. We will inspect at the optimal test-time of 37.5 minutes. For each inspection outcome $x(\tau) \in \{0, 1, \dots, 5\}$ we will compute the optimal deployment time

$$t(x(\tau = 37.5)) = \operatorname{argmax}_{t > 0} v(f(t) | x(\tau = 37.5)).$$

Figure 45. The conditional expected utility for the various test-outcomes.

From Figure 45 we can for example see that if we observe at test-time 37.5 minutes that $x(\tau = 37.5) = 0$ break wires are broken, we should deploy the antenna at 52 minutes. Optimisation with 3 significant digits gives

$$t(x(\tau = 37.5) = 0) = 51.9 \quad (146)$$

$$t(x(\tau = 37.5) = 1) = 51.9 \quad (147)$$

$$t(x(\tau = 37.5) = 2) = 51.7 \quad (148)$$

$$t(x(\tau = 37.5) = 3) = 48.7 \quad (149)$$

$$t(x(\tau = 37.5) = 4) = 37.5 \quad (150)$$

$$t(x(\tau = 37.5) = 5) = 37.5 \quad (151)$$

In conclusion, if we have one inspection of the break wires the control software architecture should be designed so that we inspect at 37.5 minutes with the conditional deployment times given in (146) to (151). This gives the maximal value of information that can be obtained in (145) when the break wires are inspected once.

6.2.7 Working Out the Probabilities

We will show how we can obtain the probabilities in (143). Note that it is possible to simplify some of the probabilities below since some of the integrals are integrating from minus infinity to infinity. We will not show how that can be done here since the difficult task is to get the integration areas correct. This is what we show below.

First we will consider the probabilities for times greater than the inspection time, $t \geq \tau$, such that the test-outcome is known before we decide. Hence, we use the joint distribution in (139) to obtain the conditional distribution $p(t < T_L | x(\tau))$:

$$\begin{aligned} p(t < T_L | x(\tau)) &\propto \int_{t_L=t}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\infty} f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \\ &= \left(\frac{5}{x(\tau)} \right) p^{x(\tau)} (1-p)^{5-x(\tau)} \int_{t_L=t}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\infty} f(t_L, t_S, t_F) dt_F dt_S dt_L. \end{aligned}$$

We now use (138) to substitute $f(t_L, t_S, t_F)$ which gives

$$\begin{aligned} p(t < T_L | x(\tau)) &\propto \left(\frac{5}{x(\tau)} \right) p^{x(\tau)} (1-p)^{5-x(\tau)} \\ &\quad \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=t}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L \\ &\quad \cdot \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))}. \end{aligned}$$

We recognise the numerator as the 5 dimensional multivariate normal distribution which we can easily solve numerically by using Genz's [2] method. Similarly we obtain the probability $p(T_L \leq t < T_S | x(\tau))$:

$$\begin{aligned}
p(T_L \leq t < T_S | x(\tau)) &\propto \int_{t_L=-\infty}^t \int_{t_S=t}^{\infty} \int_{t_F=-\infty}^{\infty} f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \\
&= \binom{5}{x(\tau)} p^{(x(\tau))} (1-p)^{5-x(\tau)} \\
&= \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^t \int_{t_S=t}^{\infty} \int_{t_F=-\infty}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L \\
&\quad \cdot \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))}.
\end{aligned}$$

To obtain $p(T_S \leq t < T_F | x(\tau))$ requires more refinement. Using that the distribution in (139) is differently specified for different intervals, Bayes' Theorem gives

$$p(T_S \leq t < T_F | x(\tau)) \propto \left[\begin{aligned} &\int_{t_L=-\infty}^{\infty} \int_{t_S=\tau}^t \int_{t_F=t}^{\infty} f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \\ &+ \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\tau} \int_{t_F=t}^{\infty} f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \end{aligned} \right].$$

We now use (139) and (138) to substitute $f(t_L, t_S, t_F, x(\tau))$ which gives

$$\begin{aligned}
p(T_S \leq t < T_F | x(\tau)) &\propto \binom{5}{x(\tau)} p^{(x(\tau))} (1-p)^{5-x(\tau)} \\
&= \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=\tau}^t \int_{t_F=t}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L \\
&\quad \cdot \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))} \\
&+ \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\tau} \int_{t_F=t}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L \\
&\quad \cdot \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))} \\
&\cdot \begin{cases} \epsilon & \text{for } x(\tau) \in \{0, 1, \dots, 4\} \\ 1 - 5\epsilon & \text{otherwise} \end{cases}.
\end{aligned}$$

Similar splitting is also necessary for the last probability $p(T_F \leq t | x(\tau))$. Using Bayes' Theorem and splitting the integral according to the distribution in (139) gives

$$p(T_F \leq t | x(\tau)) \propto \left[\begin{aligned} & \int_{t_L=-\infty}^{\infty} \int_{t_S=\tau}^{\infty} \int_{t_F=\tau}^t f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \\ & + \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\tau} \int_{t_F=\tau}^t f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \\ & + \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\tau} f(t_L, t_S, t_F, x(\tau)) dt_F dt_S dt_L \end{aligned} \right].$$

Again we use (139) and (138) to substitute $f(t_L, t_S, t_F, x(\tau))$ which gives

$$p(T_F \leq t | x(\tau)) \propto$$

$$\binom{5}{x(\tau)} p^{x(\tau)} (1-p)^{5-x(\tau)}$$

$$\frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=\tau}^{\infty} \int_{t_F=\tau}^t \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L$$

$$(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))$$

$$+ \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\tau} \int_{t_F=\tau}^t \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L$$

$$(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))$$

$$\begin{cases} \epsilon \text{ for } x(\tau) \in \{0, 1, \dots, 4\} \\ 1 - 5\epsilon \text{ otherwise} \end{cases}$$

$$+ \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\tau} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L$$

$$+ 1/6 \cdot \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L$$

$$(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))$$

Having worked out the probabilities for $t \geq \tau$ above we will now consider the interval where we deploy at time before we observe, $t < \tau$, and since we are not able to take the conditional distribution we use $f(t_L, t_F, t_S)$. The reason for this is that we do not now the test outcome before test-time τ so if we decide to deploy any time before we have inspected the break-wires we can only use the "prior" probabilities. Hence, it is sufficient to obtain the probabilities in (143) by only considering the distribution in (138). Note that these probabilities are the same as the ones we need to solve (140). We will start by finding $p(t < T_L)$.

$$p(t < T_L) = \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=t}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L = \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))}$$

The second probability is

$$p(T_L \leq t < T_S) = \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^t \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L = \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))}$$

The third probability is

$$p(T_S \leq t < T_F) = \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^t \int_{t_F=-\infty}^{\infty} \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L = \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))}$$

The last probability is

$$p(T_F \leq t) = \frac{1}{(2\pi)^{5/2} \sqrt{|\Sigma|}} \int_{t_L=-\infty}^{\infty} \int_{t_S=-\infty}^{\infty} \int_{t_F=-\infty}^t \int_{-\Delta_F}^{\infty} \int_{-\Delta_S}^{\infty} e^{-(T-ET)^T \Sigma^{-1} (T-ET)/2} d\epsilon_S d\epsilon_F dt_F dt_S dt_L = \frac{1}{(1 - \Phi(-\frac{\Delta_S}{\sigma_S}))(1 - \Phi(-\frac{\Delta_F}{\sigma_F}))}$$

6.3 References

- [1] L. Savage: "The Foundations of Statistics". Dover, New York, 1972.
- [2] A. Genz: "Numerical Computation of Multivariate Normal Probabilities", J. of Computational and Graphical Stat., 1, pp. 141-149, 1992.
- [3] ESA Public Relations Division Offices: "Ariane 5 Architecture", <http://www.esrin.esa.it/esa/ariane/archi.html>

- [4] J. G. Norstrøm: "Sneak Path Analysis for Computer Controlled Systems", ESTEC Working Paper No: EWP 1871, December 1995.
- [5] B. Dore, J.G. Norstrøm: "Pilot Application of Sneak Analysis on Computer Controlled Satellite Equipment", In P.C Cacciabue & I. A. Papazoglou (eds), "Probabilistic Safety Assesment and Management", Springer, 24-28 June, 1996, pp 1590-1596.
- [6] Ada Decision Systems 1992: "DPL Advanced Version User Guide". 2710 Sand Hill Road, Menlo Park, CA94025, USA.

Index

A

architecture, 54, 209

B

binary tree, 105
break wire monitoring, 199

C

conditional value of information importance, 186

D

decision logic, 106
decision logic map, 107
decision theory, 74
Decision trees, 82

E

ENCRESS, 5
Error, 9
exchangeability, 118

F

Failure, 9
Fault, 9
fault-tree, 154
field, 65
Fussel-Vesely, 190

G

Greedy-Sequence before Test, 129, 182
Greedy-Test before Sequence, 131, 185

I

influence diagrams, 82

L

launch finished, 202

M

Markov and Semi-Markov, 46
measurable space, 66

O

Optimal stopping problems, 117

P

partial ordering " $\geq$ " on F, 75
performance costs, 88
principle of definition, 76
principle of dominance, 76
probability space, 66

Q

qualitative probability, 77
Quantitative Probability, 78

R

random variable, 66
Rational Preference, 80
refinement principle, 77

S

Self-Conditional Expectation, 68
self-conditional statements, 67
separated, 202
Separation Sequence, 199
Set-up cost, 87
severity classification system, 10
sigma field, 66
software characteristics, 14
Software reliability, 10
Strengthened refinement principle, 79
Sub-additivity, 72
sure thing principle, 75

T

test schedule, 107
test-schedule, 107
Test-Schedule Addition, 110
Test-Schedule Sort, 188
Test-Schedule Subtraction, 110
test-schedule-partition, 103
test-sequence, 105
Total quality management, 16

U

utility, 79, 89
utility-associative, 121

utility-commutative, 123

V

value of information importance, 183

W

weak order axiom, 75

Samenvatting

'Sneak path' analyse was mijn eerste kennismaking met de betrouwbaarheid van software. Het wordt gebruikt om ontwerpfouten in software en hardware te vinden. De methode tracht alle paden te localiseren tussen essentiële functies en de commando's die ze in werking stellen. Wanneer alle paden zijn vastgesteld, worden ze vergeleken met de specificaties van de systeemvereisten. Overal waar deze niet overeenkomen wordt onderzocht welk effect dit heeft op het systeem waar het de veiligheid, betrouwbaarheid en prestatie van het systeem betreft. De methode geeft de gebruiker geen mogelijkheden om vraagtekens te zetten bij de onderliggende architectuur of bij de specificaties. Het is niet mogelijk om de methode te gebruiken om verschillende ontwerp-oplossingen te onderzoeken. De methode maakt het niet mogelijk om de vraag te stellen of het stroomschema zelf wel goed is ontworpen. Sterker nog, geen enkele standaardmethode voor het ontwerpen van software heeft instrumenten die de ontwerper in staat stellen om een optimale ordening te bepalen voor de sensor-inspecties. Aan het begin van het promotieonderzoek werd besloten dat ik zou werken in het veld waar technieken uit de besliskunde worden gebruikt voor het ontwerpen van controle-software.

Allereerst werd de literatuur over de betrouwbaarheid van software onderzocht om te zien of deze informatie bevatte over de evaluatie van verschillende ontwerp mogelijkheden in de context van het controleren en ordenen van sensors; deze literatuur is kort samengevat in hoofdstuk 2. Omdat in het bedrijfsleven de betrouwbaarheid en veiligheid van software vaak worden gerekend tot de kwaliteitsbeheersing voor deze software was dit een logisch beginpunt voor onderzoek. Ik ontdekte dat softwarebetrouwbaarheidsmodellen over het algemeen de software weergeven als een zgn. 'black box'. De literatuur richt zich voornamelijk op de concepten van het verminderen van het percentage fouten of het verminderen van de waarschijnlijkheid van fouten; de gegevens over fouten worden verkregen uit het 'debugging' proces of uit eigen ervaring. De prestatie van de software wordt beter omdat fouten na ontdekking verwijderd worden. De vereenvoudiging van software als een black box impliceert dat geen van de methoden de onderliggende architectuur van de software in twijfel trekken. Daarom kunnen deze modellen niet verklaren hoe controle-software observaties zou kunnen doen of sensoren zou moeten lezen. De modellen kunnen wel worden gebruikt om te bepalen wanneer het tijd is om een 'nieuw' software programma op de markt te brengen. Hoofdstuk 2 gaat niet verder dan de bestaande literatuur en is geschreven om inzicht te krijgen in software-betrouwbaarheid. Het constateert dat de architectuur van controle-software nog niet is onderzocht in de context van de betrouwbaarheid van software.

Om een wiskundige basis te ontwikkelen voor verder onderzoek bestudeerde ik de beslissingstheorie van Savage als achtergrondinformatie. Zijn theorie formaliseert het idee van de rationele beleidsvormer, welke kort is samengevat in hoofdstuk 3. Tijdens het onderzoek koos ik twee gebieden voor verder onderzoek. Het eerste heeft te maken met het gebruik van invloedsdiagrammen en beslissbomen. Het tweede beschrijft hoe sensoren probabilistische informatie zouden moeten verzamelen. We constateerden dat een invloedsdiagram een goede manier is om symmetrische

besluitbomen te modelleren. De grafische weergave was echter niet praktisch en modelleerde het probleem van architectuuroptimalisatie niet goed. De reden hiervoor is dat het niet mogelijk is om het effect van een verandering van de volgorde van tests/observaties te modelleren zonder het invloedsdiagram en de besluitboom opnieuw te tekenen. Dit onderzoek leverde ideeën en een goede basis voor het verdere onderzoek in hoofdstuk 4. Het andere doel was om te begrijpen hoe sensoren zouden moeten worden ontworpen om probabilistische informatie te krijgen. 'Self conditional expectations' werden bestudeerd om de waarschijnlijkheid van een gebeurtenis A te vinden op basis van de uitkomst van een willekeurige variabele T , $p(A | T = t)$, om uit te vinden of er manieren zijn om een sensor zodanig te ontwerpen dat de verwachting van de voorwaardelijke waarschijnlijkheid verandert. Dit is origineel onderzoek en sommige uitkomsten kunnen gebruikt worden als argumenten om te beweren dat sensoren zouden moeten worden ontworpen om probabilistische informatie over een fijnere onderverdeling van gebeurtenissen te krijgen. Sterker nog, probabilistische overwegingen van controle-software zouden dit soort voorwaardelijke waarschijnlijkheid moeten omvatten.

Het voornaamste doel van hoofdstuk 4 is het vinden van een manier uit de beslissingstheorie voor het weergeven van de keuze die de software heeft wanneer het moet kiezen om een observatie 'te doen' of 'niet te doen'. Een nieuwe notatie is geïntroduceerd om duidelijk de mogelijke keuzes in het plannen van tests te laten zien. Een simpel voorbeeld van een test schema is " $A \rightarrow B$ ", wat betekent "doe A en kies dan of je B doet of niet". Voor optimalisatiedoelinden was het belangrijk dat het model een beredeneerde keuze van tests mogelijk maakte op basis van het verwachte nut. Toen het begrip 'testschema' eenmaal was geïntroduceerd, volgde een groot aantal nieuwe stellingen. De stellingen en hun bewijzen kunnen niet in andere literatuur worden gevonden; ze zijn nieuw. Het werd aangetoond dat de volgorde van tests belangrijk is, in andere woorden: $A \rightarrow B$ levert een andere verwachte nut op dan $B \rightarrow A$. Daarom hangt de verwachte nut af van het beslisprobleem, het testschema en de volgorde van de tests binnen het schema. Op basis van het eerste onderzoek naar invloedsdiagrammen en beslisbomen in hoofdstuk 3 werd een nieuwe, compacte grafische weergave van een keuzeboom gebouwd om de keuze om een test 'te doen' of 'niet te doen' te laten zien. Onze theorie is verwant aan de 'optimal stopping' theorie van De Groot. Het verband is echter niet onmiddellijk duidelijk. De 'optimal stopping' theorie gebruikt statistische aannames betreffende de voorwaardelijke onafhankelijkheid van willekeurige variabelen, de tests, op basis van een eerdere distributie. Hij veronderstelt ook dat het evenveel kost om iedere test te doen. Dit zijn zware wiskundige aannames. Onder deze aannames wordt bewezen dat ons probleem vereenvoudigd kan worden tot de problemen van 'optimal stopping' in de literatuur.

De bevestiging in hoofdstuk 4 dat de verwachte gebruikswaarde afhangt van de volgorde van de tests vereiste onderzoek naar heuristische methoden om een optimale volgorde van tests in een testschema te vinden. In hoofdstuk 5 passen we de theorie van hoofdstuk 4 toe in een andere omgeving, namelijk op het probleem van het vinden van algorithmes die fouten opsporen. De Fussell-Vesely importantiemaat wordt vaak aangeraden voor het rangschikken van componenten naar inspectieprioriteit in foutopsporingsalgorithmen. Fussell-Vesely is bestudeerd naast andere heuristische concepten, zoals "Greedy", beschreven in hoofdstuk 4. Het primaire doel was ontdekken hoe goed heuristische strategieën werken. Onze bevindingen laten zien dat deze strategieën slecht presteren in het voorspellen van het verwachte aantal tests die nodig zijn om de oorzaak van de fout op te sporen. De uitkomst is tot 50% slechter

dan de beste test volgorde verkregen door middel van een 'exhaustive search'. Hierom kan men concluderen dat het niet goed is om het gebruik van de Fussel-Vesely importantiemaat voor het opsporen van fouten aan te bevelen. Omdat de resultaten zoveel verschillen probeerden we er achter te komen waarom heuristische benaderingen zoals de Fussel-Vesely en "Greedy" zulke slechte resultaten opleverden. Het was echter niet mogelijk om een simpele verklaring te vinden, en het was makkelijker om onderzoek te doen naar een nieuwe methode. Daarom ontwikkelden we een algoritme voor het sorteren van testschema's om het aantal verwachte vereiste onderzoeken om de fout te vinden te reduceren. In de bestudeerde voorbeelden verbeterde de testschemasortering de volgorde van de tests snel zodat de hoeveelheid verwachte inspecties dicht bij het optimum kwam te liggen.

Hoofdstuk 6 keert terug naar het probleem waarmee dit onderzoek begon. We bestuderen een klein systeem van satellietcontrole-software en laten zien hoe de architectuur van software kan worden verbeterd door het gebruik van besliskunde. Omdat het model dat in hoofdstuk 4 is ontwikkeld succesvol kon worden gebruikt in hoofdstuk 5 moesten we aantonen dat besliskunde praktisch kan worden toegepast in het ontwerpen van control software architectuur. Dit idee was nieuw toen ons onderzoek naar besliskunde begon. In het ontwerpen van praktische software en in het bijzonder in het ontwerpen van controle-software is het gebruik van de zgn. 'voting logic' een populaire manier om beslissingen te nemen waarin onzekerheid een rol speelt. De controle-software van satellieten kan bijvoorbeeld besluiten een antennedragers op te stellen wanneer 3 van de 5 'break wires' gebroken blijken te zijn. Deze heuristische methode is echter niet de optimale manier om te beslissen bij onzekerheid, en kan software-ontwerpen opleveren die verre van optimaal zijn. Nadat we besliskunde hadden gebruikt voor een vereenvoudigd vraagstuk vonden we de optimale tijd om een sensor af te lezen. De aangegeven waarde bepaalt nu wat de optimale tijd is om de antennedragers op te stellen. Deze informatie kan direct worden gebruikt wanneer de controle-software wordt ontworpen en worden uitgedrukt in een stroomschema.

Curriculum Vitae

Jan Gerhard Norstrøm was born on 2 September 1970 in Tønsberg, Norway and grew up on the beautiful island of Tjøme in the Oslo fjord. He finished the Grammar School, specialising in scientific subjects, at Nøtterøy in 1989. In 1994 he graduated from the Norwegian University of Science and Technology (NTNU) and received the degree *Sivilingeniør* (equivalent to M.Sc. in Engineering). His final thesis was carried out at the Norsk Hydro Research Centre in Porsgrunn, and results were published in *IEEE Transactions on Reliability*. Shortly after his graduation he moved to the Netherlands to work as Young Graduate Trainee at the European Space Research and Technology Centre (ESTEC) in Noordwijk. At ESTEC he worked on the problem of evaluating satellite control-software. This laid the foundations for his research on decision theory and value of information, which is presented in this doctoral thesis. His research was carried out in the Department of Control risk, Optimisation, Stochastics and System Theory (CROSS) of the Delft University of Technology where he worked under supervision of Prof. dr. R.M. Cooke and Dr. T.J. Bedford. The work was carried out over the period from January 1996 to January 2000.