

**Right on Time:  
Measuring, Modelling and Managing  
Time-Constrained Software Development**

**Antony Lee Powell**

Submitted for the degree of Doctor of Philosophy

University of York  
Department of Computer Science

17<sup>th</sup> August 2001

## Abstract

Commercial pressures on time-to-market often require the development of software in situations where deadlines are very tight and non-negotiable. This type of development can be termed '*time-constrained software development*.'

The need to compress development timescales influences both the software process and the way it is managed. Conventional approaches to measurement, modelling and management, treat the development process as being linear, sequential and static. Whereas, the processes used to achieve timescale compression in industry are iterative, concurrent and dynamic. That is, they replace the notion of '*right-first-time*' with one of '*right-on-time*.'

In response to these problems, we propose new techniques to aid the management of time-constrained development. We start by introducing a new measurement technique, called the Process Engineering Language (PEL), to capture important elements of a process that are obscured by conventional approaches to data collection. These fine-grained measures are essential to our observations of industrial process behaviour within Rolls-Royce and BAE SYSTEMS, and our isolation of three specific problems of time-constrained development: overloading, bow-waves and the multiplier effect. In response, we propose a new modelling technique, called Capacity-Based Scheduling (CBS), to control risk across a portfolio of time-constrained projects. Finally, we use industrial data from Rolls-Royce to evaluate the validity and utility of our modelling approach and propose new strategies for the management of time-constrained software development.

These measurement, modelling and management techniques help to bridge the gap between academic research and industrial practice. More pragmatically, we have developed and deployed methods and tools that have proved of value to managers responsible for developing software '*right-on-time*.'

# Contents

<b>CHAPTER 1: INTRODUCTION.....</b>	<b>14</b>
1.1 BACKGROUND.....	14
1.2 TIME-CONSTRAINED SOFTWARE DEVELOPMENT .....	15
1.3 AN EXAMPLE OF TIME-CONSTRAINED DEVELOPMENT .....	17
1.4 RESEARCH AIMS .....	18
1.5 THESIS STRUCTURE.....	19
<b>CHAPTER 2: LITERATURE REVIEW.....</b>	<b>21</b>
2.1 INTRODUCTION .....	21
2.2 SOFTWARE CHANGE.....	21
2.3 MEASURING SOFTWARE CHANGE .....	24
2.3.1 <i>The History of Software Measurement</i> .....	24
2.3.2 <i>Measurement Theory and Frameworks</i> .....	26
2.3.3 <i>Change Counts and Causal Analysis</i> .....	28
2.3.4 <i>Statistical Defect Models</i> .....	29
2.3.5 <i>Volatility, Completeness and Risk</i> .....	31
2.3.6 <i>Error-Proneness</i> .....	32
2.3.7 <i>The Ripple Effect</i> .....	33
2.3.8 <i>Conclusions on the Measurement of Change</i> .....	34
2.4 MODELLING SOFTWARE CHANGE .....	35
2.4.1 <i>Modelling Cost</i> .....	35
2.4.2 <i>Modelling Processes</i> .....	38
2.4.3 <i>Modelling Change</i> .....	40
2.5 MANAGING SOFTWARE CHANGE .....	44
2.6 SUMMARY OF RESEARCH ISSUES .....	46
<b>CHAPTER 3: MEASURING TIME-CONSTRAINED DEVELOPMENT.....</b>	<b>48</b>
3.1 INTRODUCTION .....	48
3.2 WORK BREAKDOWN STRUCTURES (WBS) AND THEIR PROBLEMS.....	48
3.3 THE PROCESS ENGINEERING LANGUAGE (PEL).....	51
3.3.1 <i>Background to PEL</i> .....	51

3.3.2	<i>Description of PEL</i> .....	52
3.4	THE MEASUREMENT PROGRAMME.....	55
3.5	MEASURING EFFORT USING PEL.....	57
3.5.1	<i>Collection of PEL Effort Data</i> .....	57
3.5.2	<i>Analysis of PEL Effort Metrics</i> .....	60
3.5.3	<i>Benefits of PEL Effort Measurement</i> .....	65
3.6	MEASURING DEFECTS USING PEL.....	67
3.6.1	<i>Collection of PEL Defect Data</i> .....	67
3.6.2	<i>Analysis of PEL Defect Metrics</i> .....	72
3.6.3	<i>Benefits of PEL Defect Measurement</i> .....	74
3.7	MEASURING SIZE USING PEL.....	76
3.7.1	<i>Collection of PEL Size Data</i> .....	76
3.7.2	<i>Analysis of PEL Size Data</i> .....	78
3.7.3	<i>Benefits of PEL Size Measurement</i> .....	79
3.8	SUMMARY OF MEASURING TIME-CONSTRAINED DEVELOPMENT .....	80
<b>CHAPTER 4: OBSERVATIONS ON TIME-CONSTRAINED DEVELOPMENT .....</b>		<b>82</b>
4.1	INTRODUCTION .....	82
4.2	ROLLS-ROYCE CONTROL SYSTEMS DEVELOPMENT.....	82
4.2.1	<i>Background – Products and Processes</i> .....	82
4.2.2	<i>Development Process</i> .....	84
4.2.3	<i>Concurrency and Iteration</i> .....	85
4.2.4	<i>Interdependencies between Resources, Products and Processes</i> .....	86
4.3	PROJECT-LEVEL CONCURRENCY.....	87
4.4	DELIVERY-LEVEL CONCURRENCY .....	92
4.5	PHASE-LEVEL CONCURRENCY .....	94
4.6	TASK-LEVEL CONCURRENCY.....	97
4.7	SUMMARY OF OBSERVATIONS .....	97
<b>CHAPTER 5: PROBLEMS OF TIME-CONSTRAINED DEVELOPMENT.....</b>		<b>99</b>
5.1	INTRODUCTION .....	99
5.2	THREE PROBLEMS OF TIME-CONSTRAINED DEVELOPMENT .....	99
5.2.1	<i>General Problems</i> .....	99

5.2.2	<i>Problem 1 - Overloading</i> .....	100
5.2.3	<i>Problem 2 – The Bow-Wave</i> .....	102
5.2.4	<i>Problem 3 – The Multiplier Effect</i> .....	106
5.3	GENERALITY OF FINDINGS : A COMPARISON WITH BAE SYSTEMS .....	110
5.3.1	<i>Background</i> .....	110
5.3.2	<i>Project and Delivery Concurrency</i> .....	110
5.3.3	<i>Phase Team Concurrency</i> .....	112
5.3.4	<i>Delivery Phase Concurrency</i> .....	112
5.3.5	<i>Discussion</i> .....	113
5.4	GENERAL IMPLICATIONS FOR MEASUREMENT .....	115
5.5	SUMMARY OF PROBLEMS OF TIME-CONSTRAINED DEVELOPMENT .....	115
<b>CHAPTER 6: MODELLING TIME-CONSTRAINED DEVELOPMENT .....</b>		<b>117</b>
6.1	INTRODUCTION .....	117
6.2	APPROACHES TO MODELLING.....	117
6.2.1	<i>Conventional Predictive Models</i> .....	117
6.2.2	<i>The Rolls-Royce Lead-Time Model</i> .....	118
6.3	A MODEL OF TIME-CONSTRAINED SOFTWARE DEVELOPMENT.....	120
6.3.1	<i>Overview of Capacity Based Scheduling</i> .....	120
6.3.2	<i>A Formal Description of the Primitive Model</i> .....	121
6.3.2.1	Step 1 – Input the Planned Schedule .....	122
6.3.2.2	Step 2 – Calculate the Required Effectiveness .....	123
6.3.2.3	Step 3 – Calculate the Applied Resource.....	123
6.3.2.4	Step 4 – Calculate the Applied Effort .....	124
6.3.2.5	Step 5 – Calculate the Required Productivity.....	124
6.3.2.6	Step 6 – Calculate the Achieved Work .....	124
6.3.2.7	Step 7 – Iterate .....	124
6.4	MODEL WORKED EXAMPLES .....	125
6.4.1	<i>Model Example 1: Two Deliveries, One Phase</i> .....	125
6.4.2	<i>Model Example 2: Two Staged Deliveries, One Phase</i> .....	129
6.4.3	<i>Model Example 3: Two Staged Deliveries, Two Phases</i> .....	131
6.5	SUMMARY OF MODELLING TIME-CONSTRAINED DEVELOPMENT .....	133

## **CHAPTER 7: MANAGING TIME-CONSTRAINED DEVELOPMENT.....134**

7.1	INTRODUCTION .....	134
7.2	MODEL EVALUATION.....	134
7.2.1	<i>Evaluation Method.....</i>	<i>134</i>
7.2.2	<i>Results of the Model Run .....</i>	<i>135</i>
7.2.3	<i>Model Validity - Analysis of Applied Resource .....</i>	<i>138</i>
7.2.4	<i>Model Utility - Analysis of Required Effectiveness.....</i>	<i>141</i>
7.2.5	<i>Conclusions of the Evaluation .....</i>	<i>144</i>
7.2.6	<i>Extensions to the Model.....</i>	<i>144</i>
7.3	MODEL APPLICATION .....	145
7.3.1	<i>Overview.....</i>	<i>145</i>
7.3.1.1	Step 1 – High-Level Planning.....	145
7.3.1.2	Step 2 – Detailed Planning.....	145
7.3.1.3	Step 3 – Model Plans .....	146
7.3.1.4	Step 4 – Evaluate Productivity ( <i>Required Productivity</i> ).....	146
7.3.1.5	Step 5 - Evaluate Work ( <i>Planned &amp; Achieved Work</i> ).....	148
7.3.1.6	Step 6 - Evaluate Resources ( <i>Planned &amp; Applied Resource</i> ) .....	149
7.3.1.7	Step 7 - Evaluate Effort ( <i>Applied Effort</i> ).....	150
7.3.1.8	Evaluate Schedule ( <i>Planned Start &amp; End Time</i> ).....	150
7.4	SUMMARY OF MANAGING TIME-CONSTRAINED DEVELOPMENT.....	151

## **CHAPTER 8: CONCLUSIONS .....152**

8.1	THESIS CONTRIBUTION .....	152
8.2	WORK DONE, CLAIMS AND EVIDENCE.....	153
8.2.1	<i>Literature Review (Ch. 2).....</i>	<i>153</i>
8.2.2	<i>Measuring Time-Constrained Software Development (Ch. 3) .....</i>	<i>153</i>
8.2.3	<i>Observations on Time-Constrained Software Development (Ch. 4) .</i>	<i>154</i>
8.2.4	<i>Problems of Time-Constrained Software Development (Ch. 5).....</i>	<i>154</i>
8.2.5	<i>Modelling Time-Constrained Software Development (Ch. 6) .....</i>	<i>155</i>
8.2.6	<i>Managing Time-Constrained Software Development (Ch. 7) .....</i>	<i>155</i>
8.3	FUTURE WORK.....	156
8.4	FINAL CONCLUSIONS .....	157

<b>GLOSSARY OF TERMS.....</b>	<b>158</b>
<b>APPENDIX A : MODEL IMPLEMENTATION .....</b>	<b>167</b>
A.1    BACKGROUND.....	167
A.2    MODEL SOURCE CODE.....	169
<b>APPENDIX B : MODEL ASSUMPTIONS AND EXTENSIONS .....</b>	<b>174</b>

## List of Figures

Figure 1: Three Modes of Software Development .....	16
Figure 2: Example Work Breakdown Structure (WBS).....	49
Figure 3: Example Process Lexicon (PEL) .....	53
Figure 4: Measurement using PEL .....	54
Figure 5: Goals Questions Metrics for Time-Constrained Development.....	56
Figure 6: Metrics for Time-Constrained Development.....	56
Figure 7: Cost Booking System (CoBS) – Effort Collection.....	58
Figure 8: Cost Booking System (CoBS) – Semantic Specification.....	59
Figure 9: CoBS Analysis Step 1 - Importing PEL Data into Microsoft Excel .....	60
Figure 10: CoBS Analysis Step 2 – Creating a Pivot Table .....	61
Figure 11: CoBS Analysis Step 3 – Using Pivot Table Groups .....	62
Figure 12: CoBS Analysis Step 4 – Generating Charts .....	63
Figure 13: CoBS Analysis Step 5 – Analysis Using Multiple Queries.....	64
Figure 14: CoBS – Costs and Time-Delays of Rework.....	65
Figure 15: Comparison of Queries Supported by Conventional WBS and PEL.....	66
Figure 16: RoCC – Lifecycle of a SAN.....	67
Figure 17: RoCC – Defect Metrics Collected when a SAN is Raised.....	68
Figure 18: RoCC – Defect Metrics Collected when a SAN is Investigated.....	70
Figure 19: RoCC – Defect Metrics Collected when a SAN is Closed.....	71
Figure 20: RoCC – Analysis of Defect Metrics.....	72
Figure 21: RoCC – Metrics Reporting Tool.....	73
Figure 22: RoCC – Raw Data in Microsoft Excel.....	73
Figure 23: RoCC – Defect Detection Profile .....	74
Figure 24: Comparator – Raw Size Data.....	77
Figure 25: Comparator – Analysis of Size Data .....	78
Figure 26: Comparator – Levels of Product Reuse .....	79
Figure 27: Rolls-Royce – Development Schedule .....	83



Figure 28: Rolls-Royce – Development Process.....	84
Figure 29: The Hierarchy of Concurrency.....	86
Figure 30: Rolls-Royce – BR700 Control Systems Team Structure Chart.....	87
Figure 31: Rolls-Royce – Project Level Concurrency.....	88
Figure 32: Rolls-Royce – Team Level Concurrency.....	89
Figure 33: Classical Software Lifecycle Effort Profile (McDermid and Rook 1991) .....	90
Figure 34: Time-Constrained Software Lifecycle Effort Profile .....	91
Figure 35: Rolls-Royce – Delivery Level Concurrency.....	93
Figure 36: Rolls-Royce – Delivery Template .....	94
Figure 37: Rolls-Royce – Phase Level Concurrency ( <i>Delivery C4.3</i> ).....	95
Figure 38: Rolls-Royce – Phase Concurrency between Deliveries .....	96
Figure 39: Problem – The Spiral of Process Compression.....	100
Figure 40: Problem 1 – Overloading .....	101
Figure 41: Problem 2 – Bow-Waves.....	104
Figure 42: Problem 3 – The Multiplier (Change Between Deliveries).....	107
Figure 43: Problem 3 – The Multiplier (Product Change over Time) .....	108
Figure 44: Problem 3 – The Multiplier (Relative Effort per Delivery).....	109
Figure 45: BAE SYSTEMS – Project and Delivery Concurrency.....	111
Figure 46: BAE SYSTEMS – Phase Team Concurrency .....	112
Figure 47: BAE SYSTEMS – Delivery Phase Concurrency.....	113
Figure 48: Comparison of Phase Effort in Rolls-Royce and BAE SYSTEMS.....	114
Figure 49: Conventional Predictive Models of Software Development .....	118
Figure 50: Rolls-Royce – Lead-Time Model.....	119
Figure 51: Capacity-Based Scheduling – Concept.....	120
Figure 52: Capacity-Based Scheduling – Primitive Model.....	122
Figure 53: Model Example 1 – Inputs and Outputs.....	127
Figure 54: Model Example 1 – Results (Two Deliveries, One Phase) .....	128
Figure 55: Model Example 2 – Results (Two Staged Deliveries, One Phase) .....	130

Figure 56: Model Example 3 – Results (Two Staged Deliveries, Two Phases) .....	132
Figure 57: Model Example 4 – Rolls-Royce Results.....	136
Figure 58: Model Example 4 – Rolls-Royce Results (continued) .....	137
Figure 59: Model Analysis – Model’s Allocation of Code Effort.....	139
Figure 60: Model Analysis – Actual Allocation of Code Effort .....	139
Figure 61: Model Analysis – Actual versus Predicted <i>Applied Effort</i> .....	140
Figure 62: Model Analysis – Predicted Levels of Code Phase <i>Required Effectiveness</i> .....	141
Figure 63: Model Analysis – Total Predicted Code Phase <i>Required Productivity</i> .....	142
Figure 64: Model Analysis – Total Low Level Testing Phase <i>Required Productivity</i> .....	143
Figure 65: Model Application – Evaluation of Code <i>Required Productivity</i> .....	147
Figure 66: Model Application – Evaluation of LLT <i>Required Productivity</i> .....	148
Figure 67: Model Application – Evaluation of Achieved Work.....	149
Figure 68: Model Application – Evaluation of <i>Planned Resource</i> .....	150
Figure 69: The VenSim Model.....	167
Figure 70: The VenSim Equation Editor .....	168

## Acknowledgements

This research has been supported by the Engineering and Physical Sciences Research Council (grant 95594911), the European Systems and Software Initiative (21670), Rolls-Royce plc and BAE SYSTEMS plc. It was conducted at the Department of Computer Science and Department of Management Studies at the University of York.

At York I would particularly like to thank Professor John McDermid whose guidance, patience and professionalism has been truly inspirational. I also appreciate the help of Professor Keith Mander (now University of Kent), Professor Ian Wand and John Clark for their support at the start and (eventual) end of this research.

At Rolls-Royce plc I would like to thank Dr. Eddie Williams and Dr. Brian Cooke for their support and encouragement throughout this work. I also appreciate the help of Duncan Brown, Keith Harper, Stuart Hutchesson and Andy Nolan.

At BAE SYSTEMS plc I would like to thank Graham Clark for sharing the PEL approach and supporting this research. Thanks too go to Mike Burke and Mike Mason for their assistance.

I would also like to thank Professor Manny Lehman (Imperial College), Professor Barbara Kitchenham (Keele University) and Professor Tarik Abdel-Hamid for their insightful research that has inspired many of the ideas presented here.

Finally, I would like to thank Dr. Julia Hill who has made this research worthwhile.

This thesis is dedicated to my family – past, present and future.

## Author's Declaration

Some of the material presented within this thesis has previously been published in the following papers:

- A. L. Powell, D. S. Brown, “A Practical Strategy for Industrial Reuse Improvement,” *Software Process: Improvement and Practice*, Vol.4, Iss.3, September 1999.
- A. L. Powell, K. C. Mander and D. S. Brown, “Strategies for Lifecycle Concurrency and Iteration - A System Dynamics Approach,” *Journal of Systems and Software*, Vol.46, Iss.2-3, pp.151-161, June 1999.
- G. Clark and A. L. Powell, “Experiences with Sharing a Common Measurement Philosophy,” International Conference on Systems Engineering (INCOSE'99), Brighton, May 1999.
- A. L. Powell, “A Literature Review on the Quantification of Software Change,” *University of York Computer Science Yellow Report*, YCS 305, August 1998
- A. L. Powell, K. C. Mander and J. A. McDermid, “Research Challenges in Concurrent Software Development Environments,” Process Modelling and Empirical Studies (PMESSE) Workshop at the International Conference on Software Engineering ICSE'97, Boston, June 1997.

All the work contained within this thesis represents the original contribution of the author.

## **List of Acronyms**

CBS	Capacity-Based Scheduling
CCB	Change Control Board
CMM	Capability Maturity Model
CoBS	Cost Booking System
EEC	Electronic Engine Controller
FADEC	Full Authority Digital Engine Controller
FEAST	Feedback Evolution And Software Technology
GQM	Goal-Question-Metric
IP	Implementation Policy
LoC	Line(s) of Code
PEL	Process Engineering Language
RoCC	Rolls Change Control system
SAN	System Anomaly Note
WBS	Work Breakdown Structure(s)

# Chapter 1: Introduction

---

## 1.1 Background

The development of software is dependent upon resources, processes and products. *Resources* are the people, equipment and raw materials that are used to perform a process. They are usually limited in supply and therefore have an associated *cost*. *Processes* are a “*set of interrelated activities which transform inputs into outputs*” (ISO 1995). They use resources in the generation of products over *time*. *Products* are the intermediate and final outputs of a process, and have associated *qualities*.

Ideally, software development would involve an appropriate supply of resources over a necessary timescale in the generation of the ‘perfect’ product. In practice, however, software development is a commercial endeavour that places constraints on the cost of resources consumed, the timescale of the process performed, and the quality of the finished product. The structured discipline of producing software under these constraints is termed *Software Engineering* (Naur and Randell 1969).

The goal of software engineering is “*to obtain economically software that is reliable and works efficiently on real machines*” (Bauer 1971). The term was coined in response to a growing crisis in software development; “*Existing software comes too late and at higher costs than expected and does not fulfil the promises made for it*” (ibid.). Yet, despite numerous advances in software process technology, these problems remain. It appears that the demand for increasingly complex applications is outstripping our capability to supply them (Duncan 1988).

A critical demand for many organisations is that of lead-time. The commercial rewards of being ‘first-to-market’ with a new product or service can be dramatic. Reduced development lead-times can give large increases in market share, longer product life and higher profit margins (Collier and Collofello 1995). These commercial pressures therefore require the development of systems and software in situations where project deadlines are very tight and often non-negotiable (i.e. the costs of failing are very high). This type of development can be termed *time-constrained software development*.

## 1.2 Time-Constrained Software Development

The pressures on software development timescales are growing. Improvements, such as the use of commercial off-the-shelf technology, have caused both the costs and timescales associated with hardware to fall, thus making software a critical path component in the provision of many products and services (Sims 1997).

Organisations must therefore continuously seek ways to improve their lead-time capability and achieve more with diminishing resources. These improvements must come from using: the right people, processes and tools; eliminating unnecessary work; being right-first-time; or by doing many things at once (Parkinson 1996). The '*time-to-market*' principle assumes that savings or benefits outside of the process can outweigh any additional costs of compressed timescales.

The demands on lead-time compression are increasingly being met by the application of *evolutionary development* lifecycles. These overcome the problems of conventional lifecycles, typified by the Waterfall model (Royce 1970), that imply a sequential once-through approach to development. These models assume that software can be developed 'right-first-time' and that lead-times are sufficiently long to proceed in a stepwise manner. Instead, evolutionary lifecycles such as the Spiral model (Boehm 1988) recognise the need for software artefacts to evolve over time in a controlled risk-driven manner. These lifecycles have effectively replaced the notion of '*right-first-time*' with '*right-on-time*.'

The two essential elements of evolutionary lifecycles are iteration and concurrency. *Iteration* is the repetition of development activities to deliver increments of product functionality at pre-planned intervals. Incremental development, or staged-delivery, balances progress and early feedback against the overheads of repeating the process a number of times. *Concurrency* is the simultaneous performance of development activities between projects, product deliveries, development phases and individual tasks.

The extent of concurrency and iteration therefore distinguishes time-constrained software development from more conventional processes. These differences can be illustrated as three 'modes' of software development as illustrated in Figure 1.

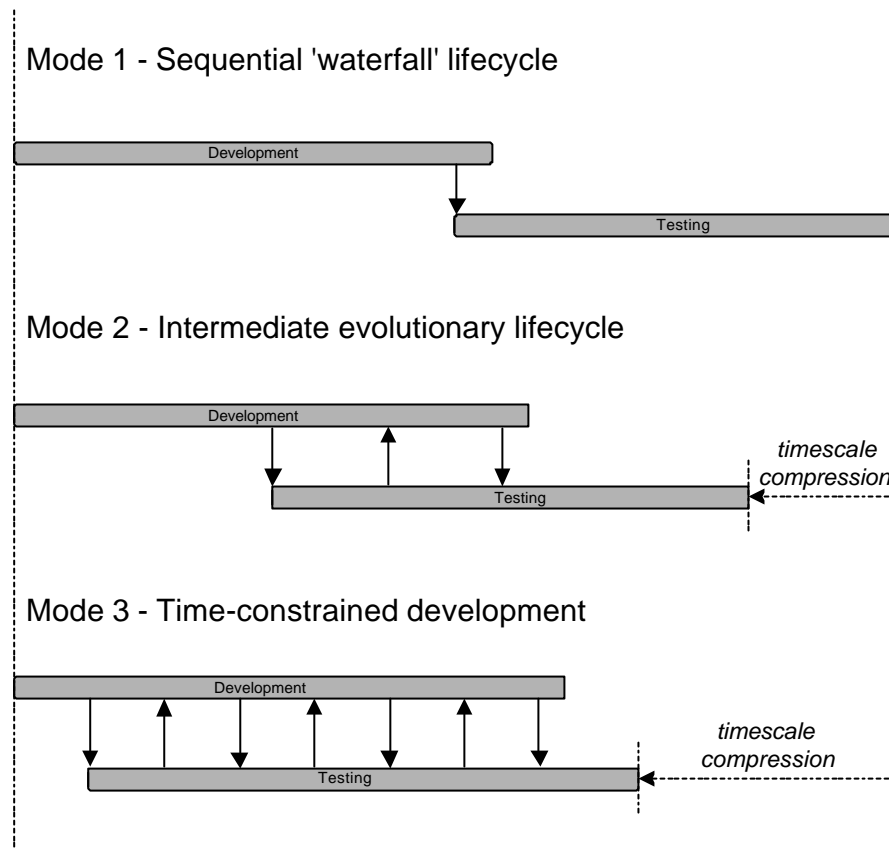


Figure 1: Three Modes of Software Development

*Mode 1* represents a sequential 'waterfall' model (Royce 1970) of software development that lays development activities end-to-end. The intrinsic assumption that software can be developed right-first-time is, however, unrealistic and the resulting timescale is commercially impractical for many organisations (Boehm 1988).

*Mode 2* represents an intermediate evolutionary lifecycle using a limited amount of concurrency and iteration. This allows feedback in the software process to detect and fix problems whilst, in theory, reducing the overall development lead-time. This approach is typical of conventional software development (Parnas and Clements 1986).

*Mode 3* represents the extreme form of evolutionary development that we have observed in industry. This exploits high levels of concurrency and staged-delivery to meet the time constraints imposed on the software process, and is based on three assumptions. First, that concurrency reduces overall development lead-times. Second, that iteration gives rise to improvements in product maturity. Third, that the benefits of reduced lead-times outweigh the costs of instability and rework.



The existence and effects of lifecycle concurrency and iteration are, however, largely implicit in the literature and are treated as an issue of project management rather than an integral part of the software lifecycle (Blackburn and Hoedemaker 1996). Conventional approaches to measurement, modelling and management treat the development process as being linear, sequential and static. Whereas, the industrial practices used to achieve timescale compression in industry are iterative, concurrent and dynamic. This thesis investigates this disparity between conventional approaches and industrial practice.

### **1.3 An Example of Time-Constrained Development**

This research uses industrial case studies of time-constrained software development in Rolls-Royce and BAE SYSTEMS, which took place from 1995 to 2000. It was our observations of the extreme time-constraints in aeroengine control systems development that originally prompted this research.

In Rolls-Royce, concurrency and iteration had been used to achieve substantial reductions in the lead-times of Engine Control system projects. However, it also caused undesirable consequences. The systems team, under pressure, were late in providing software requirements. In turn, the software team were committed to delivering increments of software functionality for engine testing to keep the whole engine programme on schedule. The pressures led to a growing number of change requests that had to be accommodated in an increasing number of unplanned ‘fix’ deliveries. It was therefore necessary to break the rework spiral. The question from the manager, Keith Harper, was: *“What is the optimum number of software deliveries?”*

At the risk of disappointing software managers, this research shows why there is no simple answer to this question. Rather, we challenge a premise of much contemporary software management literature that assumes rework is a bad thing and should be eliminated. We then propose practical strategies for the management of time-constrained software development.

## 1.4 Research Aims

This thesis investigates the following proposition:

*It is difficult to measure, model and manage time-constrained (iterative and concurrent) software development using conventional techniques. Instead, more fine-grained approaches to measurement and modelling are required in order to help plan, control and improve time-constrained software development.*

In order to do this, three main research questions were explored.

(i) *How do we measure time-constrained software development?*

Measurement is a fundamental part of any engineering discipline (DeMarco 1982). However, the nature of software development as a human and design-led activity makes it inherently difficult to quantify. This is particularly true in the case of time-constrained development where timescale reduction can drastically increase the levels of software change; “... *the effects of which can propagate explosively*” (McDermid 1991). We therefore need to re-evaluate present approaches to measurement as a basis for understanding time-constrained development.

(ii) *How do we model time-constrained software development?*

The disparity between conventional models and industrial practice suggests that we do not have representative models of time-constrained development. We need better models for three reasons: (a) *description* – to provide a basis for communicating our understanding, (b) *analysis* – to determine patterns, regularities and trends that reflect the link between process structure and behaviour, and (c) *prediction* – to reason about behaviour at some point in the future.

(iii) *How do we manage time-constrained software development?*

*Estimation and Planning.* In time-constrained development the project timescales are often dictated by the external demands of the end system and its market need, rather than internal choice. Hence, managers need to understand if these demands are reasonable and how they may be met.

*Monitoring and Control.* The iterative nature of time-constrained development can make progress difficult to judge and therefore exposes projects to increased risks of slippages, exceeded budgets or poor product quality. Hence, managers need better ways to monitor and control progress to be sure of meeting deadlines.

*Evaluation and Improvement.* The limitations in current approaches to measurement and modelling mean that we have no effective way of evaluating alternative strategies for time-constrained development. Hence, we need better measures and models to help managers make decisions that are more informed and to achieve process improvement.

## 1.5 Thesis Structure

In Chapter 2, *Literature Review*, we survey the published literature on the measurement, modelling, and management, of software change. In our findings, we identify a set of questions that motivate this research.

In Chapter 3, *Measurement*, we introduce an approach for the detailed or ‘fine-grained’ collection of software metrics called the Process Engineering Language (PEL). We describe our work to extend and apply this approach within Rolls-Royce to give benefits over conventional approaches to metrics collection and insights into time-constrained development.

In Chapter 4, *Observations*, we use PEL effort metrics to make specific observations about the nature of time-constrained software development within Rolls-Royce. The effects of concurrent and iteration are identified and used to show how time-constrained processes differ from their conventional counterparts.

In Chapter 5, *Problems*, we use our PEL effort, defect, and size, metrics to isolate three specific problems of time-scale compression: overloading, bow-waves and the multiplier effect. These problems demonstrate the need for more representative ‘fine-grained’ models of time-constrained software development.

In Chapter 6, *Modelling*, we develop a new modelling approach called Capacity-Based Scheduling (CBS). We show how schedule constraints can be modelled in order to predict the consequences of alternative plans and control schedule risk across a portfolio of time-constrained projects.

In Chapter 7, *Management*, we draw together our strands of research by evaluating the validity and utility of our modelling approach on an industrial dataset. We then show how the CBS approach can be applied to give practical strategies for the management of time-constrained software development.

In Chapter 8, *Conclusions*, we present the contributions of this research and identify areas for future work.

A glossary of terms is provided on pages 158 to 166.

# Chapter 2: Literature Review

---

## 2.1 Introduction

The specific nature and problems of time-constrained software development have not previously been considered as an academic topic in their own right. As a starting point, we therefore base our review on the broader subject of *software change* (Section 2.2). In particular, we show that change is an intrinsic part of software development and must be reflected as such in our approaches for *measurement* (Section 2.3), *modelling* (Section 2.4) and *management* (Section 2.5). Finally, we summarise the key questions for the measurement, modelling and management of time-constrained software development as a basis for this research (Section 2.6).

## 2.2 Software Change

Early models of the software development process, such as Royce's Waterfall model (Royce 1970), were based on a simple sequence of canonical phases (specification, design, coding and testing activities). However, these models were soon identified as unrealistic since they assumed that complete and consistent specifications could be produced prior to design and implementation (Parnas and Clements 1986).

Later models, such as Boehm's Spiral (Boehm 1988) and Gilb's Evo (Gilb 1985), have been more representative of the software lifecycles used in practice. These models more accurately represent the need for software artefacts, and the processes used to generate them, to evolve over time in response to their changing environment. They also acknowledge the need for change to both product and process in a systematic risk-driven manner.

Whilst the need for change has long been recognised as an intrinsic part of the software engineering process (Lehman 1976), it has not always been reflected as such. In their excellent paper, Parnas and Clements highlight the paradox between the ideal, linear, stepwise software process (captured by academics and standards authorities) and the actual iterative software process performed by practising software engineers (Parnas and

Clements 1986). They give reasons (paraphrased here for brevity) why software development is unlikely to proceed in a perfectly rational way.

- (i) The people who commission the building of a software system frequently do not know exactly what they want and are unable to tell us all that they know.
- (ii) Even if we knew the requirements, many of the details only become known to us as we progress into the implementation. Costs do not permit us to backtrack easily on designs.
- (iii) Human beings are unable to comprehend fully the plethora of details that must be taken into account in order to design and build a correct system, increasing the likelihood of errors being made.
- (iv) All but the most trivial projects are subject to change for external reasons.
- (v) Human errors can only be avoided if one can avoid the use of humans.
- (vi) We are often burdened by preconceived design ideas.
- (vii) Often we are encouraged, for economic reasons, to use software that was developed for some other project.

The conventional treatment of change in the literature has been largely confined to considering change as formal configuration management or change after delivery (software maintenance). There are, however, several types of software change.

- (i) *First-time-through* production of software artefacts can be considered as an extreme form of change.
- (ii) *Rework* is an in-process form of change caused by inconsistencies between the different abstractions of software products (specification, design, code, etc.) or between these artefacts and the world they purport to model.
- (iii) *Maintenance* is change after delivery of the product, to meet changes in the product's operational environment, to remove remaining defects, or to provide desired enhancements.
- (iv) *Reuse* is essentially about avoiding change but can require the planned change or migration of legacy software into reusable components.

The subject matter of 'software change' therefore spans a number of areas of software development, each having individual characteristics and difficulties. Despite the

conventional treatment of them as separate problems, current research acknowledges that there are underlying relationships between them. For example, Parets and Torres argue for a view of software as an evolutionary development process of interactions between the software development team and the user system (Parets and Torres 1996). Software maintenance is conceived as a natural projection of evolutionary development rather than the current assumption that there is a point in time when the software product is finished and delivered. Likewise, Basili argues for viewing maintenance as '*re-use oriented development*' (Basili 1990). The improvement of software change is therefore a key part of process improvement.

There has been relatively little research seeking to investigate and explain aspects of software change in its own right. Lehman was one of the earliest to highlight the contradiction of software engineering that, on one hand, strives for absolute correctness of artefacts, yet attempts to model a real world that is continuous and unbounded (Lehman 1976). To capture the essence of the problem, Lehman classified program types by 'correctness.' He identifies S-type, P-type and E-type programs.

- (i) *S-type* programs are required to satisfy a pre-stated specification. Correctness is the absolute relationship between the specification and the program.
- (ii) *P-type* programs are required to form an acceptable solution to a stated problem in the real world. Correctness of P-type programs is determined by the acceptability of the solution.
- (iii) *E-type* programs are required to solve a problem or implement an application in a real-world domain but they themselves change that domain (i.e. they 'change the world'). Correctness here is determined by the program's behaviour under operational conditions.

Lehman argues that there are limits to the confidence we can have in the results of execution of these E-type programs, viewing them as organisms '*driven by human experiences, interpretation, reaction, decision and all that these imply.*'

Lehman observed the feedback effects of E-type systems as growth and decay of large software systems and formulated these as laws of program evolution (Lehman 1978; Lehman 1996).

- (i) *Continuing change* – E-type systems are prone to change and become progressively less useful.

- (ii) *Increasing complexity* - software data and control structure deteriorates as more and more changes occur (unless work is done to maintain or reduce it).
- (iii) *Self-Regulation* (the fundamental law of program evolution) - software evolution is subject to feedback processes that act to stabilise evolution based on determinable trends and invariance.
- (iv) *Conservation of organisational stability* - the activity of an organisation to support an evolving program attempts to maintain stability and sustainable growth.
- (v) *Conservation of familiarity* (perceived complexity) - the release content (changes, additions, deletions) of successive releases of an evolving program is statistically invariant.
- (vi) *Continuing Growth* - the functional content of a program must be increased to maintain user satisfaction over its lifetime.
- (vii) *Declining Quality* - E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

Lehman's work has demonstrated that change is an essential and unavoidable aspect of software development. The capability of the software process to respond to and handle this change is an important driver of process quality. Measurement is therefore essential in understanding the nature of software change.

## **2.3 Measuring Software Change**

### **2.3.1 The History of Software Measurement**

Measurement is fundamental to any engineering discipline for three main reasons:

- (i) *Control*: Our desire to measure the process is driven by the need to control it (DeMarco 1982). Measurement provides feedback for reasoning about alternative actions.
- (ii) *Assessment*: Measurement gives us the ability to evaluate the status of product and process, identify trends and patterns, and make value judgements.



- (iii) *Prediction*: Measurement can be used to predict some future value of an attribute to aid decision-making.

The term ‘metrics’ refers to the “*measurement (and study) of those attributes of the development process and products which we need to understand in order to achieve project control*” (McDermid 1997). Since metrics are based on underlying assumptions about the world that they purport to capture, it is impossible to talk about measurement without implying some form of model.

The last 20 years has seen a large amount of research aimed at determining measurable properties of, and mathematical relationships between, product qualities and process quantities. Much of this work attempts to capture notions of ‘complexity’ and relate them to actual or expected external properties such as development effort or product quality. A quantitative understanding is fundamental for the control, assessment and prediction of software process attributes.

The earliest measures were based on analysis of software code, the most fundamental being a basic count of the number of Lines of Code (or LoC). Despite being widely criticised as a measure of complexity, it continues to have widespread popularity mainly due to its simplicity (Azuma and Mole 1994). This has led Basili to suggest that LoC be used as a ‘null hypothesis’ in experiments to compare the utility of other software measures (Basili and Weiss 1984).

Maurice Halstead pioneered the search for theoretically based software measures with predictive capability. Halstead’s Software Science modelled program comprehension as a function of program *operands* (variables and constants) and *operators* (arithmetic operators and keywords which alter program control flow) (Halstead 1975). The concept triggered a large number of papers correlating the measure, and proposed extensions, to various external properties (such as error-proneness) but many of these studies have been shown to be flawed (Hamer and Frewin 1982).

Another early measure, proposed by McCabe (McCabe 1976), viewed program complexity related to the number of control paths through a program module by a function

$$V(G) = e - n + 2p$$

where  $e$  is the number of edges in the control graph (equivalent to branching points),  $n$  is the number of vertices (equivalent to a sequential block of code) and  $p$  is the number

of connected components. Modules with higher ‘cyclomatic complexity’ had more control loops and selections, and were considered more difficult to test. Despite its limitations (such as its neglect for data flow complexity) McCabe’s cyclomatic complexity measure was an improvement over existing methods because of its use in predicting testing effort and aiding evaluation of alternative decomposition strategies.

The general problems of early software measures have been well documented in work such as that of Shepperd (Shepperd 1992). These problems revolve around the use of the measures on software code (thereby lacking predictive capability at an early stage in the process), the use of erroneous assumptions (such as a particular programming environment) and being based on poorly articulated models (with ill-defined aims and counting rules).

### **2.3.2 *Measurement Theory and Frameworks***

The software measurement field has been plagued by problems of defining measures, collecting consistent and useful data cost-effectively, performing analyses, interpreting results and putting these to good use. As Briand observes, hundreds of measures have been defined in the literature but few have been accepted in academia and even less in industry (Briand, Morasca et al. 1996). Further confusion arises over whether measurements are taken to analyse the past, monitor the present, or predict the future.

The last few years have seen a watershed in software measurement research with the recognition of the need for a more disciplined and scientific approach. This work is still developing but has paved the way for major improvements in the quality of measurement research. Notably it has demonstrated

- (i) the importance of rigorous application of measurement principles (Baker, Bieman et al. 1990) and statistical methods (Kitchenham, Pickard et al. 1990),
- (ii) the importance of a sound methodological and experimental basis for empirical studies (Kitchenham, Linkman et al. 1994) and explicit documentation such that they can be validated (Fenton 1994),
- (iii) the importance of multiple measures to capture multi-dimensional facets of software such as ‘maintainability’ and to capture differences in software environments (Munson and Khoshgoftaar 1992),

- (iv) the importance of predictive measures (or indicators) earlier in the process (i.e. at specification or design rather than code) where strategic change is easier to accommodate (Fenton 1991),
- (v) the importance of practical application, such as the embedding of software measures within a decision setting (Baker, Bieman et al. 1990).

Further research is still needed to identify frameworks and techniques for measure evaluation, selection, integration and application (Schneidewind 1992; Evanco and Lacovara 1994; Fenton 1994). As Shepperd and Ince write *‘There is no shortage of metrics; what is less in evidence though, are metrics that are generally valid, usable, and useful, and thus widely accepted’* (Shepperd and Ince 1994). The difficulty of this task represents the unsuitability of the software environment for conducting experiments due to the reliance on, and inherent variability of, humans in the process.

Another factor hampering progress is the difficulty of collecting ‘valid’ (i.e. meaningful and precise) engineering data (Basili and Weiss 1984; Andersen 1992). The lack of suitable data extraction and analysis tools has implied costly and unpopular manual collection, and often results in data that is corrupted by project or organisational ‘noise.’ Organisations are frequently unwilling to publish or share data because of its commercially sensitive nature, and, even when they do, the data has a limited ‘shelf-life’ because of rapid changes in development environments and practices. These problems have led to the formation of large collective bodies such as the Software Engineering Institute (SEI), the Software Engineering Laboratory (SEL) and the International Software Engineering Research Network (ISERN). Although the situation is improving, the difficulty of collecting good quality data will continue to hinder large-scale measurement research for some time to come.

The null hypothesis of lines of code has commonly been found to outperform many of the more complex composite measures of software development (Lind and Vairavan 1989). Considering these circumstances, it is perhaps unsurprising that the most practical and successful measures to date appear to be based on simple atomic counts.

The earlier sections have shown change to be an intrinsic feature of software development. The remainder of this section reviews current research relating to the measurement of software change in five broad categories:

- (i) *Change counts and causal analysis* - measures of the quantity and causes of change.
- (ii) *Statistical defect models* - models to predict future change from defect profiles.
- (iii) *Requirements volatility* - measures of the likelihood of requirements change.
- (iv) *Error-proneness* – predicting the need for change from errors.
- (v) *Ripple effect* - the impact of ‘second-order’ change on the system.

The following sections cover each of these areas in turn.

### **2.3.3 Change Counts and Causal Analysis**

The most basic change measures quantify the amount and nature of change at different periods in time. Linkman distinguishes between three types of monitoring (Linkman 1990).

- (i) *Checkpoint monitoring* takes a snapshot at the end of stages defined in a lifecycle. Comparison against quantifiable targets for activities and outputs indicates progress and anomalous behaviour.
- (ii) *Time-based monitoring* of an individual entity (module, document or subsystem) detects deviation from an expected trend.
- (iii) *Inter-checkpoint monitoring* observes inter-phase build-up status of product and activities towards a checkpoint.

The basic data collected is the number, impact (both cost and quality), reason for and type of change. Causal analysis can be performed to determine and eliminate the root cause of problems. This information can be analysed using techniques such as classification, Pareto analysis, scatter-plots and rank correlation to identify patterns in the data and areas requiring attention (Daskalantonakis 1992).

The cost of performing software changes has been shown to rise rapidly as the software lifecycle progresses (Basili and Weiss 1984). For example, a study by Jaffe found the cost of requirements errors not caught until software test to be as much as 200 times more than if they were caught during requirements engineering (Jaffe, Leveson et al. 1991). Defect removal programmes therefore account for many of the published studies

of industrial process improvement, such as those by Hewlett-Packard (Grady and Caswell 1987), IBM (Mays, Jones et al. 1990) and NASA (Rombach, Ulery et al. 1992).

There are, however, limitations to change measurement programmes. Their reliance on historical comparison means that their predictive potential is limited. By the time problems are observed it can be too late to take corrective action. The main benefit therefore results from improving the process for next time.

### **2.3.4 Statistical Defect Models**

There has been a large amount of research aimed at the prediction of software defects from historical patterns of defect detection and/or removal. Much of this work has been driven by the desire to prove the software is reliable and fit for release

Reibman (Reibman 1991) identifies the general applications of reliability models:

- (i) to help set and interpret system-reliability requirements;
- (ii) to predict the reliability of different system configurations;
- (iii) to identify reliability problems early in the product lifecycle so that designers can make changes when they are less expensive; and
- (iv) to determine cost-effective maintenance strategies.

Software reliability modelling has become a significant research area in its own right (Musa, Lannino et al. 1987). The intention here is not to review specific models but to make judgements on their overall success and usefulness as dynamic models of software change.

Yu (Yu, Shen et al. 1988) identifies the general form of parametric defect models as:

$$D = f(M, ED, T, \text{others})$$

where

- D Number of software defects found during a certain phase of the software lifecycle. This is the dependent variable of the model.
- M Program measures such as program size, number of decisions, number of variables, etc.

ED     Number of defects detected during an earlier phase of the software lifecycle.

T       Testing time measured in CPU time, calendar time or some other effort measure.

Others include hardware facilities, types of software, development effort, programmer's experience, design methodologies etc.

The models are generally based on the assumption that an increasing time between detected defects (developmental or operational failures) indicates increasingly stable reliability.

Much progress has been made since the early models like that of Musa (Musa 1975) but their predictive accuracy has been traditionally poor. This makes them unsuitable for use on very high reliability applications with requirements of  $10^{-6}$  failures per hour or greater (Brocklehurst and Littlewood 1992). There have been fundamental problems in many of the models suggested to date. They tend to oversimplify the failure process and assume perfect removal of flaws (i.e. no potential for bad fixes) and a flat rate execution profile (i.e. in practice some parts of the system are executed more than others).

The majority of models make assumptions about the system behaviour and its particular environment (equivalent to presuming the reliability growth profile of a telecommunications product to be the same as that of a nuclear reactor protection system). It is therefore unsurprising that published empirical validations have widely different results. Comparative studies such as that by Nikora (Nikora and Lyu 1995) have demonstrated that no one model will be trustworthy in all circumstances. Consequently, researchers such as Abdel-Ghaly (Abdel-Ghaly, Chan et al. 1986) have resorted to providing statistical methods to evaluate and select the most appropriate model for a particular application or domain.

More specific to the problem of software change is that these models are available late in the lifecycle (integration testing phase of development) thereby limiting their predictive potential. Accordingly, the models appear too late to control a process where such problems are dramatic and expensive to change (representing high-effort 'after the fact'). They also give no idea of the scale of change required (unless a mean fix time is specified). However, such models have had evidence of success in decisions of when to stop testing, an economic problem of diminishing returns, such as Dalal's net benefit

plot (Dalal and McIntosh 1994) and the model of Kanoun and Laprie (Kanoun and Laprie 1994).

In response to these problems, attention has shifted to the problem of predicting and proving product reliability during design and development (as opposed to testing). These include building reliability models into product design techniques and evaluating the integrity of software engineering tools and environments (Reibman 1991). However, we are still a long way from having statistical methods to prove product reliability or, for that matter, predictive models of software change.

### **2.3.5 Volatility, Completeness and Risk**

Requirements volatility measures provide a method of assessing if the quantity of, and reasons for, requirements change are consistent with current development activities. The basic form of requirements volatility analysis is based on counts of requirements added, deleted and modified, as well as classification of the reasons for change.

The paper by Costello and Liu on metrics for requirements engineering (Costello and Liu 1995) provides a comprehensive list of measures and analysis that can be performed on requirements. This includes measures of requirements completeness indicating the number of requirement decompositions still to be completed. The information highlights problems of requirement elicitation, capture, representation, as well as external causes of change. Analysis can identify the most changeable parts of a system, and the reasons, so that appropriate action can be taken. Trends in requirements volatility can also be used to indicate the status of a product in a process.

Although extremely valuable for control purposes, there are obvious limits to the predictive capability of such techniques, being constrained by the similarity of each application and the inherent volatility of its external environment. Ultimately this volatility reflects the uncertainty of software change (Section 2.1).

There has been a limited amount of research using requirements volatility in project risk analysis. For example, the semantic classification approach by Palmer (Palmer and Evans 1994). Such work has an important role to play in the management of software change but the majority of work in this area is currently based on qualitative assessment.

### 2.3.6 Error-Proneness

The main body of software measurement research has tried to extract certain properties of software design that form a notion of ‘complexity.’ The basic assumption is that more of this property a design or implementation has, then the more likely it is to contain defects, and the more effort is required to produce or test it. However, despite the many suggested measures there is still no accepted definition of what is meant by the term ‘complexity.’

The state of research in this area has been summarised by the points made in Section 2.3.2. The hope that a single measure can capture these properties is, to say the least, misguided, but experimental work into the elements of complexity does have a valuable role in providing feedback for improvement of design practices. A second approach was to focus testing resources more efficiently on modules with a high probability of error.

Space does not permit a thorough review of the numerous measures that have been suggested as indicators of defect potential. Generally they fall into the classes of *module length* (Basili and Weiss 1984), (Shen, Yu et al. 1985), (Withrow 1990), *token measures* (such as Halstead’s Software Science (Halstead 1975)), *control flow measures* (such as McCabe’s cyclomatic complexity (McCabe 1976)) and *information flow measures* (like Henry and Kafura’s information flow measure (Henry and Kafura 1981)). Measurement research has been dedicated to the study and evaluation of these measures and their failings are widely documented (Ince and Sheppard 1988), (Kitchenham and Linkman 1990), (Kafura and Canning 1985).

A further class of studies uses pattern recognition or classification based approaches to identify and predict high-risk components based on historical data. These observe that faults do not fall uniformly across components of software products, but rather concentrate on a small subset of components. Boehm, for instance, quotes a rule of thumb that 20 percent of a software system is responsible for 80 percent of its errors, cost and rework (Boehm 1988). Porter therefore proposed a classification tree that can be calibrated for a certain dataset and used to predict high-risk modules in order to focus testing resources (Porter and Selby 1990). This approach is, however, highly dependent on the selected characteristics of the classification tree.

In a similar vein, Munson and Khoshgoftaar (Munson and Khoshgoftaar 1990) have used discriminant analysis to examine the relationship between program faults and various



complexity attributes. To identify the unique contribution of each metric to the model they map the metrics into ‘orthogonal complexity domains.’ Discriminant analysis is then used to classify the dataset as modules that are fault-prone or not fault-prone. This technique was later applied to a large telecommunications project (Khoshoftaar, Allen et al. 1996) with some success.

Another approach tried by Khosgoftaar (Khoshoftaar and Lanning 1995) is the use of neural networks to perform the classification task. The benefit of neural nets is the ability to operate on noisy and non-normally distributed datasets. An inherent assumption is that considering as many metrics as possible can lead to a more complete model.

Briand (Briand, Basili et al. 1992) provided a good review of the criteria under which conventional regression analysis techniques and classification-based approaches are most appropriate. These classification techniques are notable for their expressive nature (less so for neural nets) and potential for automation. Their main problem, however, is the potential for a least common denominator to be returned (i.e. they can ignore a useful predictor). They have therefore been criticised by Shepperd (Shepperd 1992) for their scattergun and black-box approach to measure evaluation and selection. There is evidence, however, that both conventional regression and classification-based approaches are useful in developing understanding of the effects of different design strategies on defect potential (Briand, Basili et al. 1992).

### **2.3.7 *The Ripple Effect***

The earlier sections have considered what might be termed ‘first-order’ measures of software change, that is, the direct impact of change on a product. Change also causes ‘second-order’ impacts that can be seen to propagate through a system through a series of consequential changes that need to be made (i.e. a change to one module requiring change to other modules and so on). This ‘ripple’ effect is due to the structural properties of representations of software artefacts (inter-relatedness) and is a reflection of the cohesion and coupling between artefacts.

The work of Haney (Haney 1972) was among the earliest to attempt to measure the tendency for change to ripple through a product. Haney modelled a large software system using an impact matrix, each element representing the probability that change in

one module gives rise to change in another. This model was used to predict the number of changes required during maintenance in order to determine a strategy for staging releases of a system. The technique assumed that system requirements were frozen and relied on manually provided subjective probabilities thereby reducing its accuracy.

Yau (Yau and Collofello 1980) identifies general problems of the early measures proposed by Haney and those of Myers (Myers 1979). They assume all modifications to a module have the same ripple effect, and these are symmetrical (that if a change to module **a** impacts on module **b**, then the reverse must be true). Yau proposes a measure of *stability* (resistance to amplification of change) based on the connection of modules in a system by parameters and global variables based on both (i) intra-module change propagation, and (ii) inter-module change propagation. These measures are used to identify certain characteristics of a good software design. In their later work, Yau and Collofello turn their attention to design-based measures of potential ripple effect (Yau 1985) as early indicators of maintainability and a measure of design quality.

A main feature of Yau's work is his distinction between logical and performance stability. *Logical stability* is a measure of the resistance to the impact of modification on other modules in the program, expressed in terms of logical considerations. *Performance stability* is a measure of the resistance of a modification on other modules in the program, expressed in terms of performance considerations. Yau therefore highlights the importance of modelling both functional and non-functional impacts of change – a point largely neglected elsewhere in the literature.

There have been other attempts to model the ripple of change at a process level. Wohlin (Wohlin and Körner 1990), for example, captured the spread of faults and how they are affected by detection and correction activities.

The measures of stability have been demonstrated to be useful indicators of the maintainability of a software system. However, further empirical evaluation is required in order to validate these relationships in the light of new development practices.

### **2.3.8 Conclusions on the Measurement of Change**

Software change presents many problems for the control of software development; “...it is deceptively easy to introduce changes into software, the effects of which can propagate explosively” (McDermid 1991). It is possible to measure these dynamic

properties to provide feedback for process control (albeit somewhat delayed) and indicate opportunities for improvement of development techniques. These mechanisms provide the capability to evaluate current and expected amount of change (change counts), methods to reduce or minimise change (causal analysis and the impact of design techniques) and an ability to evaluate attempts to limit the impact of change (error-proneness and ripple effects).

The immaturity of the mechanisms described indicates the scale of research still required in understanding these low-level properties. The lack of progress can be attributed to difficulties of empirical validation as well as representing the numerous attributes of change. More fundamentally, they represent our inadequate understanding about the relationship between software product and process.

The divide between accidental and essential problems of software measurement has fundamental implications for the development of the field. Accidental problems can be avoided by a more rigorous scientific method as described in the papers above. Essential problems, however, must be handled either by adjusting our methods to account for them, or by adjusting our expectations of the methods themselves. Both types of problem imply a need for improved models of the software development process.

## **2.4 Modelling Software Change**

The phenomenon of software change dynamics is based on a number of highly interrelated factors: technical, operational and sociological. Our ability to control such a process depends on the quality of our models of these relationships. Models can aid reasoning about trade-offs in product design decisions (e.g. decomposition strategy) and process decisions (e.g. first-time-through versus cost of defect detection activities). They can also support the evaluation of alternative software engineering approaches (such as the use of formal methods). Having briefly surveyed research that quantifies dynamic aspects of change, this section considers attempts at developing predictive models of change.

### **2.4.1 Modelling Cost**

The earliest attempts at software process modelling were driven by the desire to accurately predict the cost of software development at an early stage in the process.

Models such as Boehm's COConstructive COst MOdel (COCOMO) (Boehm 1981) and Putnam's SLIM model (Putnam 1978) have been widely applied and have been subjected to critical evaluation and extension (Mukhopadhyay and Kekre 1992). The fundamental weakness of the early macro cost estimation models was their basis on lines of code (LoC) as a measure of system size - accurate figures of LoC are generally only available late in the process by which time estimation is less of a problem.

Alternative sizing mechanisms that take early information from designs or specification have been suggested. The most notable is Albrecht's Function Point Analysis (Albrecht and Gaffney 1983) that estimates product size by counting the number of inputs, outputs, inquiries, logical files and interfaces. The counts are then empirically adjusted to normalise their impact, and the total is adjusted by weighting it for complexity (of code, data, algorithms), product domains, timing and storage constraints, reuse percentage, distribution degree and reliability requirements. There has been significant industrial uptake of Function Points because they are relatively simple to measure and calculate, are generic, and available early in the process. Function Points have been criticised for inconsistent counting rules but there have been attempts to correct this problem (Verner and Tate 1992). Whilst they have been demonstrated to be useful in some domains, there has been little analysis of the variation of the measure when applied to different domains and applications (such as function-strong versus data strong environments).

The accuracy and maturity of cost estimation models is, to say the least, poor. A number of independent evaluations like those by Kemerer (Kemerer 1987), Genuchten (Genuchten and Koolen 1991), Heemstra (Heemstra 1992) and Pengelly (Pengelly 1995) have all reported mixed successes with the accuracy of these models. They warn that even with local calibration cost estimation models should only be used as a check for manual estimates.

A general problem with the majority of published cost models is that they deal with change implicitly rather than explicitly. For example, Subramanian demonstrates that, whilst the factors on which most estimates are based are highly vulnerable to change, conventional cost models give discrete values and no indication of likely error margins (Subramanian and Breslawski 1995). The inevitable changes make it unreasonable to expect accurate estimates early in the process it is therefore more desirable to have models that produce estimates with stated bounds of accuracy.

There has also been a tendency in the literature to treat cost estimation and cost control as two separate and distinct problems. The results of Lederer's survey into the causes of inaccurate cost estimates (Lederer and Prasad 1995) demonstrated that whilst managers perceived the problem as user sourced changes in requirements, the fundamental problem was inadequate monitoring and control of development and change.

Abdel-Hamid has suggested a maintenance metaphor of adapting, correcting and perfecting estimates to overcome some of these problems (Abdel-Hamid 1993). Current estimating methods are again criticised for being discrete, whereas the software artefact itself is seen to change continuously. He argues that adaptive, corrective and perfective maintenance should take place on an estimate to increase its accuracy and act as a focus for alternative decision strategies. He also highlights the need for more understanding of the consequences of rework and requirements change as well as just productivity.

In response to these problems, Abdel-Hamid proposes the use of systems dynamics simulation models as having significant benefits over current analytical models.

- (i) Their causal structure provides a superior explanatory power - regression models do not explain why or under what conditions a relationship exists.
- (ii) The solution of analytical models usually specifies only the terminal or steady state that eventually results from changing the values of the controlled variables. Simulation exposes continuous transitions.
- (iii) They can support controlled experimentation.

Their continuous and self-correcting nature means that systems dynamics techniques appear to be a promising approach to the construction of predictive software models. However, like conventional parametric models they remain inherently constrained by our inadequate understanding of fundamental product and process inter-relationships.

Kitchenham makes explicit the inherent problems faced by all predictive models of the software process; *"Software development, like economics and management science, is a complex, non-linear adaptive system and as such any attempt to estimate project effort or timescales is likely to result in very inaccurate estimates"* (Kitchenham 1998). She concludes, *"Project managers need to concentrate more on managing estimate risk than looking for a magic solution to the estimation problem."* These observations underpin the techniques that we have developed during this research.

## 2.4.2 Modelling Processes

There is general acknowledgement that theoretical models of software development do not reflect actual processes seen in industry (Curtis, Kellner et al. 1992). Experiences in both lifecycle and cost modelling (Madhavji 1991) have highlighted the inadequacy of large-grained techniques to capture and convey the low-level characteristics of the software process, such as:

- (i) the process steps required to resolve customer reported software problems;
- (ii) triggering and terminating conditions of an activity;
- (iii) the state of a product component before, during, and after a process step;
- (iv) the notational, methodological, operational and other tools to be used in different process steps;
- (v) the inputs and outputs of an activity and the sources and destinations of the data;
- (vi) the manner in which data flows from one activity to another;
- (vii) the roles played by humans in the process;
- (viii) the constraints on the process steps;
- (ix) how communication among humans is supported;
- (x) where parallel and sequential process steps exist.

The aim of modelling is to: (i) provide a framework for understanding, experimenting with, and reasoning about the process; (ii) facilitate process guidance, enforcement and automation; (iii) provide a basis for process management; and (iv) allow packaging of experience for future applications. Process modelling therefore has the potential to be a key factor in improving software development effectiveness (Krasner, Terrel et al. 1992). As Bandinelli writes, *“Improvement must be based on a solid methodological and organisational background to guide and direct assessment and decision making processes effectively”* (Bandinelli, Fuggetta et al. 1995).

Process models can be seen to reflect certain general constructs (Curtis, Krasner et al. 1987):

- *Agent* - an actor (human or machine) who performs a process element;

- *Role* - a coherent set of process elements to be assigned to an agent as a unit of functional responsibility;
- *Artefact* - a product created or modified by the enactment of a process element.

A large number of process modelling approaches have been suggested. They differ in nature as (i) *prescriptive* – requiring that process should be performed in a particular way, (ii) *proscriptive* – requiring that a process should not be performed in a particular way, or (iii) *descriptive* – describing the way in which development is actually conducted (Madhavji 1991).

Curtis has classified approaches to process modelling according to four perspectives (Curtis, Kellner et al. 1992)

- (i) *Functional* - represents what process elements are being performed, and what flows of informational entities (e.g. data, artefacts, products) are relevant to these process elements.
- (ii) *Behavioural* - represents when process elements are performed (e.g., sequencing) as well as aspects of how they are performed through feedback loops, iteration, complex decision-making conditions, entry and exit criteria etc.
- (iii) *Organisational* - represents where and by whom process elements are performed, the physical communication mechanisms used for transfer of entities, and the physical media and locations used for storing entities.
- (iv) *Informational* - represents the informational entities (e.g. data or products) produced or manipulated by a process.

Each perspective has particular advantages for different audiences (engineers, technical management, general management, quality assurance and customer). It is therefore generally accepted that no single modelling perspective can capture the intricacy of the complex software process. Hence, other research has been focused on the evaluation and integration of process modelling techniques (McChesney 1995).

Rombach (Rombach and Verlage 1995) has performed a comprehensive review of current research and future research directions in process modelling. He finds that despite the large number of different representations few have empirical evidence of successful application - representing the relative immaturity of the discipline. As Rombach states “*Most software process representation languages do not reflect*

*practical needs, are not documented well and are not supported by appropriate tools. Only a few languages have been used in real pilot projects, and only few tools and process-sensitive software engineering environments can be bought.”*

One promising application of process modelling is in helping to define measurement needs since the meaning of a metric is dependent on the process in which it operates. Shepperd (Shepperd 1992) argues for product measures embedded in process descriptions to (i) aid application, (ii) aid reuse of successful applications, (iii) support training and educational aspects of introducing measures, (iv) aid potential for powerful advanced tools and environments with automated collection, and (v) aid understanding. By considering *when* a product is measured, as well as *how* it is to be measured, it is hoped that some of the problems of measurement research (such as misapplication, problems of validation, incorrect usage) can be overcome.

Process modelling can play a significant and useful role in capturing and developing our understanding of the mechanisms by which software is created. It is a rapidly expanding research area and, like software measurement, it is inherently difficult to assess the contribution of each technique to the empirical problem domain. There is a need for improved frameworks and techniques as a basis for evaluating different approaches and integration of successful elements. There is also a need for more evaluation.

### **2.4.3 Modelling Change**

There are relatively few examples of quantitative models of software change. The majority of related work has been in the field of software maintenance, largely for the prediction of maintenance effort.

The performance of predictive models of maintenance effort is generally poor. Jörgensen observed that most organisations continue to rely on expert judgement or function-point analysis (Jörgensen 1995). He compares a number of different approaches to estimating maintenance effort including regression analysis, neural networks and pattern recognition - all using a measure of size based on lines of code (LoC):

$$\text{Size} = \text{LoC Inserted} + \text{LoC Updated} + \text{LoC Deleted}$$

The author highlights the critical failures of this measure. First, that it does not reflect task characteristics such as the change in product functionality or quality. Second, it



does not reflect actual maintenance tasks such as design, test and documentation. Despite these problems, Jørgensen proceeds to use LoC as a measure of all these factors (with mixed success). This paper unintentionally demonstrates the need for improved measures of change ‘size’ as a basis for estimating effort.

A more convincing approach is presented by Evanco (Evanco 1995) who attempts to model the correction effort required to (i) *isolate* the cause of a fault, and (ii) *fix* the fault once it is isolated. This correction effort information can be used in two ways: *Predictive* - to evaluate effort and resources required to complete the task; and *Prescriptive* - to evaluate the impact of specific software changes on the effort required to correct faults potentially leading to design or implementation modifications. Evanco’s model is based on three categories of variables relating to fault locality (‘spread’ of the fault through the software), software characteristics (structural and complexity features of the code) and cumulative changes to the software (based on a questionable assumption that more changes imply increased familiarity and therefore less correction effort).

Current attempts to model the maintenance or fault correction tasks are inadequate. This is analogous to the state of estimation models of development effort. Their inaccuracy prevents meaningful results from being produced outside of the original dataset.

One of the few models of maintenance suitable for management decision making outside of cost estimation is presented by Papapanagiotakis and Breur (Papapanagiotakis and Breuer 1994) who use a service agent based model of maintenance. A queuing network is used to provide information on the consequences of assigning particular individuals to particular kinds of tasks (in interactive mode) or adopting certain general management strategies (in automatic mode). The model appears to be one of the most comprehensive in its appreciation of the separate tasks of change.

The majority of measurement and modelling research largely concerns ‘waterfall’ single-team developments. The work of Aoyama at Futjitsu (Aoyama 1993) reports on the successful use of concurrent processes to reduce software lead-times by 75%. This paper is notable in that it indicates the absence of research into advanced techniques for managing software projects that are concurrent and distributed (between development teams). The work suggests that there is a great deal of research yet to be done in the control of these complex processes.

There have been other attempts at developing non-linear models of change. Olsen (Olsen 1993) proposes a management model of software change based on dynamically overloaded queues. The model is one of the few that treat development and maintenance activities as essentially the same thing (i.e. ‘change’). Olsen uses a fluid approximation model that separates changes that create *demands* on staff (like requirements, reorganisations, market shifts etc.) from the *services* that they can perform (like documentation, coding, testing, inspecting and defect resolution). The model presented is over-simplistic, but does serve to demonstrate project phenomena such as the ‘mythical man-month’ (Brooks 1975) and demonstrates queuing models to be a useful approach to change modelling.

In a similar manner, Hansen (Hansen 1996) has simulated the software process using a dynamic model of maintenance tasks. Hansen argues for explicit rather than implicit accounting of rework in estimation techniques. Simulation can aid reasoning about basic feedback effects of change on the overall project outcome but, again, the model given is simplistic in terms of both its parameters and underlying assumptions.

A more fundamental approach has been taken by Lehman (Lehman 1995) who is performing a new investigation of the role of feedback in the software process (he acknowledges that the importance of feedback has been apparent at least as far back as 1969). Lehman’s earlier investigations into software system stability and evolution (Section 2.1) identified both positive and negative feedback. Positive feedback arises from ambitious growth targets. Negative feedback arises from techniques such as inspections, reviews, prototyping and measurement. The observed phenomena of oscillation in system size were hypothesised to be a symptom of excessive positive feedback leading to periods of instability (a consequence of human nature in evolution decisions). Lehman asserts that pre-existing negative feedback mechanisms act as a limiting factor in improvements in the forward path of a software process, formulating his FEAST (Feedback, Evolution, And Software Technology) conjecture; “*As a multi-loop feedback system the E-type software process will display global invariance characteristics.*” The intention is to use process modelling and systems dynamics techniques to capture properties of these feedback mechanisms. The FEAST conjecture could provide a valuable basis for quantitative models of software change.

Many of the problems of change are due to the limitations of current software development technology in supporting change handling and providing information for

change decision-making. There have been attempts to create new paradigms of software development that are more amenable to assessment and accommodation of change.

A research strand of the PROTEUS project (Sugden and Strens 1996) highlighted the importance of having software-engineering environments that facilitate effective change. The PROTEUS team stress the importance of assessing the suitability of approaches to requirements engineering and methods of systems design in dealing with requirements instability. An outcome of the project was the development of criteria that can be used to evaluate a method's contribution to minimising the adverse impact of change (Strens and Sugden 1996). Fundamentally, however, PROTEUS does little to address the wider problem of change handling.

There have been less 'revolutionary' attempts to improve an environment's capability for change handling particularly to reduce problems of software maintenance. This work is largely driven by the need to provide for evolution in systems definition and record more contextual information about software designs. Perhaps the most notable is the work of Baxter (Baxter 1992) who explains the loss of two important kinds of design knowledge: (i) the problem specification, and (ii) the design justification. This has led to techniques to record (or even re-constitute) design rationale and capture resultant changes over time (for example, the work of Lanubile (Lanubile and Visaggio 1995)).

A number of other tools and environments have been proposed such as that by Ajila (Ajila 1995) who has investigated the information needed for impact analysis and change propagation. There is also continuing research to improve configuration technology to support more advanced process models (Bersoff and Davis 1991) and to give guidance in change handling (Gulla, Karlsson et al. 1991). A review of these environments is presented by Fernström (Fernström, Narfelt et al. 1992).

One other major research area involving software change is that of software reuse. Both economic and technical benefits have motivated the industrial uptake of reuse techniques including reduced development timescales and effort, increased system reliability, and simplified estimation. Despite the problems of classifying, cataloguing and retrieving software components (Mili, Mili et al. 1995), there are many recorded case studies identifying that reuse benefits are achievable. A good example is Hewlett-Packard (Griss 1993) where product costs have been reduced by a sustainable 10 to 12 percent (with defect rates dropping to 10 percent of their former levels and long-term maintenance costs dropping by 20 to 50 percent).

## 2.5 Managing Software Change

There has been significant research aimed at capturing high-level mechanisms for achieving process improvement. The basic assumption behind the majority of process improvement work is that a good process is instrumental in producing a good product. However, practical experience has shown that progress at transferring new tools and methodologies into industrial practice is being limited by largely non-technical factors (Basili and Musa 1991). Much of this work has its origins in management research and its practical application in fields such as manufacturing. Concepts such as statistical process control (Deming 1986), continuous improvement (Imai 1986) and their associated techniques (such as Pareto analysis and control charts) have been applied to software development, with mixed success.

The need to evaluate an organisation's process capability has led to the development of capability models such as the Capability Maturity Model (CMM) developed by the Software Engineering Institute (SEI). The CMM is based on the desire to have a repeatable (and therefore predictable) process, i.e. a process free from variation. The CMM framework rates an organisation on one of five levels according to process capability.

Level 1: Initial (software process poorly controlled)

Level 2: Repeatable (management has basic control of software process)

Level 3: Defined (software process is standard and consistent)

Level 4: Managed (software process is quantitatively understood and controlled)

Level 5: Optimising (software process focuses on continuous improvement)

These levels are based on recommended practices in key process areas. Assessment is largely question driven, although there have been attempts to attach specific quantitative measures to each level such as that by (Pfleeger and McGowan 1990). The CMM and contemporary models (such as SPICE ISO/IEC 15504 (ISO/IEC 1996)) are serving to encourage (or force) organisations to evaluate their process maturity and improve it over time.

Another significant contribution to process improvement has been made by Basili et al. at the Software Engineering Laboratory (Basili and Rombach 1988). The TAME model (Tailoring A Measurement Environment) presents a framework to guide and support

process improvement. It is based on a four step Quality Improvement Paradigm (QIP) to (i) characterise current maturity, (ii) define quantifiable goals and improvement strategies, (iii) evaluate achievements on measures of performance, and (iv) package, disseminate, and reuse this experience within the organisation (Oivo and Basili 1992). A second thread is the use of the Goal, Question, Metric (GQM) paradigm that encourages measures to be specified in terms of goals (to achieve), questions (that need to be answered) and metrics (needed to answer the questions). GQM aids the focused use of measures and captures a measure's context to aid communication, and its suitability in each environment. The high-level TAME environment uses the concepts of QIP and GQM to capture process understanding and aid reuse between projects (Basili and Weiss 1984).

There are a number of published studies on measurement based process improvement, relatively few of which contribute (in research terms) anything more than a documented application of, and pragmatic advice on, metrics programmes. Some studies do indicate measures that have worked and given improvement in certain applications, for example the work at Bull Information Systems (Weller 1994). There are however a handful of notable exceptions:

- Hewlett-Packard (Grady 1990). Grady reports on the application of work product analysis for process improvement. The outputs of all developmental stages can be passed through analysis tools including: (i) *design* - Tom DeMarco's design weight measure is generated to compare module complexity versus system complexity; (ii) *code* - a graphical plot of McCabe's cyclomatic complexity measure provides visual impact of complexity; (iii) *test* - test coverage measures; and (iv) *documentation* - readability statistics (Flesch-Kincaid indices). The automated data collection and analysis techniques reduced the overheads of the measurement programme whilst keeping managers better informed of progress.
- Motorola (Daskalantonakis 1992). The work at Motorola is a good example of a carefully focused measurement programme. It emphasises the benefits of having fewer but more informative measures that are meaningful to the appropriate audiences (users, software engineers, senior managers, software managers and quality teams). The essence of a successful measurement scheme is captured by analysis of the different dimensions that have to be considered (metrics usefulness,

audience, goals etc.). An important feature was the comprehensive one-page ‘vital signs’ summary to management.

- Raytheon (Dion 1993). This report describes attempts to calculate the cost savings from the measurement programme. Like the majority of these studies the main benefits have come from reducing rework, but more notable is the account of less tangible rewards such as reduced pressure (hence less late nights and weekends worked) and improved communication. The paper describes how the claimed dramatic improvements (a two-fold increase in productivity and a \$7.70 return on every dollar invested) were calculated.
- Lockheed-Martin (Henry, Blasewitz et al. 1996). This study is very interesting in that it is one of the few to focus on implementing a measurement based maintenance process. Three main benefits are experienced. First, the impacts of different types of change are quantified and strategies for their management are in place. Second, the effectiveness of process phases can be quantitatively assessed (such as the ability to detect defects early in the lifecycle). Third, and most significantly, they quantify the impact of late specification changes on the software product and the maintenance process.

Software process improvement has become a large industry in its own right, with many academic institutions having turned into consultancies. An unfortunate consequence is the amount of hype surrounding different techniques for process improvement that can conflict with attempts to assess the true nature of benefits achieved. A general observation about the majority of reported studies of measurement-driven process improvement is that the claimed benefits are largely centred on the reduction of rework. Techniques for more fundamental process improvements remain unproven.

## **2.6 Summary of Research Issues**

This chapter has explored research on the measurement, modelling, and management, of software change. We have found that the fields are more notable for their limitations rather than their achievements, particularly if judged by the rate of industrial uptake on new methods. More specifically, the specific nature and problems of time-constrained software development are not described in the published literature.

Three pieces of research are of particular relevance to our investigation. First, Lehman argued that the (E-type) software process forms a dynamic feedback system and needs to be managed as such (Lehman 1998). Second, Abdel-Hamid used system dynamics to study software process feedback and illustrated the limitations of static models of software development (Abdel-Hamid and Madnick 1991). Finally, Kitchenham argued that, since the software process is a complex, non-linear, adaptive system, there are inherent limits on the potential accuracy of our predictions and, as such, we need to adjust our methods and expectations accordingly (Kitchenham 1998).

These insights raise a number of issues for further investigation and form the starting point for the research presented in this thesis. These are:

- (i) *Measurement*. The need for time-constrained software processes to operate at their limits makes them increasingly exposed to the risks of change. Are conventional approaches to measurement suitable for time-constrained software development? (Chapter 3);
- (ii) *Observations*. The specific nature of time-constrained software development has not been studied before as a topic in its own right. How do time-constrained developments differ from their conventional counterparts? (Chapter 4);
- (iii) *Problems*. The academic literature contains little evidence of the practical consequences of reduced development lead-times. What are the specific problems of time-constrained development? (Chapter 5);
- (iv) *Modelling*. The majority of predictive models are based on single-project waterfall developments. Are conventional approaches to modelling suitable for time-constrained lifecycles? (Chapter 6); and
- (v) *Management*. The software process is a dynamic feedback system and needs to be managed as such. What strategies are suitable for managing time-constrained software development? (Chapter 7).

In the chapters that follow, we look at each of these research issues in turn.

## Chapter 3:

# Measuring Time-Constrained Development

---

### 3.1 Introduction

In the previous chapter, we highlighted areas of theoretical and practical weakness in software measurement, and consequent problems for software engineering as a whole. The specific nature and problems of time-constrained software development were, however, not evident in the literature.

In this chapter, we explain the problems of measurement using *Work Breakdown Structures* (WBS) (Section 3.2). These problems led us to introduce a fine-grained measurement technique, called the *Process Engineering Language* (PEL), as a basis for understanding the nature problems of time-constrained development (Section 3.3). We use GQM to describe our introduction of PEL into Rolls-Royce Engine Control System projects as part of a full-scale measurement programme (Section 3.4). We explain the PEL approach, and tools used, for the collection and analysis of *effort* (Section 3.5), *defects* (Section 3.6), and *size* (Section 3.7) metrics. In each case, we show how PEL provides benefits over conventional approaches to measurement.

### 3.2 Work Breakdown Structures (WBS) and their Problems

The most pressing measurement problem was the identification of where effort is being applied in a process and thus where costs are being incurred. This information has historically been organised and collected using a *Work Breakdown Structure* (WBS). A WBS decomposes projects into a set of tasks using a hierarchy or ‘tree’ structure, as illustrated in Figure 2. The WBS is broken down for each team (e.g. *Systems*, *Software*), for each delivery (e.g. *D1.0*, *D2.0*) and finally each activity (e.g. *Code*).

A WBS serves four main purposes. First, it *describes* the process as a set of tasks that must be achieved to complete the project. Second, it allows managers to *budget* time and expenditure across these tasks. Third, it allows managers to *measure* the time spent by engineers on each individual task. Finally, it allows users to *control* the project by examining the task effort expended against the budget.



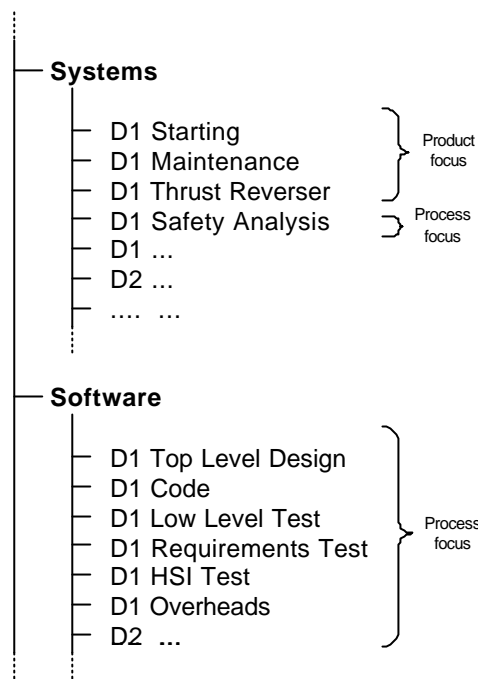


Figure 2: Example Work Breakdown Structure (WBS)

There are, however, a number of problems with conventional WBS with respect to measuring effort and cost.

(i) *Confusion Between Product and Process.*

The first problem is the confusion between product and process. The natural decomposition of products into components and sub-components makes WBS good at capturing product-related information. However, they are less appropriate for gathering process-related information since processes can relate to many or all products (Hitchins 1992).

This confusion can be found in the breakdown structures within Rolls-Royce (Figure 1). The *Systems* team needs to understand the cost of each functional area of the system (e.g. the cost of the Thrust Reverser) and thus decompose their WBS by product. In contrast, the *Software* team is more concerned with understanding the cost per lifecycle phase (e.g. Coding, Testing) and thus decompose their WBS by process. The combined WBS therefore collects data across a mixture of product and process descriptions.

However, whilst a WBS allows the total cost of specific projects to be established, it is extremely difficult to find the cost of the constituent software development activities. For example, the systems team does not know the cost of their review process and the

software team does not know which functional areas of the system take longest to code. The WBS therefore loses detail that is important, if not critical, to project managers.

In order to collect more ‘fine-grained’ (i.e. detailed) measurements, it would be necessary to decompose the WBS to capture every combination of product and process (for each project and delivery). For example, a project would have to be broken down into *Delivery1 Thrust Reverser Coding*, *Delivery 1 Thrust Reverser Low Level Testing* and so on. However, even a small parallel development of 3 projects, each with 5 stages (e.g. *D1*, *D2*), 5 product functions (e.g. *Thrust Reverser*) and 5 work-products (e.g. *Requirements*, *Design*) would consist of some 375 (3x5x5x5) task descriptions. Clearly, for all but the most basic projects, such structures would soon become unmanageable; The benefits of accurately allocating an engineer’s time between WBS activities would soon be outweighed by a vastly increased collection and administration overhead.

(ii) *Lack of Flexibility for Analysis*

The second problem is that WBS can only support the aggregation of effort data within the structure chosen. For example, since the WBS in Figure 2 (Page 49) is decomposed according to project stages, it is possible to discover the total effort of Systems and Software teams by aggregating the effort for each branch in the tree. Finding the total cost of a particular product is considerably more difficult since the data may be split across many different branches, if it is collected at all.

(iii) *Inconsistent Language Set*

The third problem is that the high degree of variability in the terminology used and structures created. This means, despite informal ‘cut and paste’ reuse, the design of a WBS remains largely at the discretion of the project manager. Any differences in naming and granularity make it difficult to perform ‘like-for-like’ comparisons between (and even within) projects. For example, it is commonplace to find task descriptions such as ‘*Low Level Test*,’ ‘*Low Level Testing*’ and ‘*Testing*.’

(iv) *Ambiguous Task Descriptions*

The fourth problem is that, even where terminology is used consistently, task descriptions are not always meaningful or sufficiently descriptive. For example, a WBS in Rolls-Royce Marine Power included the tasks ‘*Dev/proj queries*,’ ‘*NDE proc. IMI/2 NDE proc*,’ ‘*RPV body*’ and ‘*Arisings*.’ Their meanings are not clear to anyone outside of the project or, indeed, to someone who in future might have to refer back to data

from this project. The lack of rigour in the usage of WBS can thus mean that the data collected is limited, subject to interpretation, and comes with a ‘shelf-life.’

The way a WBS is structured therefore has a major influence on the way data is collected and, critically, on the types of questions that can later be asked about the process. It was therefore evident that WBS approaches could not capture the level of detail that we needed to understand the complex behaviour of time-constrained software projects.

### **3.3 The Process Engineering Language (PEL)**

#### **3.3.1 Background to PEL**

The Process Engineering Language (PEL) was devised in 1994 by Graham Clark, BAE SYSTEMS. Since 1995, the author has worked with Graham Clark to use PEL to provide a consistent measurement philosophy between the two companies (Clark and Powell 1999). The specific contribution of the author has been to introduce a PEL-based measurement programme (Section 3.4) for the collection and analysis of *effort* (Section 3.5), *defects* (Section 3.6), and *size* (Section 3.7) metrics.

In Clark and Powell (1999), we explain the need for a structured language for process description and measurement: *“Most projects have their own Management Information System vernacular. Each has a unique work breakdown structure, organisational breakdown structure, product breakdown structure and particular philosophy for distributing budgets. Some projects collect achievement against budgets, others do not, some produce vast amounts of data, some hardly any. Data is collected against the products on some projects, and processes on others; some pieces of data are combined with others, some are not. There is little intra-project exchange of data and very little inter-project correspondence. Projects, by their nature, tend to be selfish; they are inclined to have no unifying portfolio between them, no reciprocity of information”* (ibid).

### 3.3.2 Description of PEL

It was clear that inconsistencies in process description presented a significant barrier to understanding time-constrained development. PEL overcomes these problems by using a structured language to support the description, measurement and control of a process.

Like any language, PEL consists of a lexicon and a grammar.

The first component of PEL is the *lexicon*. It provides a constrained vocabulary of terms (verbs and nouns) that are used to describe a process. The PEL lexicon is divided into four dimensions: *Actions*, *Stages*, *Products* and *Representations*.

**Actions** are the generative, evaluative and supportive verbs that describe what we do, for example: *Produce*, *Review*, *Manage*.

**Stages** are time-based (milestone or cost) breakdowns of logically sequenced partitions of work, for example: *Project A*, *Delivery D2*.

**Products** are the configurable physical or functional components or systems that are produced, for example: *Engine Control System*, *Thrust Reverser*.

**Representations** are the (intermediate) outputs or work-products of a process, for example: *Requirements*, *Designs*, *Code*.

The lists of Actions, Stages, Products and Representations serve as a vocabulary for process description and measurement. The author defined a vocabulary for Rolls-Royce Controls Software projects using terms from existing WBS, procedural documents, and in meetings with engineers. The resulting lexicon is shown in Figure 3.

<u>Action</u>	<u>Stage</u>	<u>Product</u>	<u>Representation</u>
Produce	Project A	ActuatorP30Gain	Code
Perform	A5.0	ActuatorValidation	ConfigurationManagement
Understand	A6.1	AircraftComms	FunctionalRequirements
Review	A6.1.1	AirDataAcquisition	InterfaceRequirements
Manage	Project B	AirDataMaintenance	LowLevelTest
Attend	B3.1.1	AirDataSelection	Plans
Illness	B3.2	ARINCAFIInputs	ProceduralDocuments
Holiday	B3.3	ARINCInputs	RequirementsTest
	B4.0	ARINCOutputs	SoftwareRequirements
	Project C	BleedValveControl	SPARK
	C2.0	BoosterBleedValveControl	Static Analysis
	C2.1	BoxManagement	SWRDTraceability
	Project D	BoxTemperatures	SystemAcceptanceTest
	D1.0	ControlLaws	SystemConcept
	Course or Seminar	ControlLaws -BBVControl	TimingAndMemoryAnalysis
	Debriefs	ControlLaws -FanDamage	TopLevelDesign
	Improvements	ControlLaws -FuelDemand	Traceability
	Methods	ControlLaws -LightUp	
	Etc...	ControlLaws -Limiters	
		Etc...	

Figure 3: Example Process Lexicon (PEL)

The second component of PEL was the *grammar*. The grammar combines the four dimensions of the lexicon to form an activity description for which various properties (e.g. *effort*, *size*, and *defects*) can then be measured. For example, the line in Figure 4 shows how we can form a description of the *Effort* (in person-days) that was required to ‘*Produce Delivery B3 ARINC Code.*’

The descriptions of process and product are therefore combined since each sentence captures both the process performed (‘*Produce B3.3 Code*’) and the product created (‘*AircraftComms Code*’). A critical feature of PEL is therefore its ability to capture both the measure and its *context*. This is particularly important in time-constrained software development where the performance of an activity needs to be considered in the context of both simultaneous and subsequent activities (as we will explain in Chapter 4).

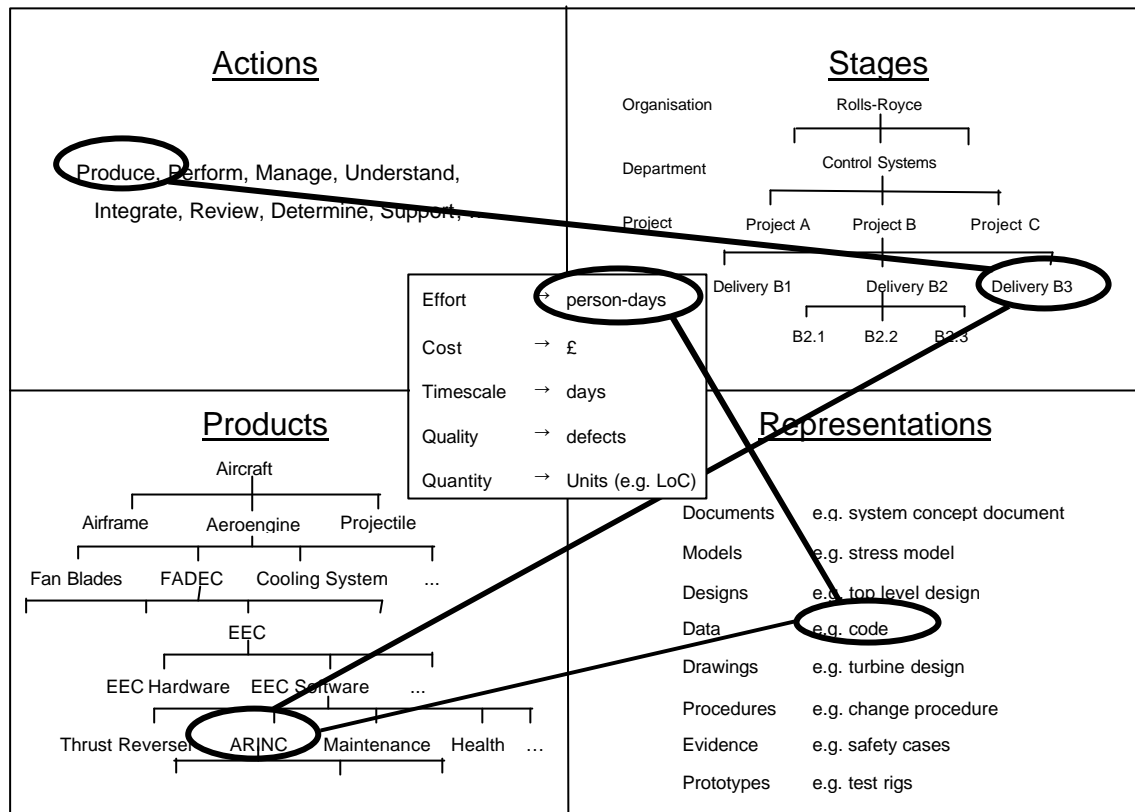


Figure 4: Measurement using PEL

The benefits of PEL arise because it allows us to query any dimension (or term) within our activity descriptions.

The dimensions of *Stage* and *Product* consist of hierarchies through which information can be aggregated (shown in Figure 4). For example, we can view *Product* effort at the levels of function (e.g. *Thrust Reverser*), system (e.g. *Engine Controller*), or systems (e.g. *Aeroengine*). Likewise, we can view the *Stage* effort at the levels of task, delivery, project or programme.

The dimensions of *Action* and *Product Representation* describe operational activities as performed by engineers, such as 'Produce Code' or 'Review Design.' We can therefore view the effort of each lifecycle phase (e.g. *Code* and *Low Level Test*). Alternatively, we can view the relative spend of each activity, for instance, evaluating the time spent to *Produce* or *Review* a particular product.

The PEL approach to process description and measurement overcomes the problems of WBS described in Section 3.2. It does this by (i) *integrating* product and process views in task descriptions to ensure that valuable data is not lost, (ii) enabling users to

*aggregate* data in ways not limited by the way a WBS is structured, (iii) supporting *consistency* in process description and measurement across projects, teams and organisations, and (iv) capturing *meaning* by ensuring tasks are descriptive and easily understood. In the following sections, we show how fine-grained measurements using PEL are necessary for understanding the nature of time-constrained development.

### 3.4 The Measurement Programme

The author has been involved in the introduction of measurement programmes into aeroengine controls projects in Rolls-Royce since 1995 (Powell 1995; Powell 1997; Powell 1999). The author's work to introduce PEL into the measurement programme began in early 1996.

The philosophy behind the measurement programme was to use a few simple metrics that were easy to collect and analyse. The Goal Question Metric (GQM) approach (Basili 1988) suggests the metrics should be defined in a top-down manner starting with organisational goals. However, in practice, our measurement programme evolved using a mixture of top-down, bottom-up, and trial-and-error, approaches. GQM was instead more useful as means of documenting the measurement programme (Powell 1997). The author's basic GQM set for time-constrained development is shown in Figure 5.

The four goals, of reducing costs and lead-times, whilst increasing reuse and quality, were chosen for their simplicity and importance in time-constrained development. The goals are broken down into questions that each evaluate an important characteristic of time-constrained development and indicate if the high-level goals are being met. These questions are then answered using a basic set of metrics (identified by a reference in Figure 5) to capture: *effort*, *size*, and *defects* (Figure 6).

These metrics were gathered as a by-product of three collection tools (Cost Booking System, Rolls Change Control tool, and the Comparator tool) introduced during this research to collect and analyse *effort* (Section 3.5), *defect* (Section 3.6) and *size* (Section 3.7) metrics.

Goal	Questions	Metrics	Indicator of:
G1: Reduce cost	Q1.1: What is the process effort/cost?	<i>M1</i>	Relative/comparative effort/cost per process
	Q1.2: What is the effort productivity?	<i>M3/M1</i>	Productivity per process
G2: Reduce lead-times	Q2.1: What is the process lead-time?	<i>M2</i>	Comparative process lead-times
	Q2.2: What is the lead-time productivity?	<i>M3/M2</i>	Comparative lead-time for system size
G3: Increase reuse	Q3.1: What is the percentage reuse?	<i>M4</i>	Level of reuse
	Q3.2: What are the cost savings of reuse?	<i>M1, M4</i>	Bottom-line savings attributable to reuse
G4: Increase quality	Q4.1: Where were defects introduced, detected and fixed?	<i>M5.1</i>	Quality of development and testing process
	Q4.2: What types of defects were introduced?	<i>M5.2</i>	Nature of problems
	Q4.3: What was the severity of the defects?	<i>M5.3</i>	Impact of each problem
	Q4.4: What was the cause of the defects?	<i>M5.4</i>	Learning from mistakes

Figure 5: Goals Questions Metrics for Time-Constrained Development

	<u>Metric</u>	<u>Tool</u>	<u>Frequency</u>	<u>Indicates</u>
<b>Effort</b>	<i>M1</i> : Activity effort (person-hours)	Cost Booking System (CoBS)	Daily	Product and process effort (implicit cost)
	<i>M2</i> : Activity duration (time)			Process lead-times
<b>Size</b>	<i>M3</i> : Product size (LoC)	Comparator Tool	Per Change	Product size and complexity
	<i>M4</i> : Reuse between deltas (%)			Level of work-product reuse
<b>Defects</b>	<i>M5.1</i> : Defect count	Rolls Change Control (RoCC) System	Per Anomaly	Product and process quality
	<i>M5.2</i> : Defect type			
	<i>M5.3</i> : Defect severity			
	<i>M5.4</i> : Defect cause			

Figure 6: Metrics for Time-Constrained Development



## 3.5 Measuring Effort using PEL

### 3.5.1 Collection of PEL Effort Data

The first strand in the measurement programme was to collect PEL effort (and implicitly cost and timescale) metrics. In September 1996, the author commissioned a prototype tool based on PEL that was used by part of the software team on a pilot study. The prototype enabled engineers to select from four pull-down menus corresponding to *Action*, *Stage*, *Product*, and *Representation*, to form a PEL description of the task performed. These task descriptions and effort bookings were stored in a database and used to generate a standard weekly timesheet for entry into the Rolls-Royce payroll system.

The immediate benefit of the tool was to automate the previous paper-based timesheet system (which had been time-consuming to complete and prone to manual errors). However, as the scope of the trial was extended to include more teams, the limitations of the prototype became apparent. As the scope and thus size of the lexicon grew, so too did the length of the menus. Faced with over 60 products to choose from, the engineers soon found the lists unworkable and mis-bookings started to occur. Furthermore, since the tool did not constrain the tasks that could be booked, users could allocate time to any combination of terms whether valid or invalid. Hence, the effort data it collected could not be relied upon. Nevertheless, the prototype was successful in demonstrating the value of PEL and justified corporate funding for the development of a full production version.

In October 1997, the Rolls-Royce BR700 Controls Methods team, in conjunction with the author, set out to design a replacement tool that could overcome the problems of the prototype, whilst being robust enough for large-scale application. The PEL '*Cost Booking System*' (CoBS) tool was developed and introduced in January 1998 (Hayes 1998).

The CoBS tool (shown in Figure 7) enables engineers to describe the task performed by selecting from pull-down menus corresponding to the four PEL dimensions. Engineers then allocate their effort, to the nearest half-hour, to form a standard timesheet. This information is then be exported to traditional WBS tasks in the Rolls-Royce corporate

payroll system. CoBS also enables engineers to recall their previous bookings and avoid repeated entry of common tasks.

1. The user describes the task performed by selecting from the pull-down menus

2. The effort (in hours) spent on the task is then recorded.

3. Once accepted, the task is added to the timesheet.

4. The PEL timesheet forms a record of the effort expended and the process performed.

ACTION	STAGE	FUNCTION	PRODUCT	MON	TUE	WED	THU	FRI	SAT	SUN
Manage	BR715 Methods	HSI Rig	Improvements	0	2	3	0	0	0	0
Produce	D4.2	Aircraft Comms	Code	7	3	0	0	0	0	0
Review	BR715 Methods	ITV Rig	Documents	0	0	0	3	0	0	0
Produce	BR715 Methods	Software Tools	Documents	0	2	0	0	5	0	0
RoSEC Debrief										
Produce	BR715 Methods	ITV Rig	Documents	0						

AP 10 Week 2 Total 33

HAYES SOFTWARE METHODS AP= 10 WEEK=2

Figure 7: Cost Booking System (CoBS) – Effort Collection

The main benefit over the prototype was the incorporation of semantic constraints. The specification of these constraints, shown in Figure 8, allows the tool administrator to enforce (i.e. link or break) valid relationships between PEL items. These relationships make each pull-down list context sensitive to selections made in the other lists. In the example, the administrator has allowed the activities of *Review Requirements*, *Review Code*, *Review Design* and *Review Low Level Test*. If a user then selects ‘Review’ (from the Action list) only *Requirements*, *Design*, *Code* and *Low Level Test* will be available for selection in the cost booking screen. Similarly, the tabs on the menu allow the user to specify acceptable relationships between the other combinations of PEL dimensions.

The constrained PEL grammar thus gave a minimum subset of task descriptions that were: (i) appropriate to the user’s role, (ii) presently active, and (iii) semantically valid (i.e. meaningful). It also formed a description of the process that was remarkably consistent over time, changing only when stages of a project started or finished, when

new products were being developed or (less frequently) when the process itself was changed.

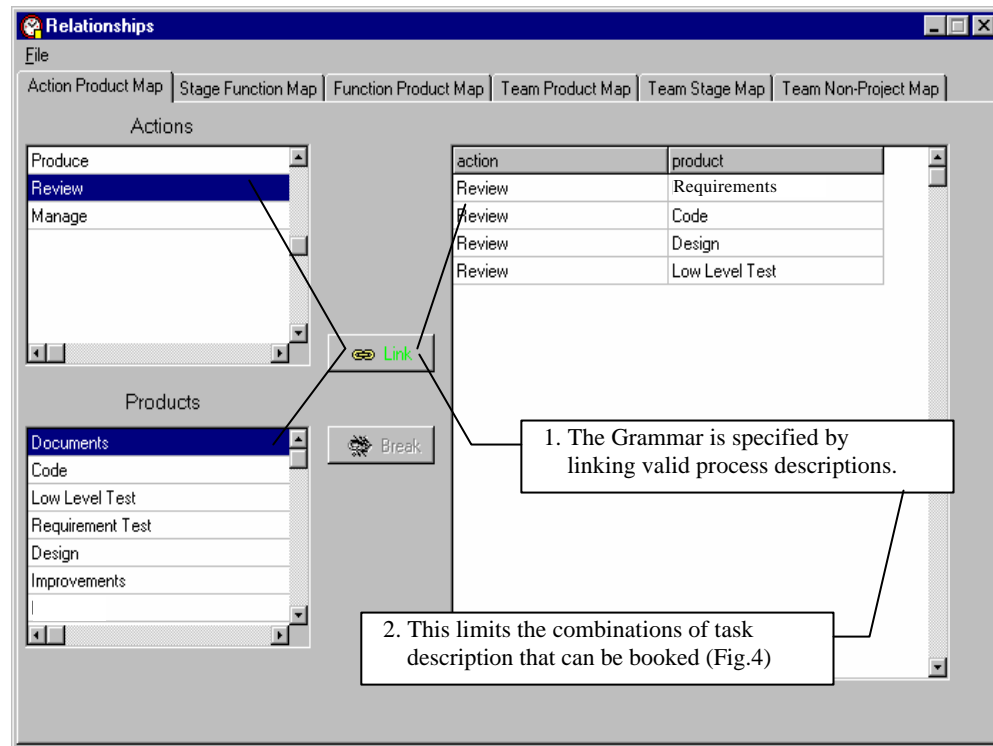


Figure 8: Cost Booking System (CoBS) – Semantic Specification

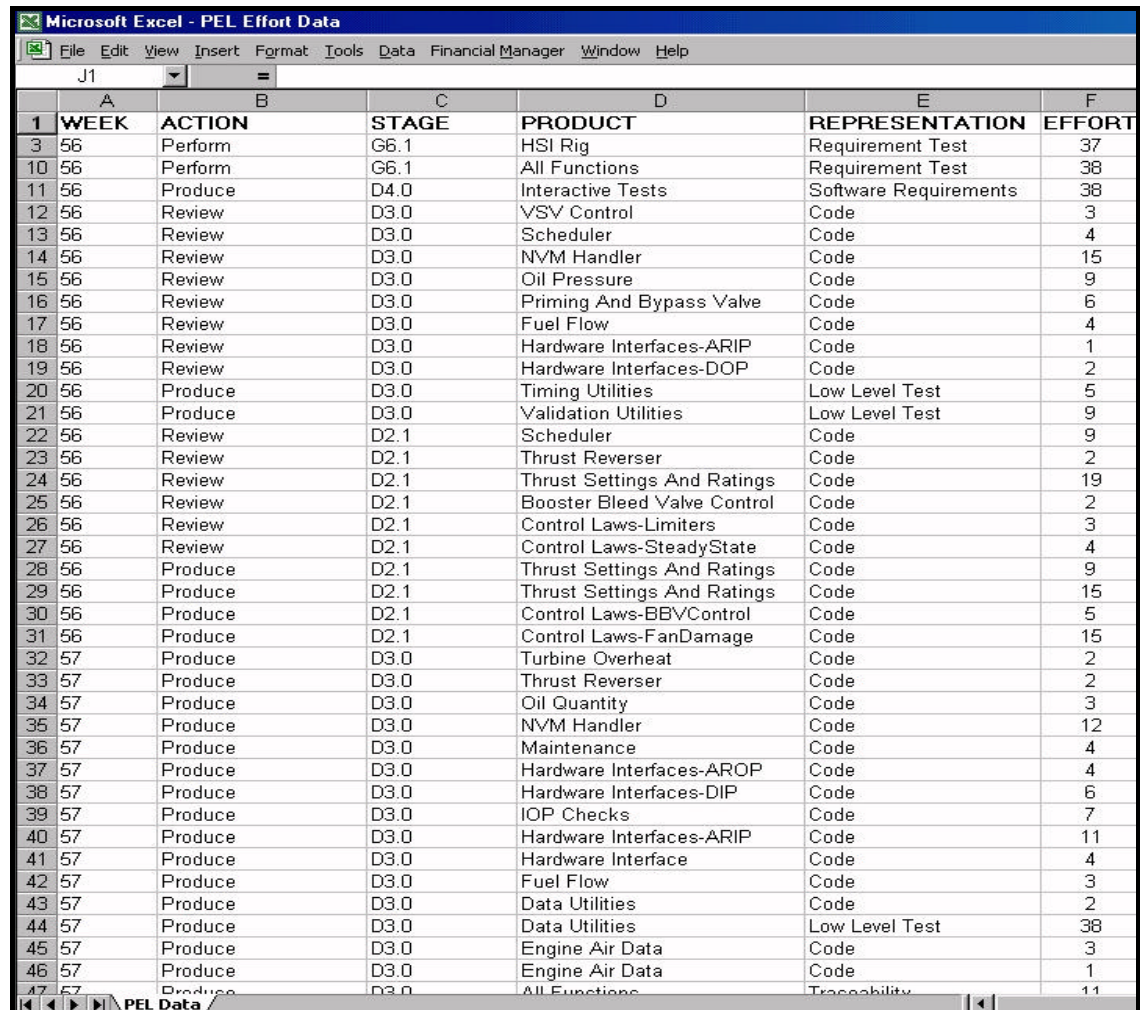
The PEL technique therefore provided a much finer level of data granularity, and more reliable recording, for no additional cost. The previous corporate system had used twelve-digit codes to represent each task in the WBS. If a team required more fine-grained measurements they had to subdivide the WBS by creating more codes. In contrast, PEL can support any combination of valid task descriptions from the lists of Action, Stage, Product and Product Representation. Hence, just four items in each of the PEL lists would require 256 (i.e.  $4^4$ ) cost codes to be raised in a WBS. Faced with fewer, standard, task descriptions, engineers' bookings became quicker and more reliable.

PEL approach also kept the control of metrics collection local to the team. Only those activities that need to be controlled at corporate level (e.g. major project milestones) are rolled-up into the corporate planning system. This mapping between PEL sentences and corporate cost codes means that this relationship can be administered locally and, thus, metrics information only exists at its most appropriate level in the organisation.

### 3.5.2 Analysis of PEL Effort Metrics

#### (i) Step 1 - Importing PEL Data into Microsoft Excel

The PEL data is analysed using database queries or, more commonly, by exporting it to a spreadsheet using a comma-delimited file. The data includes the PEL task descriptions and the effort booked (as shown in Figure 9).



	A	B	C	D	E	F
	WEEK	ACTION	STAGE	PRODUCT	REPRESENTATION	EFFORT
3	56	Perform	G6.1	HSI Rig	Requirement Test	37
10	56	Perform	G6.1	All Functions	Requirement Test	38
11	56	Produce	D4.0	Interactive Tests	Software Requirements	38
12	56	Review	D3.0	VSV Control	Code	3
13	56	Review	D3.0	Scheduler	Code	4
14	56	Review	D3.0	NVM Handler	Code	15
15	56	Review	D3.0	Oil Pressure	Code	9
16	56	Review	D3.0	Priming And Bypass Valve	Code	6
17	56	Review	D3.0	Fuel Flow	Code	4
18	56	Review	D3.0	Hardware Interfaces-ARIP	Code	1
19	56	Review	D3.0	Hardware Interfaces-DOP	Code	2
20	56	Produce	D3.0	Timing Utilities	Low Level Test	5
21	56	Produce	D3.0	Validation Utilities	Low Level Test	9
22	56	Review	D2.1	Scheduler	Code	9
23	56	Review	D2.1	Thrust Reverser	Code	2
24	56	Review	D2.1	Thrust Settings And Ratings	Code	19
25	56	Review	D2.1	Booster Bleed Valve Control	Code	2
26	56	Review	D2.1	Control Laws-Limiters	Code	3
27	56	Review	D2.1	Control Laws-SteadyState	Code	4
28	56	Produce	D2.1	Thrust Settings And Ratings	Code	9
29	56	Produce	D2.1	Thrust Settings And Ratings	Code	15
30	56	Produce	D2.1	Control Laws-BBVControl	Code	5
31	56	Produce	D2.1	Control Laws-FanDamage	Code	15
32	57	Produce	D3.0	Turbine Overheat	Code	2
33	57	Produce	D3.0	Thrust Reverser	Code	2
34	57	Produce	D3.0	Oil Quantity	Code	3
35	57	Produce	D3.0	NVM Handler	Code	12
36	57	Produce	D3.0	Maintenance	Code	4
37	57	Produce	D3.0	Hardware Interfaces-AROP	Code	4
38	57	Produce	D3.0	Hardware Interfaces-DIP	Code	6
39	57	Produce	D3.0	IOP Checks	Code	7
40	57	Produce	D3.0	Hardware Interfaces-ARIP	Code	11
41	57	Produce	D3.0	Hardware Interface	Code	4
42	57	Produce	D3.0	Fuel Flow	Code	3
43	57	Produce	D3.0	Data Utilities	Code	2
44	57	Produce	D3.0	Data Utilities	Low Level Test	38
45	57	Produce	D3.0	Engine Air Data	Code	3
46	57	Produce	D3.0	Engine Air Data	Code	1
47	57	Produce	D3.0	All Functions	Traceability	11

Figure 9: CoBS Analysis Step 1 - Importing PEL Data into Microsoft Excel

(ii) Step 2 – Creating a Pivot Table

The data can be queried directly or by using a powerful feature of Microsoft Excel called *Pivot Tables*. The Pivot Table function allows users to build interactive tables to rapidly summarise and filter lists of data. For example, the spreadsheet in Figure 10 shows a Pivot Table displaying the sum of *effort* for each *stage* over *time*. Using the Pivot Table Wizard (inset in Figure 10) the user can query any combination of PEL terms by dragging and dropping the PEL dimension to form a new Pivot Table query.

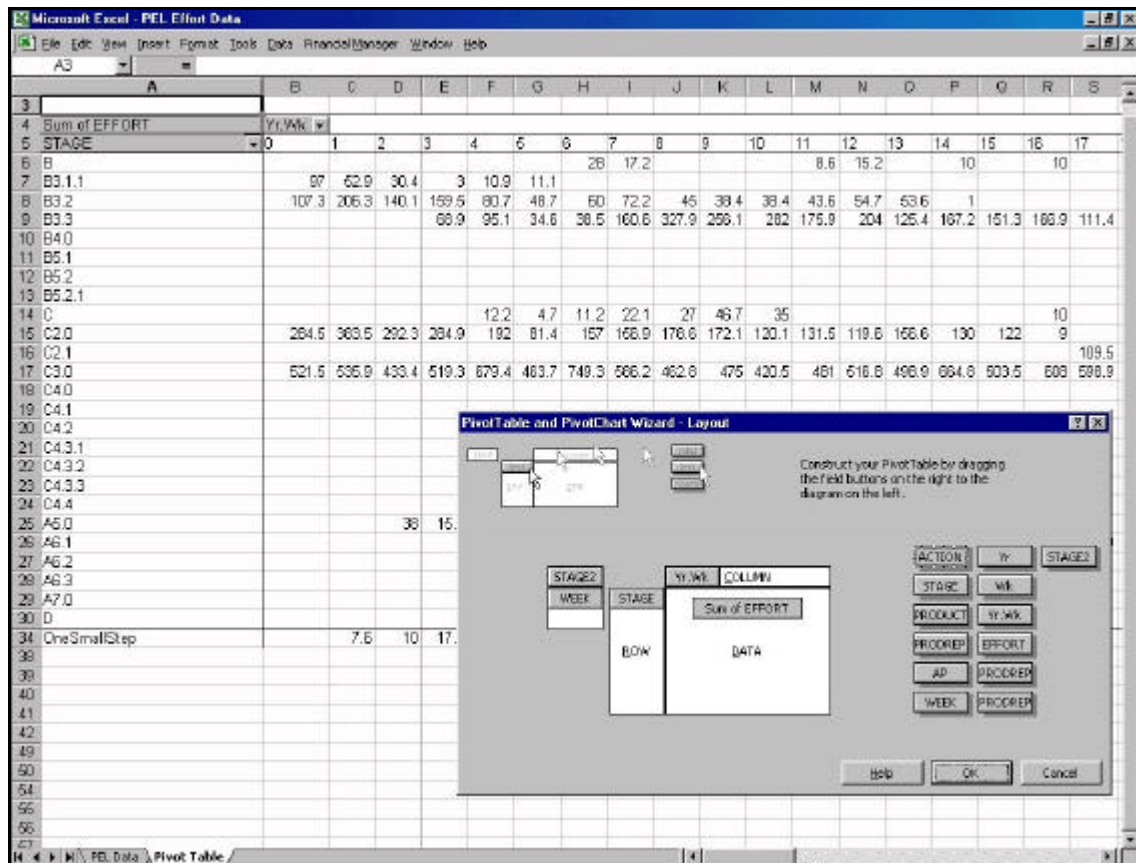


Figure 10: CoBS Analysis Step 2 – Creating a Pivot Table

(iii) *Step 3 – Specifying Pivot Table Groups*

The Pivot Table function is made even more powerful by its capability to aggregate data using *Pivot Table Groups*. For example, in Figure 11 the user can group stages into projects. Similarly, the user can aggregate product information in the manner that we described in Section 3.4.

STAGE	Project A	Project B	Project C
Holidays	106.7	201.1	202.4
Non-Project	107.3	124.1	596.3
Improvements	4.7	11.8	24.5
Misc Methods	1.5	5.3	12.9
OneSmallStep	7.6	10	17.9
Systems and Test Improvements	7.6	22.7	23.2
Systems Methods	20	28.4	17
Project A	38	15.8	18.3
Project B	28	17.2	8.6
Project C	12.2	4.7	11.2
C2.0	284.5	383.5	292.3
C2.1	192	81.4	157
C3.0	521.5	536.9	433.4
C4.0	679.4	463.7	749.3
C4.1	586.2	452.8	475
C4.3.3	420.5	481	516.8

Figure 11: CoBS Analysis Step 3 – Using Pivot Table Groups



(iv) *Step 4 – Generating Charts*

The major benefit of analysing PEL data using Pivot Tables is that the user can quickly and interactively query the data at all levels of granularity, right from large-grain data on projects and subsystems, down to individual tasks and work-products. The user can then easily generate charts like, for example, the one shown in Figure 12 (which shows the total effort on four projects over time) by selecting the Pivot Table and ‘Insert Chart.’

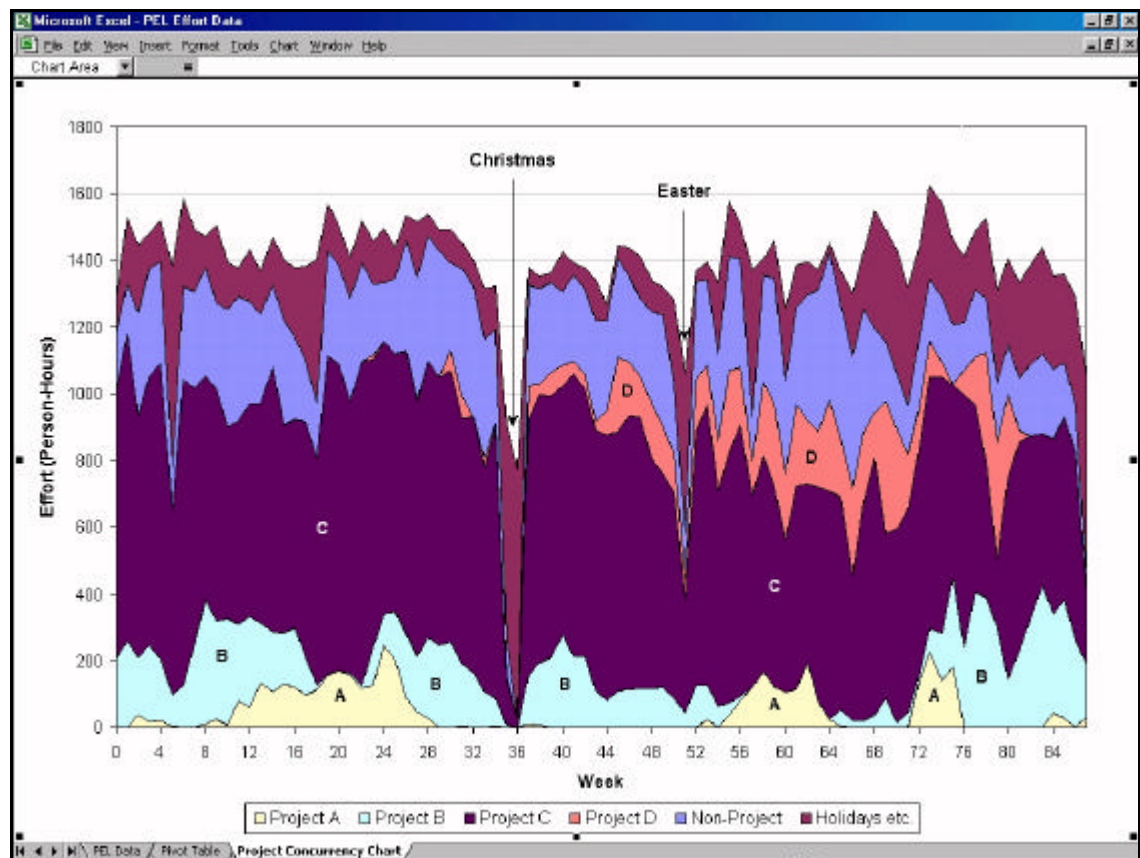


Figure 12: CoBS Analysis Step 4 – Generating Charts

(v) *Step 5 – Analysis Using Multiple Queries*

To make the analysis of data even easier, the author has used a series of Pivot Tables to create a multiple query interface shown in Figure 13. Each of the four queries shown at the bottom of the worksheet is used to plot a related line on the chart at the top of the worksheet. The user then selects from the pull-down menus to display the answer to a given query. For example, the queries in Figure 13 show the relative effort spent by the software team on delivery *D3.0 Top Level Design* (Query 2), *Code* (Query 3) and *Low Level Test* (Query 4) against the total spend on delivery *D3.0* (Query 1). The results shows how the *D3.0 Top Level Design* effort peaked and then tailed-off, while the *D3.0 Code* phase went on to absorb the majority of the code team's effort over the period. This capability to interactively query the data enabled users to quickly get a feel for the scale, duration and relative performance of concurrent activities at all levels in the process.

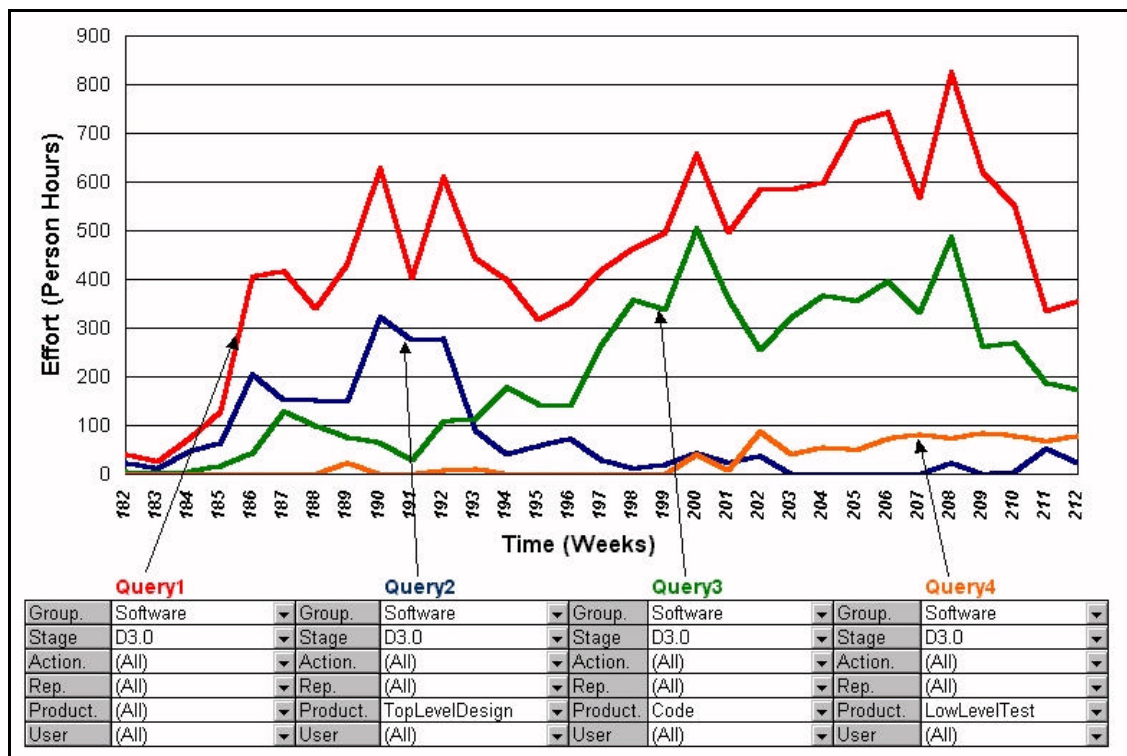


Figure 13: CoBS Analysis Step 5 – Analysis Using Multiple Queries



### 3.5.3 Benefits of PEL Effort Measurement

The development of the CoBS tool was an essential step in the introduction of PEL. The total cost of implementing CoBS was around £50,000 of which it is estimated that £10,000 was saved in administration costs in the first year alone (Brown 1999). The major benefit, however, from the manager's point of view, was the ability to measure costs more accurately and understand the impact of improvements.

In respect to the first part of our hypothesis (Section 1.4), we can now illustrate the benefits of fine-grained measurement in time-constrained software development. One of the most notable benefits for managers has been the ability to measure the scale and costs of rework and rework prevention.

In our implementation of PEL, rework is not treated as an activity in its own right, e.g. '*Rework Code*.' Rather, rework is measured by using stages that reflect project milestones or 'maturity gates.' Any work performed after a maturity gate can be considered rework. For example, once *D1.0 Code* has been delivered and signed-off, users can no longer allocate effort to the *D1.0 Code* task, but they can allocate it to the rework delivery, e.g. *D1.1 Code*. The PEL data can be used to show the effort spent coding, and low level testing, a particular function over time (Figure 14).

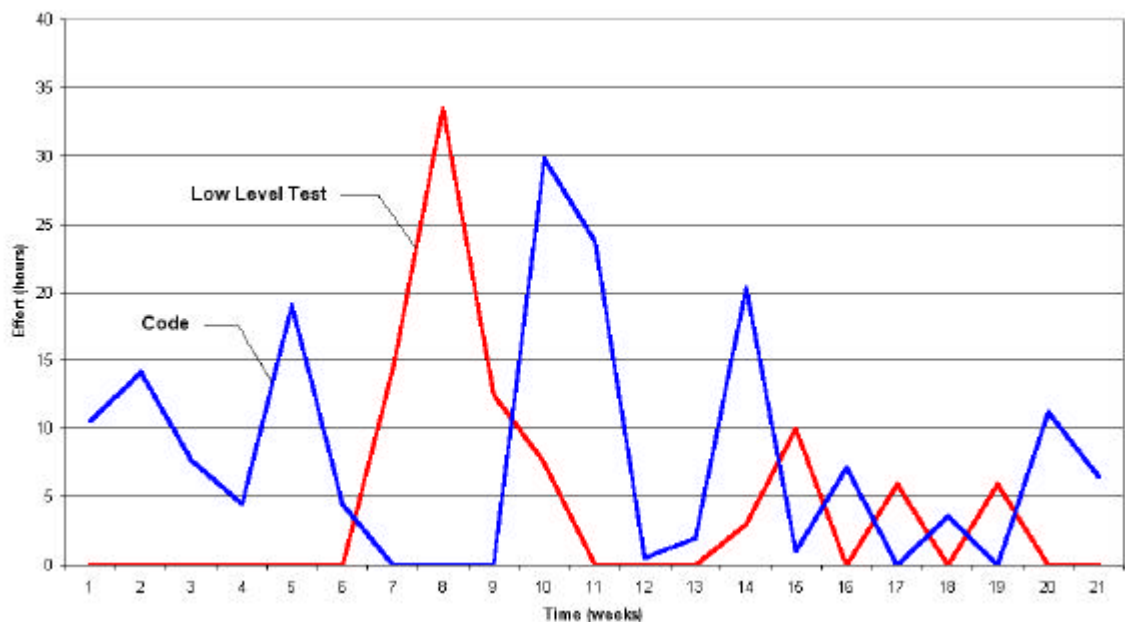


Figure 14: CoBS – Costs and Time-Delays of Rework

The chart in Figure 14 shows peaks and troughs in the two profiles where the Thrust Reverser function was repeatedly worked and reworked between coding and testing phases, over a number of successive software deliveries. This reveals the high cost of reworking this particular function and, equally critical to time-constrained development, the time-delays of the rework cycle. The impact of this rework was to divert resources away from their planned work, causing further pressure, delays and slippages. This analysis led to a number of process changes including re-engineering of the Thrust Reverser function and increasing the rigour of Code Reviews (Nolan 1999).

The introduction of PEL thus enabled managers to collect measures that would have been difficult to collect using conventional WBS-based approaches. Critically, by using PEL, we no longer have to presuppose the types of questions that we might later want to ask of our effort metrics (Figure 15).

Example Query	Conventional WBS	PEL
Effort per software delivery	Yes	Yes
Effort for the coding phase	Yes	Yes
Effort producing versus reviewing	No	Yes
Effort producing a particular component	No	Yes
Effort reworking a particular component	No	Yes
Effort spent on rework per delivery	No	Yes

Figure 15: Comparison of Queries Supported by Conventional WBS and PEL

The BR700 Controls Software manager explained why it was therefore necessary to collect fine-grained data. *‘The complexity of software development is such that a single partitioning of projects at the lowest level can make it hard to understand anomalies of spend, and near impossible to measure the effects of process change (software is a black art). Using PEL, software project expenditure is a function of Action, Stage, Product and Representation. There are cases when an improvement actually increases the cost of some functions but actually decreases the cost of coding. If you measure coding it might be deemed a success, if you measure the function it might be deemed a failure. If you use PEL you may actually be able to understand it!’* (Brown 1999).

## 3.6 Measuring Defects using PEL

### 3.6.1 Collection of PEL Defect Data

The second strand in the measurement programme was the collection of defect metrics (Powell 1995). The general approach to defect measurement was defined by McDermid: *“to aid the project manager by identifying ‘trouble-spots’ in the development process rather than providing predictions of ultimate product quality”* (McDermid 1997). This involved the collection of metrics on the types, causes and impacts of defects along with descriptions of where they were introduced, detected, and removed. In the absence of direct measures of ‘quality,’ these defect measures were used as a leading indicator of problems in both product and process quality.

The defect metrics had originally been gathered using a combination of a configuration control system (called PCMS) and a paper-based problem report known as a *Systems Anomaly Note* (SAN). However, the lack of validation and enforcement on data entry resulted in data that was inconsistent, inaccurate and, in cases, missing. The introduction of a new change control tool, during the course of this research, provided an opportunity to resolve these measurement problems.

The *Rolls Change Control* (RoCC) tool was developed by the Rolls-Royce BR700 Controls Methods Team to control all forms of systems and software change (Mathews 1997). The RoCC tool controls the lifecycle of a SAN as shown in Figure 16.

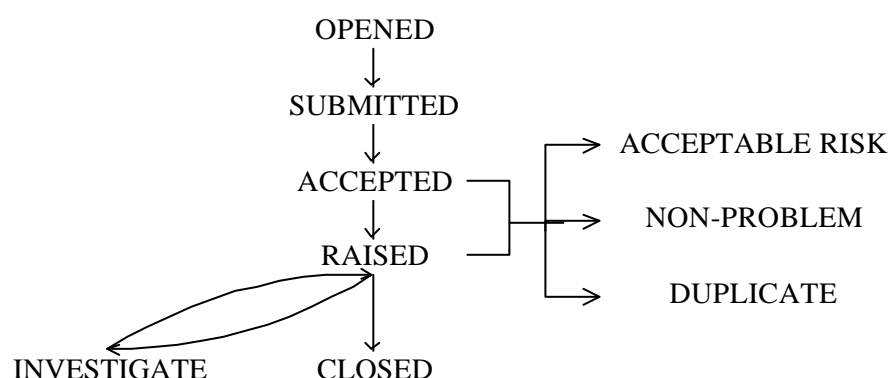


Figure 16: RoCC – Lifecycle of a SAN

The SAN lifecycle is managed by a cross-project committee of senior engineers called the ‘*Change Control Board*’ (CCB) who meet to review the *submitted* SANs. They may

decide to reject SANs that are (i) *duplicates* of other SANs, (ii) *acceptable risks*, or (iii) *non-problems*. The *accepted* SANs are then formally *raised*, given to the appropriate team to *investigate* and fix the SAN before it is finally *closed*.

The existence of a single system to support the entire change lifecycle provided a basis for the collection of defect data. The contribution of the author was to use PEL to ensure that defect information was collected in a consistent way to that of effort (Figure 3, Page 53). The defect metrics were collected when SANs were *raised*, *investigated* and *closed*.

(i) *Defects Metrics Collected when a SAN is Raised*

The RoCC Problem Report form, shown in Figure 17, is completed when a SAN is first raised. The *Information* frame identifies the title, unique ID, originator, date and version for each anomaly. The status field tracks the SAN through the lifecycle of being *Opened*, *Raised*, *Investigated* and *Closed*; or rejected as *Duplicate*, *Acceptable Risk* or *Non-Problem*. The tabs at the bottom of Figure 17 are used to collect a textual description of the problem, details of affected components, references to related documents, an early interpretation of the problem cause, and any graphics (e.g. state charts) to describe the problem or support its resolution.

**Problem Report**

**Information**

Problem Title: ARINC Output Data Can Overflow into sign Bit      Number: 543

Owner: MATHEWS      Version: 1

Originator: HEWITT      Date: 26/05/95      Status: OPENED

**Details**

Anomaly Category: Software Error

Problem Environment: Design Review

Functional Effect: Minor Functional Effect

Liability: SI

Delivery Identifier: G3.0

Originating project: BR710-GV

Unit Serial Number:

OK    Cancel    Help

Description   Details   Components   References   Interpretation   Graphics

Edit Mode

Figure 17: RoCC – Defect Metrics Collected when a SAN is Raised

The *Details* frame collects defect metrics on the process that discovered the problem, the problem type, and functional impact, using the following fields:

- *Anomaly Category* is the type of problem that has been observed, e.g. *Requirements, Software, Document/Traceability, or Improvement*;
- *Environment* is the process that located this particular problem (and maps to PEL Representations) e.g. *Code, Low Level Test* etc;
- *Functional Effect* is the functional impact of the problem on the hardware and software, i.e. *No Functional Effect, Minor Functional Effect, Cockpit Instrumentation Effect*, etc;
- *Liability* is the organisation responsible for providing a solution to the problem;
- *Delivery* is the software delivery in which the problem was discovered (and maps to PEL Stages), e.g. *C3.0, C3.1*;
- *Project* is derived automatically from the delivery identifier; and
- *Unit Serial Number* is the hardware on which the problem was identified.

The defect data is mandatory and verified during a review by the Change Control Board (CCB). The SANs are then grouped by the CCB, according to technical and schedule priorities, as an *Implementation Policy (IP)* of changes allocated to future deliveries.

It is difficult to determine precisely where defects have been introduced in evolutionary lifecycles. It was therefore necessary to give the users some guidance in determining the phase and stage where a problem was introduced. The phase was identified as the highest-level representation (e.g. Requirements) that required a change in order to fix the problem. The use of Static Analysis, Low Level Testing, Integration Testing, Requirements Testing and Rig testing all meant that errors were unlikely to slip from delivery to delivery undetected. The *Stage* where the problem was introduced was therefore assumed to be the most recent delivery. In all cases, the diagnosis of the problem type by the engineer, and its review by the CCB, was sufficient to provide accurate data.

(ii) *Defects Metrics Collected when a SAN is Investigated*

In the process of formulating a solution, the user must identify the components that will be affected when providing the fix (Figure 18). These map to PEL Products and Representations, e.g. *System Concept Document, Top Level Design, Code*.

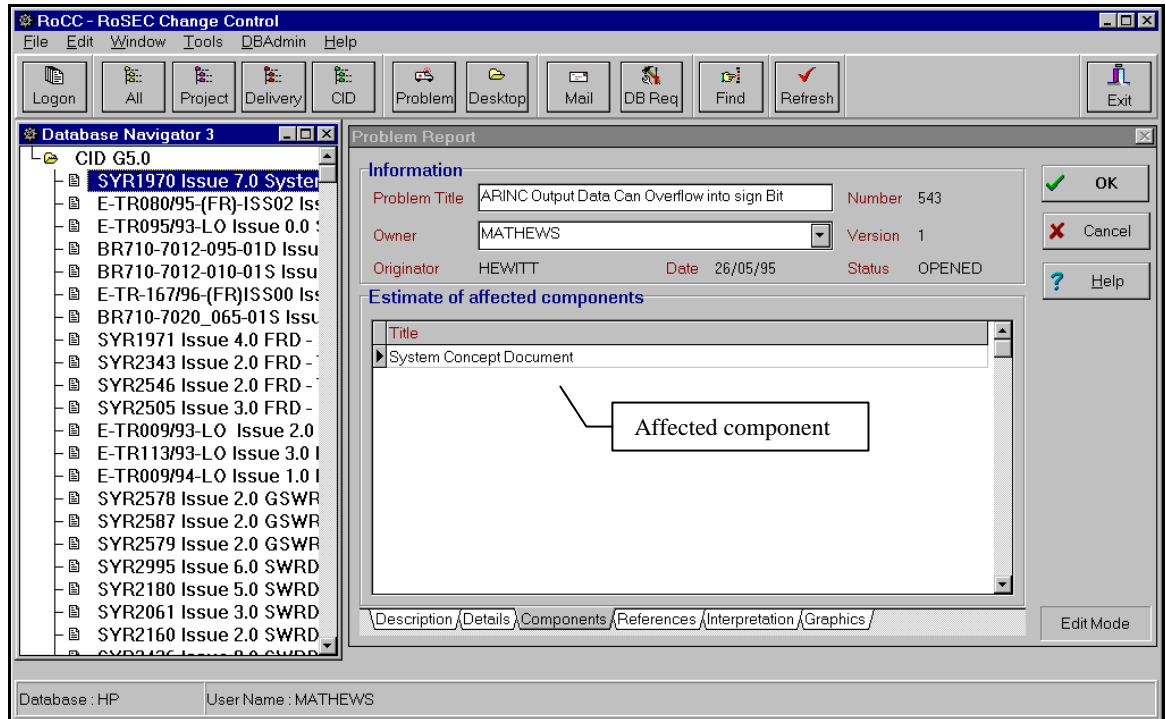


Figure 18: RoCC – Defect Metrics Collected when a SAN is Investigated

(iii) *Defects Metrics Collected when a SAN is Closed*

The RoCC Problem Closure form, shown in Figure 19, is completed when the SAN is closed. The *Evidence* frame records the following defect data.

- *Environment* is the process where the problem was fixed (and maps to PEL Representations) e.g. *Code, Low Level Test* etc.
- *Category* records the results of the closure testing, e.g. *Open, Closed* etc.
- *Delivery* is the software delivery in which the change was performed (and maps to PEL Stages) e.g. *C3.2, C3.2.1*.
- *Project* is derived automatically from the delivery identifier, e.g. *Project A*.
- *Unit Serial Number* identifies the hardware under test when the problem was closed.
- *Supporting Reference* is used to identify any associated documentation.

Figure 19: RoCC – Defect Metrics Collected when a SAN is Closed

(iv) *Summary of Defect Metrics*

The RoCC tool was therefore used to collect seven basic measures of defects:

<i>Defect Type</i>	e.g. <i>Ambiguity, Initialisation Error</i> etc;
<i>Defect Severity</i>	e.g. <i>No Functional Effect, Performance Degradation</i> etc;
<i>Affected Products</i>	e.g. <i>ARINC, Starting</i> ;
<i>Where Introduced</i>	e.g. <i>Produce Code Delivery D3.0</i> ;
<i>Where Detected</i>	e.g. <i>Perform Low Level Test Delivery D3.1</i> ;
<i>Where Fixed</i>	e.g. <i>Produce Code Delivery D3.2</i> ;
<i>Problem Status</i>	e.g. <i>Open, Closed, Duplicate</i> etc.

The measures themselves are standard to most defect measurement programmes. The major difference here is that *Affected Products*, *Where Introduced*, *Where Detected*, and *Where Fixed*, are collected using a PEL-compliant description of process (i.e. using *Action*, *Stage*, *Product* and *Representation*). Our measures of defects and effort are therefore collected in a consistent way using PEL. This makes it easy to integrate effort and defect data by querying CoBS and RoCC. For example, RoCC gives details of the defects fixed in ‘*Delivery D3.2*’ and CoBS then provides the corresponding effort.

In practice, there are still problems with defect data collection in the RoCC tool that the author could not influence during this study. The PEL lexicon is not directly linked to that of RoCC so the category values must be synchronised manually. RoCC would also benefit from the use of pull-down menus to describe defects like those used in the Cost Booking System. Nevertheless, the tool does collect fine-grained defect data that is consistent and compliant with PEL.

### 3.6.2 Analysis of PEL Defect Metrics

The RoCC tool contains features to help present and analyse defect metrics (Figure 20).

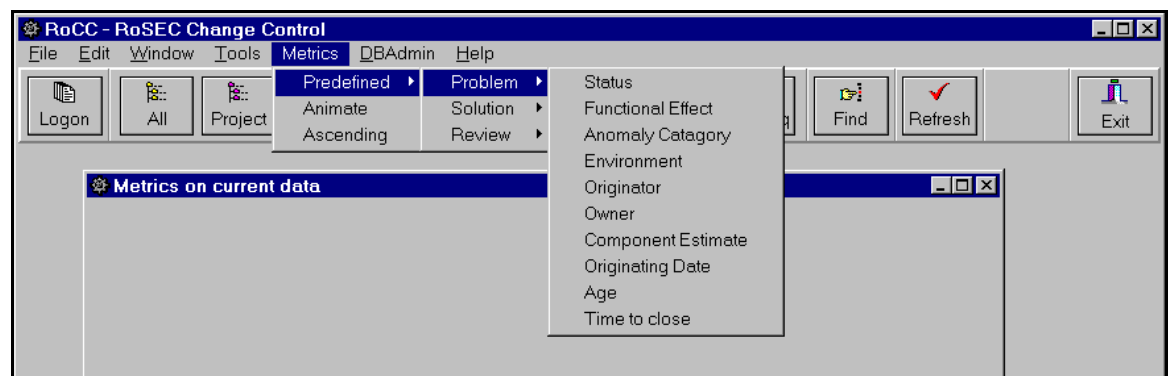


Figure 20: RoCC – Analysis of Defect Metrics

The chart in Figure 21 illustrates one type of analysis that can be performed. It shows, using illustrative data, a breakdown of defects in one software delivery by their *Severity*. Charts can also be produced to show the *actual* status of problems over time, *accumulated* running total of problems at each status, or *rate* per day that problems are reaching a status. The RoCC defect metrics can also be exported to a spreadsheet as shown in Figure 22 and analysed in the same way as effort (Section 3.5.2).



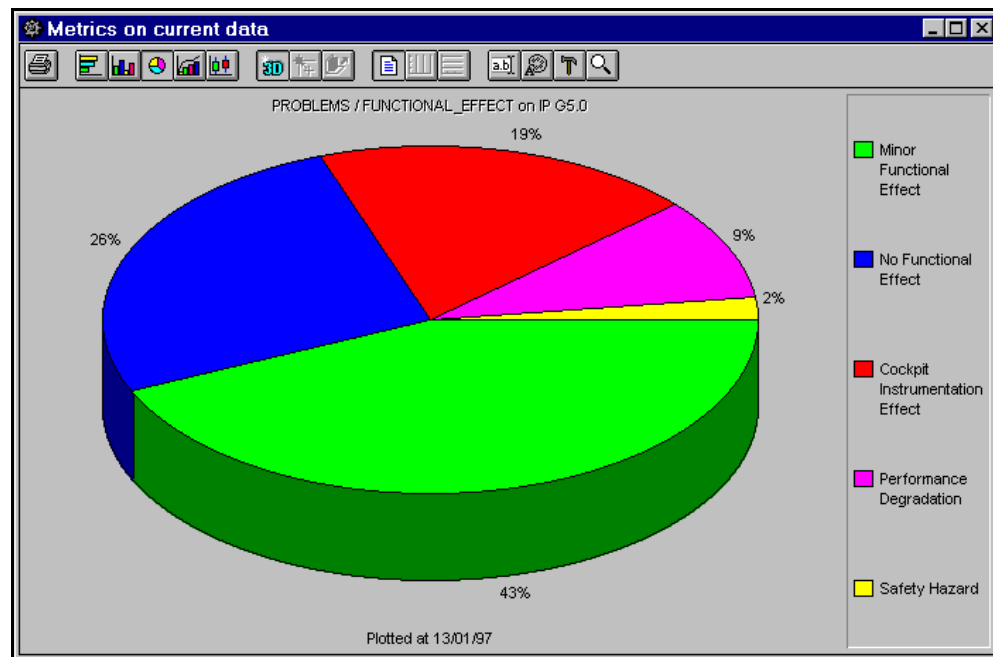


Figure 21: RoCC – Metrics Reporting Tool

Microsoft Excel - SAN Error Data													
File Edit View Insert Format Tools Data Financial Manager Window Help													
All Date													
1	A	B	C	D	E	F	G	H	I	J	K	L	M
	Date	Raised	Closed	Anomaly Category	Functional Effect	Environment							
102	14-May-98	C4.2	C4.2	Software	No Functional Effect	Design Review							
116	18-May-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Design Review							
117	3-Jun-98	C4.2	C4.4	Requirement	No Functional Effect	Design Review							
118	16-Jun-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
119	22-Jun-98	C4.2	C4.3.1	Software	Minor Functional Effect	Low-Level Test							
120	1-Jul-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
122	1-Jul-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
123	10-Jul-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
124	14-Jul-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
125	15-Jul-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
126	16-Jul-98	C4.2	C4.4	Software	No Functional Effect	Low-Level Test							
127	17-Jul-98	C4.2	C4.2	Document/Traceability	No Functional Effect	Requirements Review							
128	22-Jul-98	C4.2	C4.4	Software	No Functional Effect	Low-Level Test							
129	24-Jul-98	C4.2	C4.4	Software	Cockpit Instrumentation Effect	Low-Level Test							
137	7-Aug-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Low-Level Test							
1002	10-Aug-98	C4.2	C4.4	Requirement	Minor Functional Effect	System Acceptance Test							
1008	10-Aug-98	C4.2	C4.4	Requirement	Cockpit Instrumentation Effect	Design Review							
1449	10-Aug-98	C4.2	C4.3.1	Software	No Functional Effect	Code							
1450	11-Aug-98	C4.2	C4.3.1	Software	Minor Functional Effect	Low-Level Test							
1491	11-Aug-98	C4.2	C4.3.1	Software	No Functional Effect	Code							
1492	11-Aug-98	C4.2	C4.3.1	Software	No Functional Effect	Code							
1471	12-Aug-98	C4.2	C4.3.1	Software	Minor Functional Effect	Low-Level Test							
1472	13-Aug-98	C4.2	C4.4	Requirement	No Functional Effect	System Acceptance Test							
1473	18-Aug-98	C4.2	C4.3.1	Software	No Functional Effect	Low-Level Test							
1474	18-Aug-98	C4.2	C4.3.1	Software	Minor Functional Effect	Design Review							
1475	19-Aug-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Design Review							
1476	1-Oct-98	C4.2	C4.4	Requirement	Minor Functional Effect	Requirements Review							
1477	5-Oct-98	C4.2	C4.4	Requirement	Minor Functional Effect	Engine Test							
1478	15-Oct-98	C4.2	C4.4	Requirement	Minor Functional Effect	System Acceptance Test							
1500	15-Oct-98	C4.2	C4.4	Requirement	Performance Degradation	Flight Test							
1502	21-Oct-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Design Review							
1503	22-Oct-98	C4.2	C4.4	Requirement	Minor Functional Effect	Requirements Review							
2265	22-Oct-98	C4.2	C4.4	Requirement	Minor Functional Effect	System Acceptance Test							
2294	26-Oct-98	C4.2	C4.4	Requirement	Performance Degradation	System Acceptance Test							
2295	30-Oct-98	C4.2	C4.4	Requirement	Minor Functional Effect	System Acceptance Test							
2296	11-Nov-98	C4.2	C4.4	Requirement	Cockpit Instrumentation Effect	System Acceptance Test							
2297	12-Nov-98	C4.2	C4.4	Requirement	Minor Functional Effect	Requirements Review							
2307	16-Nov-98	C4.2	C4.4	Requirement	Minor Functional Effect	System Acceptance Test							
2308	23-Nov-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Requirements Review							
2309	23-Nov-98	C4.2	C4.4	Requirement	Cockpit Instrumentation Effect	System Acceptance Test							
2310	24-Nov-98	C4.2	C4.4	Document/Traceability	No Functional Effect	Requirements Review							
2314	25-Nov-98	C4.2	C4.4	Requirement	Cockpit Instrumentation Effect	Flight Test							
2333	28-Jan-99	C4.2	C4.4	Document/Traceability	No Functional Effect	System Acceptance Test							
2334	2-Mar-99	C4.2	C5.0	Document/Traceability	No Functional Effect	Design Review							

Figure 22: RoCC – Raw Data in Microsoft Excel

An example of defect analysis particularly relevant to time-constrained software development is shown in Figure 23. This shows the relative percentage of SANs caught by each detection phase for four consecutive projects: *A*, *B*, *C* and *D*.

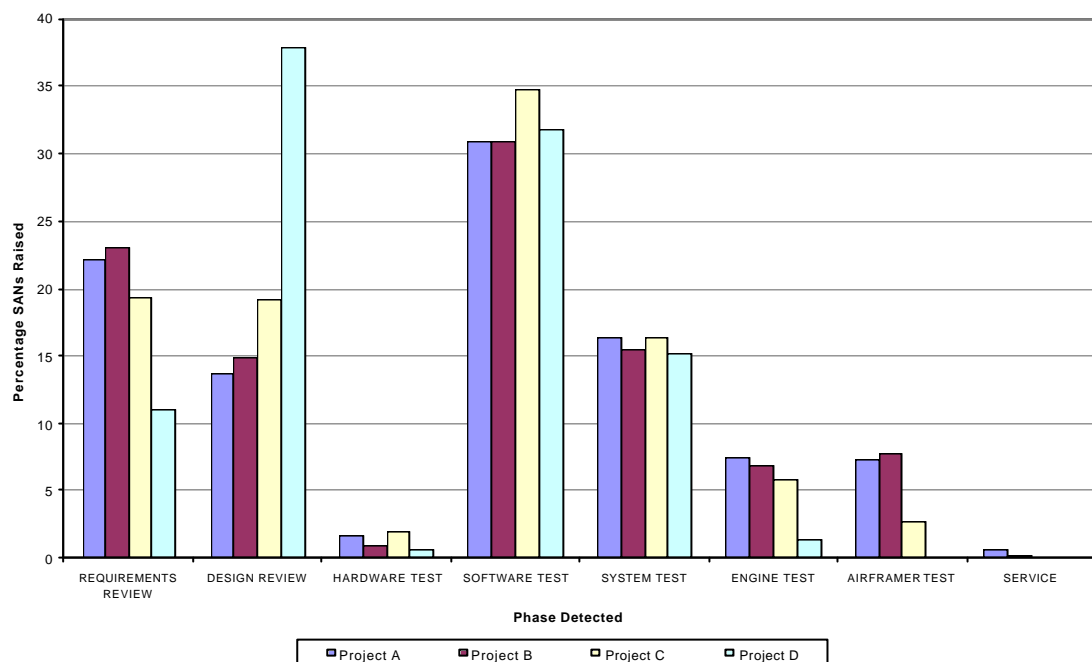


Figure 23: RoCC – Defect Detection Profile

If we compare the profiles, the projects show a shifting pattern in the relative number of defects caught by each detection phase. For example, in *Project A* a higher percentage of defects were found in Requirements Review than *Project B*, but the reverse is true for the number of defects found during Design Reviews. There are a number of explanations for this shift. For example, the increased software reuse could be leading to more stable requirements and a greater emphasis on designing reusable components. This analysis was used by managers evaluate the reuse process.

### 3.6.3 Benefits of PEL Defect Measurement

The author identified six key questions that were used to monitor process quality (Powell 1995). These questions are listed below along with observations from analysis performed by Mike Bardill of the BR700 Engine Controls Methods Team.

1. Which system aspects generate the most problems? (i.e. are there any particularly troublesome areas to target?). Bardill observed that “*a small number of functions are*

*responsible for a large number of anomalies - 41% of SANs are attributable to just 6 functional areas [out of 40].”*

2. Which types of anomaly occur most frequently? (i.e. do we keep making the same sort of mistakes and, if so, which parts of the process need attention?). Bardill observed that *“as well as causing a large number of anomalies, these 6 functional areas also create significant numbers of ‘late process’ changes, take many iterations to reach stability and cause many of the highest risk SANs.”* As a result of questions 1 and 2, a decision was taken to incrementally improve ‘key’ functional areas rather than attempt wholesale process improvement.
3. Which test environments detect the most anomalies? (i.e. are we spending money testing and not finding anything?). Bardill observed that *“the majority (70%) of errors are due to software or requirement errors”* and *“too few errors are being detected by the requirements review process.”* This observation that requirements errors were not being detected until late in the process led to the requirements and interface testing activities to be performed earlier in the process. Furthermore, he observed, *“only 12% of changes were due to new requirements.”* This showed that changes in high-level (customer) requirements were significant, but not as numerous as sometimes perceived.
4. Which system aspects tend to generate particular types of anomaly? (i.e. are there any aspects that tend to generate problems that are expensive to fix?). Bardill observed, *“there are specific problem areas such as the interface between the system and the airframe.”* He therefore recommended better modelling and testing techniques for airframe and engine interfaces.
5. Which system aspects tend to generate anomalies late in the development process? (i.e. the most costly time to rectify problems). Bardill observed, *“the ‘Pressures’ function causes a significant number of ‘late process’ problems.”* Before this analysis, the pressures function was not typically known as a problem area.
6. Do the number of anomalies for a particular system aspect diminish over time? (i.e. do we learn from our experience, or do we keep getting it wrong?). Bardill observed, *“key functions appear to take several attempts to reach maturity.”* This suggested that iteration was a necessary part of the development of control systems.

This analysis is typical of the types of questions that are now being answered as a result of the defect measurement programme.

### 3.7 Measuring Size using PEL

#### 3.7.1 Collection of PEL Size Data

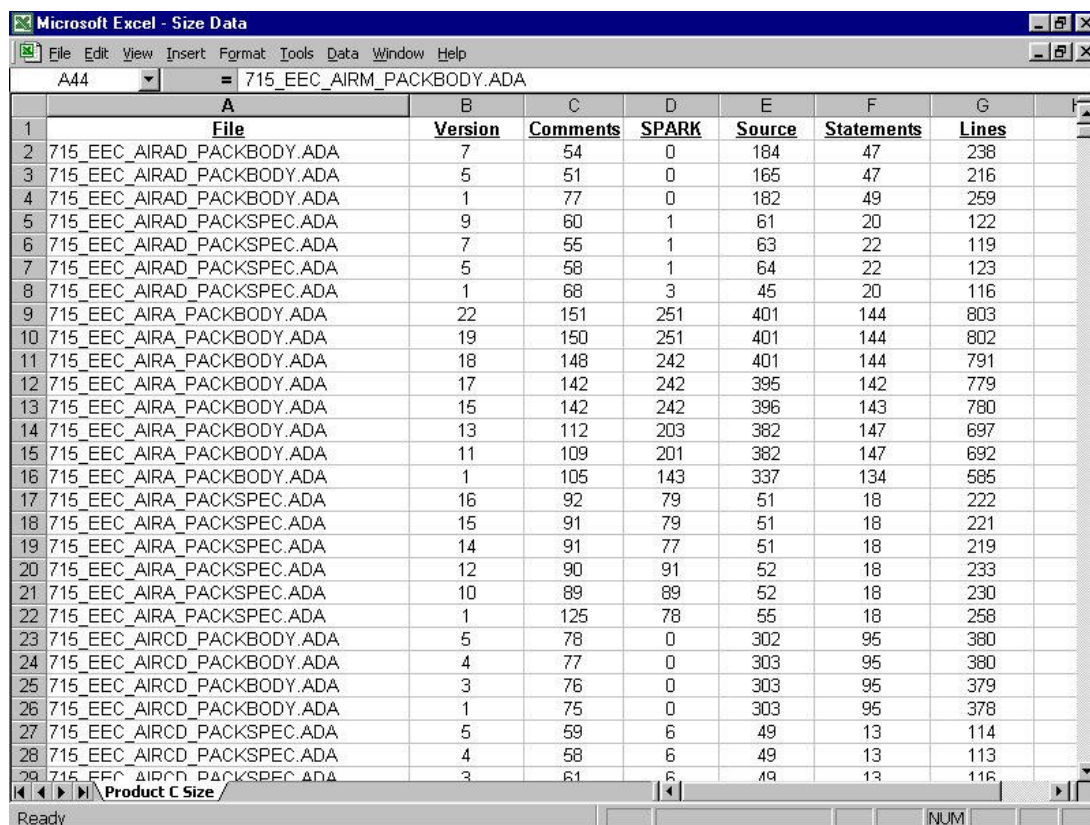
The third and final strand of our measurement programme was the collection of data on the size of the work-products created. The size measures used were simple counts of SAN change requests and Lines of Code (LoC).

SAN change requests were used to measure a unit of ‘change.’ The historical change measure used in Rolls-Royce was the *Code Item (CI)*. The CI originally equated to 100 LoC or ‘half an Ada module’ and was used as a rule of thumb for sizing code changes. Over time, the CI became widely used as a surrogate measure of effort, e.g. 1 CI takes 5 person-days. The CI had, however, lost its empirical relationship to both size and effort principally due to a lack of re-validation. The author therefore persuaded the team to replace the use of Code Items with an empirical understanding of the size and effort of SAN change requests to indicate the levels of change in a project. These SAN counts were collected using the RoCC tool (described in Section 3.6.1).

LoC was used in the absence of a better candidate and because of its general acceptance in Rolls-Royce. Despite the numerous measures that have been suggested, LoC remains the most widely used indicator of software size (Basili and Weiss 1984; Fenton 1991; Shepperd 1992; Azuma and Mole 1994). Indeed, in an internal study by Rolls-Royce, found LoC to be a better indicator of effort than Function Points (Nolan 1999).

The author’s role as manager of the Practical Reuse Improvement MEtrics (PRIME) project gave an opportunity to commission a tool to measure the size of code changes. The Comparator tool was developed in 1997 by Steve Kerr (Powell 1997) and later improved by Andrew Nolan (Nolan 1999) of the Rolls-Royce Control Systems Methods Team. It measures the change to the Ada code at the line, file and function level. A *line* represents a physical line of code, excluding comments, annotations and blank lines. A *file* is typically an Ada function, package body, package specification or module. A *function*, or functional area, is a set of files that form a high-level abstraction of the system functionality and requirements.

The Comparator calculates the number of differences between baseline releases in the configuration management system. A *release* is a defined set of code modules (consisting of many functions) released for operation. The *baseline* refers to the release of software from which changes have been subsequently applied. By comparing the items changed, added and deleted, it is possible to calculate the levels of change (and reuse) at the line, file and function level. The raw data produced by the Comparator tool is shown in Figure 24.



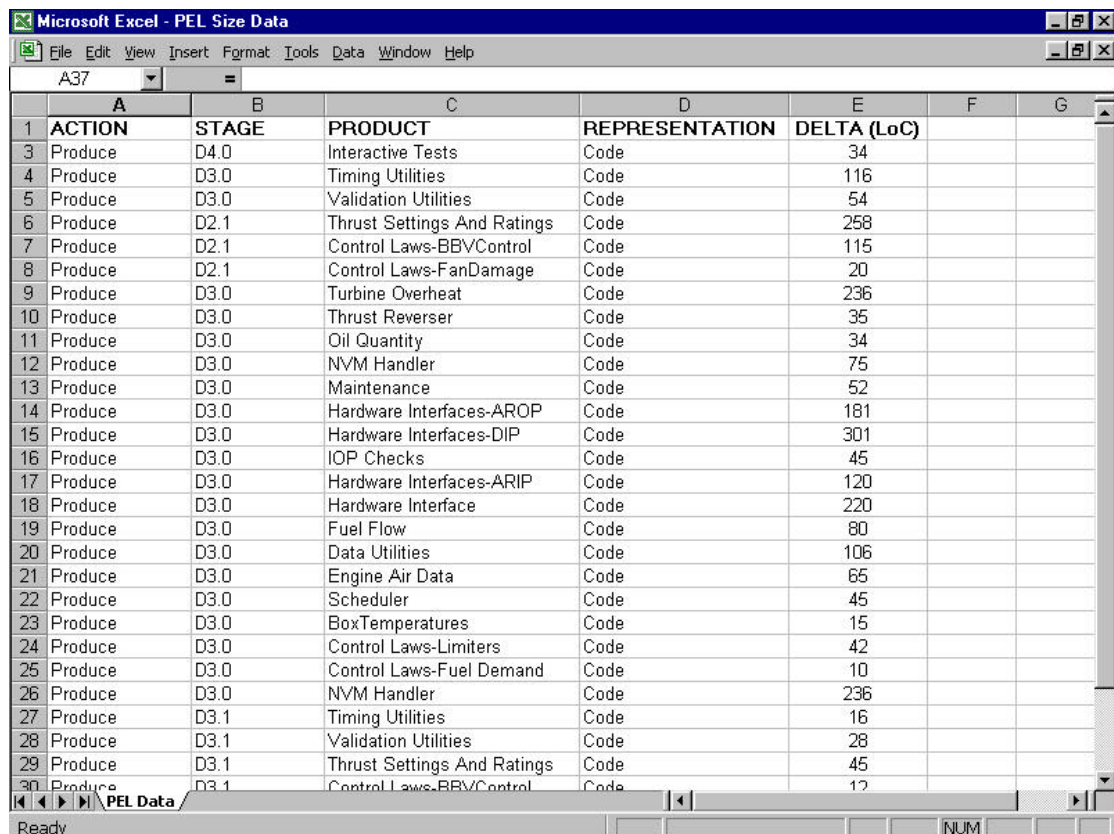
The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Size Data". The active cell is A44, and the formula bar shows "= 715\_EEC\_AIRM\_PACKBODY.ADA". The table has 8 columns: File, Version, Comments, SPARK, Source, Statements, and Lines. The data is organized into rows, with the first row being a header. The table lists various files and their corresponding version, comments, SPARK, Source, Statements, and Lines.

	A	B	C	D	E	F	G
	File	Version	Comments	SPARK	Source	Statements	Lines
2	715_EEC_AIRAD_PACKBODY.ADA	7	54	0	184	47	238
3	715_EEC_AIRAD_PACKBODY.ADA	5	51	0	165	47	216
4	715_EEC_AIRAD_PACKBODY.ADA	1	77	0	182	49	259
5	715_EEC_AIRAD_PACKSPEC.ADA	9	60	1	61	20	122
6	715_EEC_AIRAD_PACKSPEC.ADA	7	55	1	63	22	119
7	715_EEC_AIRAD_PACKSPEC.ADA	5	58	1	64	22	123
8	715_EEC_AIRAD_PACKSPEC.ADA	1	68	3	45	20	116
9	715_EEC_AIRA_PACKBODY.ADA	22	151	251	401	144	803
10	715_EEC_AIRA_PACKBODY.ADA	19	150	251	401	144	802
11	715_EEC_AIRA_PACKBODY.ADA	18	148	242	401	144	791
12	715_EEC_AIRA_PACKBODY.ADA	17	142	242	395	142	779
13	715_EEC_AIRA_PACKBODY.ADA	15	142	242	396	143	780
14	715_EEC_AIRA_PACKBODY.ADA	13	112	203	382	147	697
15	715_EEC_AIRA_PACKBODY.ADA	11	109	201	382	147	692
16	715_EEC_AIRA_PACKBODY.ADA	1	105	143	337	134	585
17	715_EEC_AIRA_PACKSPEC.ADA	16	92	79	51	18	222
18	715_EEC_AIRA_PACKSPEC.ADA	15	91	79	51	18	221
19	715_EEC_AIRA_PACKSPEC.ADA	14	91	77	51	18	219
20	715_EEC_AIRA_PACKSPEC.ADA	12	90	91	52	18	233
21	715_EEC_AIRA_PACKSPEC.ADA	10	89	89	52	18	230
22	715_EEC_AIRA_PACKSPEC.ADA	1	125	78	55	18	258
23	715_EEC_AIRCD_PACKBODY.ADA	5	78	0	302	95	380
24	715_EEC_AIRCD_PACKBODY.ADA	4	77	0	303	95	380
25	715_EEC_AIRCD_PACKBODY.ADA	3	76	0	303	95	379
26	715_EEC_AIRCD_PACKBODY.ADA	1	75	0	303	95	378
27	715_EEC_AIRCD_PACKSPEC.ADA	5	59	6	49	13	114
28	715_EEC_AIRCD_PACKSPEC.ADA	4	58	6	49	13	113
29	715_EEC_AIRCD_PACKSPEC.ADA	3	61	6	49	13	116

Figure 24: Comparator – Raw Size Data

### 3.7.2 Analysis of PEL Size Data

The size metrics collected by the comparator tool can be imported into a spreadsheet for analysis (Figure 25) and mapped to PEL statements in the same way as for effort and defects. For example, the measure of size can be expressed ‘*Produce Delivery D3 Code*’ = 565 Lines of Code. It is therefore possible to query the results of the three data collection tools to determine the total effort, number and type of defects, and size (LoC, SANs or reuse levels) for any combination of PEL dimensions.



	A	B	C	D	E	F	G
	ACTION	STAGE	PRODUCT	REPRESENTATION	DELTA (LoC)		
1	Produce	D4.0	Interactive Tests	Code	34		
4	Produce	D3.0	Timing Utilities	Code	116		
5	Produce	D3.0	Validation Utilities	Code	54		
6	Produce	D2.1	Thrust Settings And Ratings	Code	258		
7	Produce	D2.1	Control Laws-BBVControl	Code	115		
8	Produce	D2.1	Control Laws-FanDamage	Code	20		
9	Produce	D3.0	Turbine Overheat	Code	236		
10	Produce	D3.0	Thrust Reverser	Code	35		
11	Produce	D3.0	Oil Quantity	Code	34		
12	Produce	D3.0	NVM Handler	Code	75		
13	Produce	D3.0	Maintenance	Code	52		
14	Produce	D3.0	Hardware Interfaces-AROP	Code	181		
15	Produce	D3.0	Hardware Interfaces-DIP	Code	301		
16	Produce	D3.0	IOP Checks	Code	45		
17	Produce	D3.0	Hardware Interfaces-ARIP	Code	120		
18	Produce	D3.0	Hardware Interface	Code	220		
19	Produce	D3.0	Fuel Flow	Code	80		
20	Produce	D3.0	Data Utilities	Code	106		
21	Produce	D3.0	Engine Air Data	Code	65		
22	Produce	D3.0	Scheduler	Code	45		
23	Produce	D3.0	BoxTemperatures	Code	15		
24	Produce	D3.0	Control Laws-Limiters	Code	42		
25	Produce	D3.0	Control Laws-Fuel Demand	Code	10		
26	Produce	D3.0	NVM Handler	Code	236		
27	Produce	D3.1	Timing Utilities	Code	16		
28	Produce	D3.1	Validation Utilities	Code	28		
29	Produce	D3.1	Thrust Settings And Ratings	Code	45		
30	Produce	D3.1	Control Laws-BBVControl	Code	12		

Figure 25: Comparator – Analysis of Size Data

### 3.7.3 Benefits of PEL Size Measurement

The results have been used to evaluate the levels of change and reuse at all levels; from a change to one Ada module, to assessing the total level of reuse between two projects. For example, the bars in Figure 26 show the percentage reuse per module for (i) *Project B* (compared to the baseline of *Project A*), and (ii) *Project C* (compared to the baseline of projects *A* and *B*),

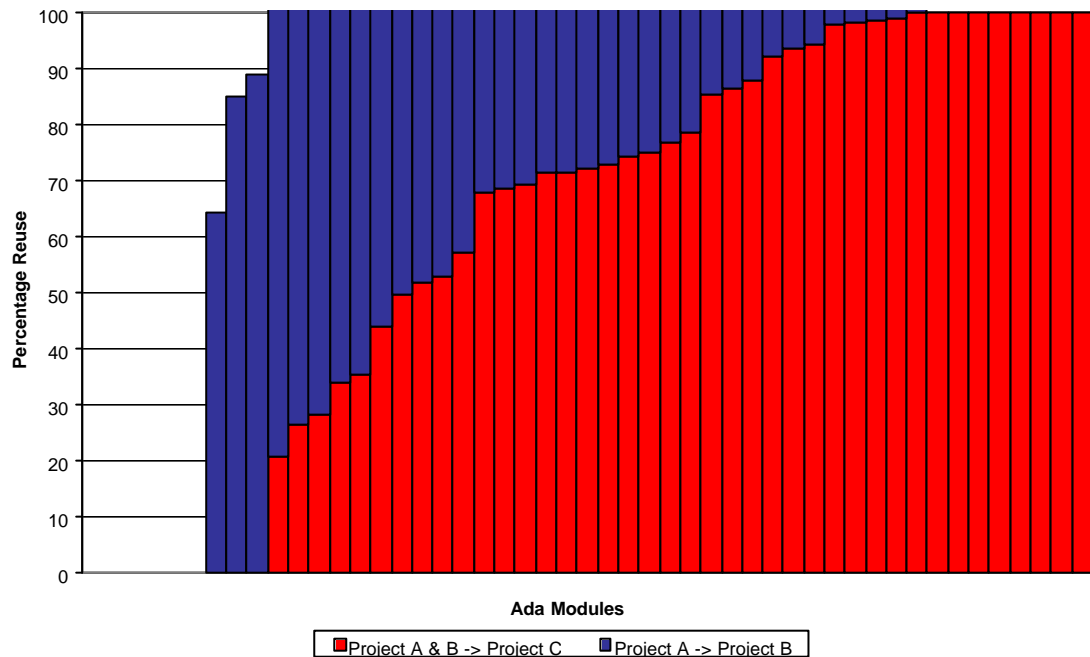


Figure 26: Comparator – Levels of Product Reuse

The results in Figure 26 show that the majority of modules in *Project B* were 100% reused (including their corresponding documentation and tests). In contrast, *Project C* has a much lower level of reuse involving the partial change of many modules (and thus a need to change their documentation and tests). This difference in reuse levels is partly explained by the fact that projects *A* and *B* were for variants of the same engine, whereas *Project C* was for a different engine application (i.e. the levels of reuse achieved in *Project C* were in fact quite good). Hence, this form of analysis shows if the reuse process is working and how, and where, it could be improved.

The comparator tool has provided clear evidence of the change between any two releases of the system. The key benefits come when the information is combined with effort and defect information as a basis.

### 3.8 Summary of Measuring Time-Constrained Development

In this chapter, we have identified problems and solutions in the measurement of time-constrained software development.

First, we have described specific problems of using Work Breakdown Structures for measuring effort. These are due to inconsistencies and ambiguities in the way activities are structured and described. In particular, due to their hierarchical structure, it is impractical to create a WBS that describes every combination of product and process that we need for fine-grained measurement. In other words, the way a WBS is described limits the information that it collects and the questions that can later be answered.

Second, we have used the Process Engineering Language (PEL) approach as a basis for understanding time-constrained development. PEL resolves the problems of WBS by using a consistent and concise language set to build a structured description of activity. Since PEL captures a matrix of both the products created and process performed, it is more flexible (compared to WBS) in the types of questions that can later be answered.

Third, we defined a practical set of Goals, Questions and Metrics for the collection of metrics on effort, defects, and size, in time-constrained software development.

Fourth, we have introduced PEL effort collection using the Cost Booking System (CoBS) tool. The CoBS tool was first introduced in 1996 and its use has steadily grown to be used by over 150 engineers. This enabled the analysis of effort metrics to support our observations of time-constrained development in Chapter 4.

Fifth, we have introduced PEL defect metrics into the Rolls Change Control tool (RoCC). The use of a PEL ensured that these defects metrics were consistent with effort metrics collected using CoBS. This enabled the analysis of the impact of defects and some of the insights that we describe in Chapter 5.

Sixth, we have introduced the collection of PEL size metrics using the Comparator tool. This measures the size of code changes and the level of change between two software releases. The use of PEL ensured that these size metrics were consistent with the effort and defect metrics. This enabled analysis of the scale of change and the insights we describe in Chapter 5.

Finally, we have successfully achieved technology transfer of these tools and approaches into industrial practice. Andrew Nolan of Rolls-Royce stated that: “*Metrics*



*collection should be a natural by-product of the process and should not be an addition to it. In this way, metrics are automatic. For example, in the BR700 FADEC Controls group, time bookings can only be made through the PEL tool and change requests can only be made through the RoCC tool. These tools automatically produce the metrics [we need]” (Nolan 1999). In addition, the Rolls-Royce Controls Business Development Manager said that PEL gave “[...] significant benefits in gathering metrics to identify and drive business improvements” (Cann 1999).*

The introduction of PEL was fundamental to understanding the nature and problems of time-constrained software development.

## Chapter 4:

# Observations on Time-Constrained Development

---

### 4.1 Introduction

The previous chapter introduced a new fine-grained approach for the description and measurement of software development. It was argued that the Process Engineering Language (PEL) has a number of advantages over present techniques, most notably its ability to capture attributes of a process that would otherwise be obscured by current approaches to data collection.

In this chapter, we investigate how time-constrained software developments differ from their conventional counterparts. To do this, we use our PEL effort measurements from Chapter 3 to observe the nature of time-constrained projects within Rolls-Royce. We start by describing the processes by which aeroengine control systems and software are developed (Section 4.2). We then study the interacting effects of concurrency and iteration at the levels of *Project* (Section 4.3), *Delivery* (Section 4.4), *Phase* (Section 4.5), and *Task* (Section 4.6).

### 4.2 Rolls-Royce Control Systems Development

#### 4.2.1 Background – Products and Processes

In the development of engines for civil aircraft, the time-to-market of new products can mean the difference between winning and losing multi-million pound supply contracts. Lead-time compression is therefore a critical driver of competitive performance.

An essential part of modern aeroengines is the Full Authority Digital Engine Controller (FADEC). The FADEC is a system responsible for the control of engine thrust, performance, monitoring and cockpit communication. The major component of the FADEC is an Electronic Engine Controller (EEC) – the microprocessor hardware and real-time embedded software controls an engine.

The EEC software consists of around 210,000 lines of code including 100,000 lines of Ada or assembler source code, 46,000 lines of design annotations, and 64,000 lines of comments. However, whilst the software is relatively small, the development process is expensive (costing millions of pounds) and time-consuming (taking around two years). It is therefore a prime target for lead-time improvement.

The Control Systems in this study were being developed by the Rolls-Royce BR700 Engine Controls Team for the BR700 series of aeroengines being produced by the BMW Rolls-Royce. The BR700 series includes a BR710 engine and a larger thrust BR715 engine. These engines serve the mid-range market of business, cargo and passenger aircraft (i.e. up to 130 passengers). The metrics were collected using the PEL approaches described in Chapter 3 and consider a 20-month snapshot of EEC Software development (as shown in Figure 27).

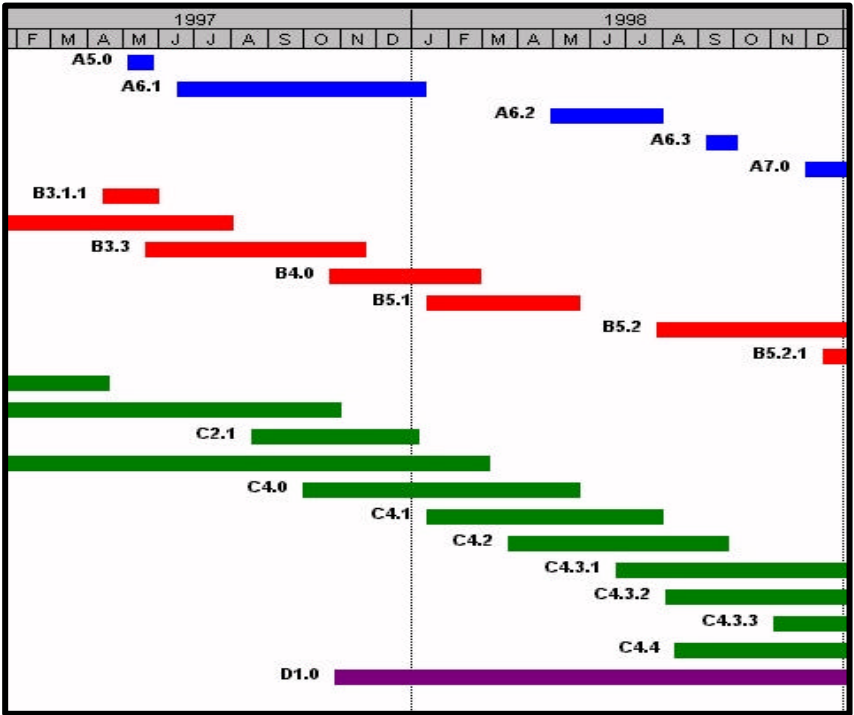


Figure 27: Rolls-Royce – Development Schedule

During this time, around 35 engineers were working on four projects (A, B, C and D), all in different states of maturity, to deliver control systems for different engine *variants*, airframes and airline customers. *Project A* was the launch application for the BR700 engine on the Gulfstream V aircraft and was undergoing final changes for entry into service. *Project B* was the second application of the same engine for a different

airline and airframe (Canadair Bombardier Global Express) and was undergoing final flight-testing and certification. *Project C* was the launch application for the larger thrust BR715 engine for the Boeing 717 aircraft and was in its main stages of development. *Project D* was in the initial stages of the third application of the BR700 engine for the Nimrod aircraft.

#### 4.2.2 Development Process

The development of engines and control systems forms a nested set of activities (Figure 28). At the highest level is the aeroengine project that provides the requirements and context for the concurrent development of the engine control system (FADEC). Within this is the EEC software development project that is itself broken down into a number of software deliveries. Each delivery of control system software provides new, or updated, functionality to support key milestones in the engine rig and flight-testing programmes. The result is a hierarchy of projects with interacting constraints and performance.

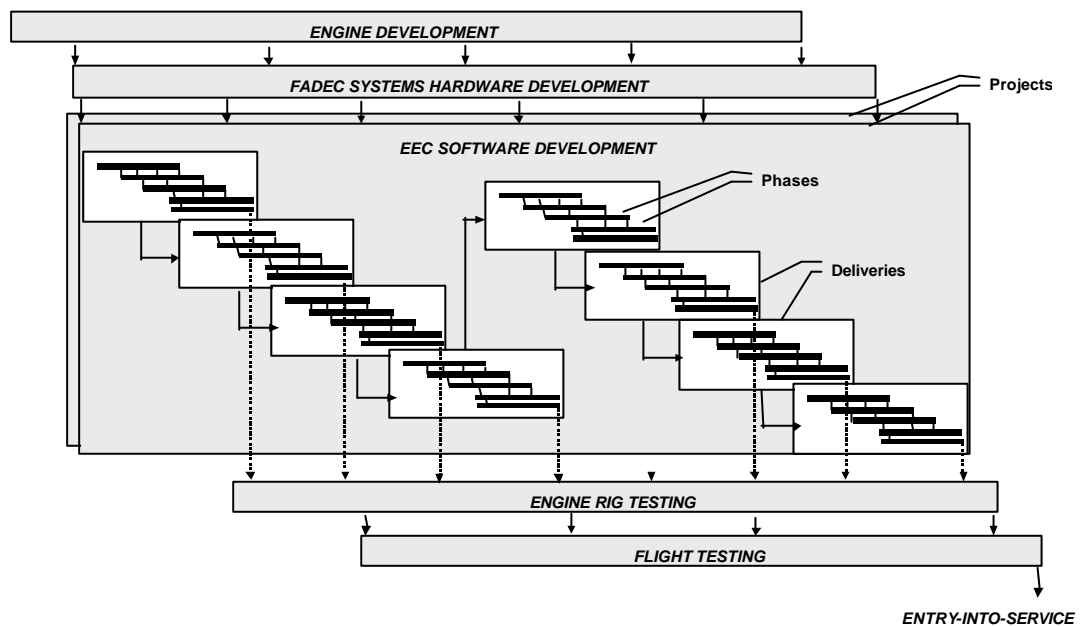


Figure 28: Rolls-Royce – Development Process

The control systems project is heavily time-constrained within the boundaries of the engine development and engine testing programmes. The control system cannot be finalised until the engine is designed. Likewise, the engine cannot be tested until a control system exists. The control system software is therefore on the critical path of the

whole engine project. Any slippage in the control system project can cause delays in the engine programme with significant commercial implications. Timescales are therefore fixed and subject to great pressure for further compression. It is thus an example of time-constrained software development.

Significant lead-time reduction in aeroengine development has been achieved by the simultaneous development of EEC hardware and software, in parallel with the development and testing of engines themselves. This reduction was due to the combined application of concurrency and iteration. Concurrency is the simultaneous performance of development activities between projects, product deliveries, development phases and individual tasks. Iteration is the repetition of development activities to deliver increments of product functionality at pre-planned intervals. It is the extent to which concurrency and iteration are used in EEC development that makes the Rolls-Royce process extremely relevant to this research.

#### ***4.2.3 Concurrency and Iteration***

It was observed that concurrency and iteration are used to reduce the timescales of aeroengine and control system projects in a number of ways (Figure 28). First, a number of projects to develop new engines or engine variants are performed at the same time. Second, the engine development, control system development, and engine testing are conducted in parallel. Third, hardware and software for the control system are developed together. Finally, within software projects the deliveries and phases (e.g. code and test) are extensively overlapped. For the purposes of this investigation, we focus on the concurrency within the software development process. This can be represented as a hierarchy of concurrency (Figure 29).

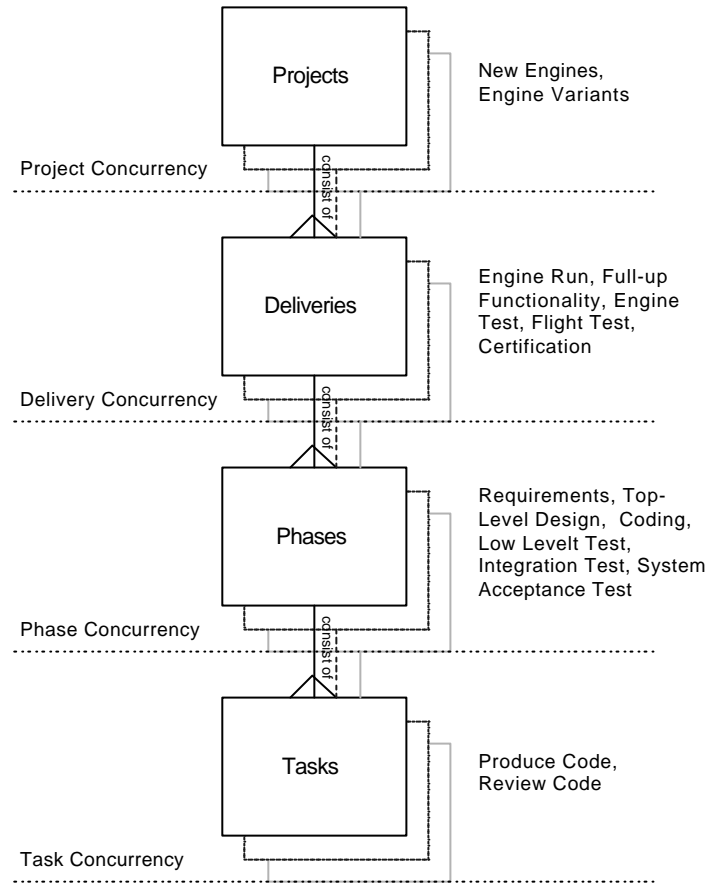


Figure 29: The Hierarchy of Concurrency

The identification of this structure has guided our analysis and laid the foundation for a number of insights that follow. At the highest level is *project concurrency* where the software team works concurrently on software for different engine projects (Section 4.3). Each project uses *delivery concurrency* to iterate through the software lifecycle and provide successive increments of software for engine testing (Section 4.4). Within each delivery, *phase concurrency* is used to overlap the development phases, e.g. code and testing (Section 4.5). Finally, each phase uses *task concurrency* by allocating tasks to be worked in parallel by different engineers (Section 4.6). In the following Sections, we consider each level of this hierarchy in turn, but first we give a background summary of the Rolls-Royce dataset.

#### 4.2.4 Interdependencies between Resources, Products and Processes

The four projects must be studied together because of interdependencies between resources, products and processes.

*Resource interdependencies* exist because the projects are organised around shared phase teams as opposed to project teams (Figure 30). This maintains the independence of development and testing activities for reasons of product assurance. However, it means that different projects must compete for resources from a shared pool.

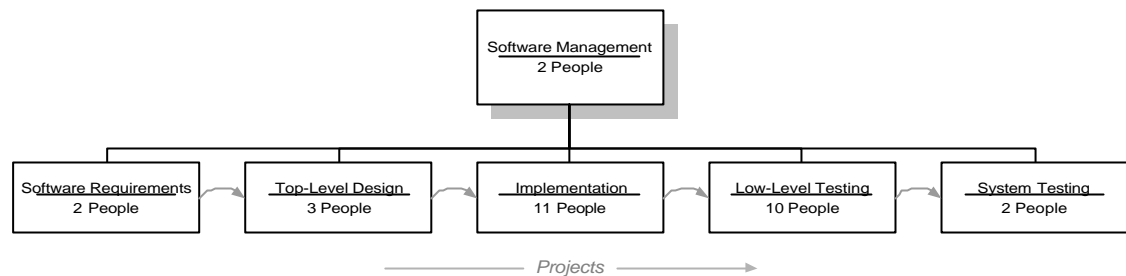


Figure 30: Rolls-Royce – BR700 Control Systems Team Structure Chart

*Process interdependencies* occur within and between the four projects. They exist within projects because the product is developed in sequence through the stages of the software lifecycle (i.e. from Requirements to Design to Code). They occur between projects because the constraints on the order and duration of these transformations have consequent effects on the resources used and products created.

*Product interdependencies* arise from the reuse of components (Requirements, Designs, Code and Tests) between the co-evolving products. The projects share a common product architecture that enabled 80% reuse from *Project A* to *B*, and 42% reuse from projects *A* and *B* to the much different *Project C* (Section 3.7.2). A 1% improvement in the level of reuse achieved meant a saving of around £100,000. This product-line development had therefore delivered significant reductions in cost and timescales.

In the following sections we investigate these interactions within, and between, each level of concurrency using the PEL collection and analysis techniques described in Section 3.5.

### 4.3 Project-Level Concurrency

At the highest level of the hierarchy, there is concurrency between the four control system development projects *A*, *B*, *C* and *D*. Figure 31 shows the total effort spent by the whole software department on each project over the 20 month period, as well as non-project time such as holidays and illness.

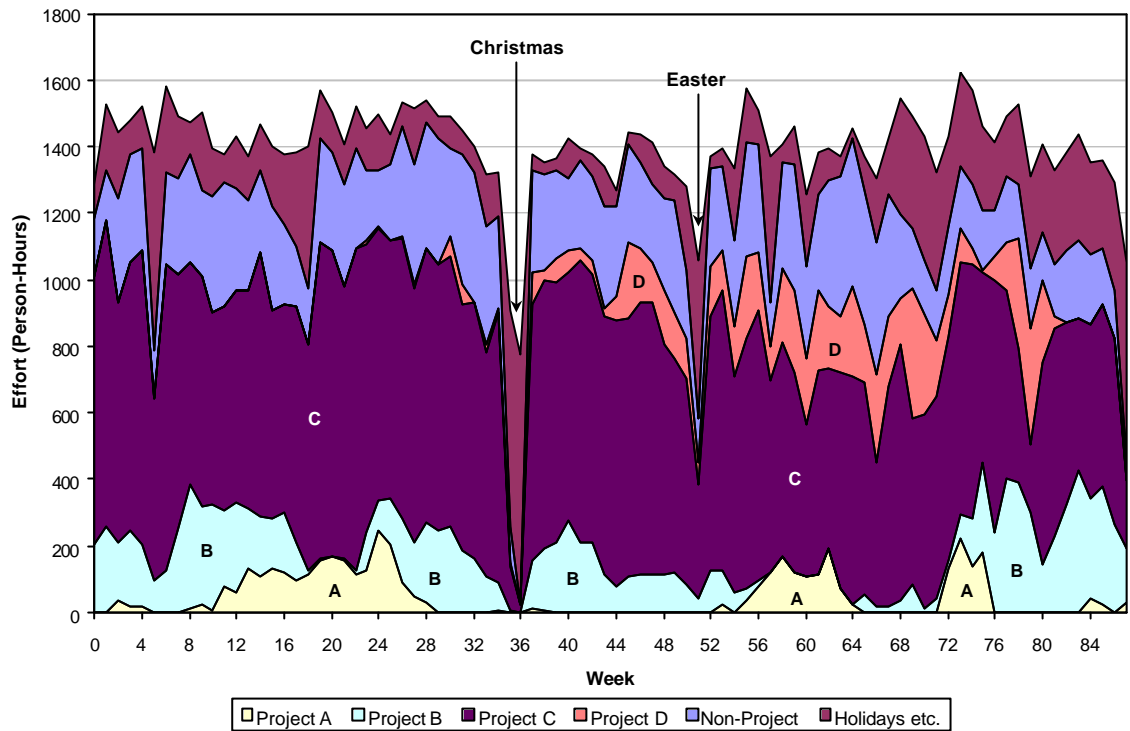


Figure 31: Rolls-Royce – Project Level Concurrency

The variations in effort show how the team divides its time between the four projects at any point in time. The peaks in resource effort represent coinciding work demands causing pressure and overtime. The troughs represent holidays (e.g. Christmas and Easter) or periods where project pressures are relatively low. The overall effort profile is, however, relatively consistent (at around 1400 person-hours per week). This reflects the stability in the size of the software department.

If we now look at the team-level concurrency, Figure 32, we can see the effort by the individual phase teams (e.g. *Code*, *Low Level Test*) over the same period. This shows that the total effort by each team is also relatively consistent over time reflecting the stability in the size of the phase teams. The phase teams are therefore sharing their effort among the concurrent projects. Whilst this is not in itself surprising, it does tell us something about resource *supply* and *demand*.



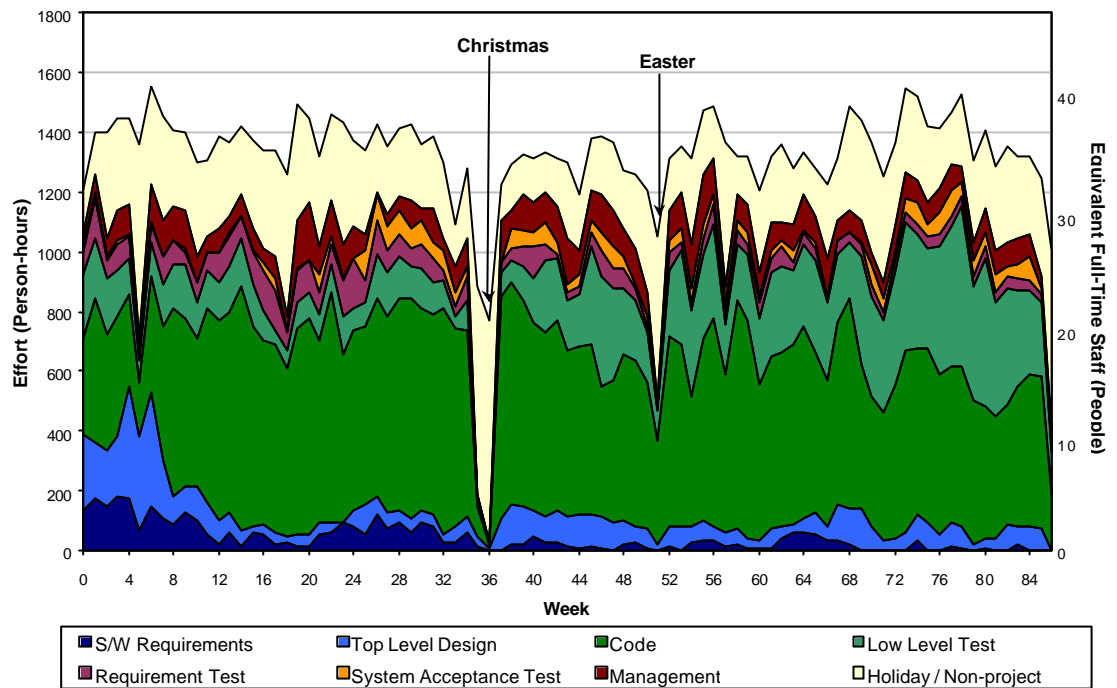


Figure 32: Rolls-Royce – Team Level Concurrency

The *demand* for resource in software development is usually thought of as a Rayleigh distribution (Putnam 1978) illustrated by the ‘total’ resource profile in Figure 33. The demand for engineers slowly ramps-up through requirements and design, peaks during implementation, and then ramps-down through testing and support (McDermid and Rook 1991).

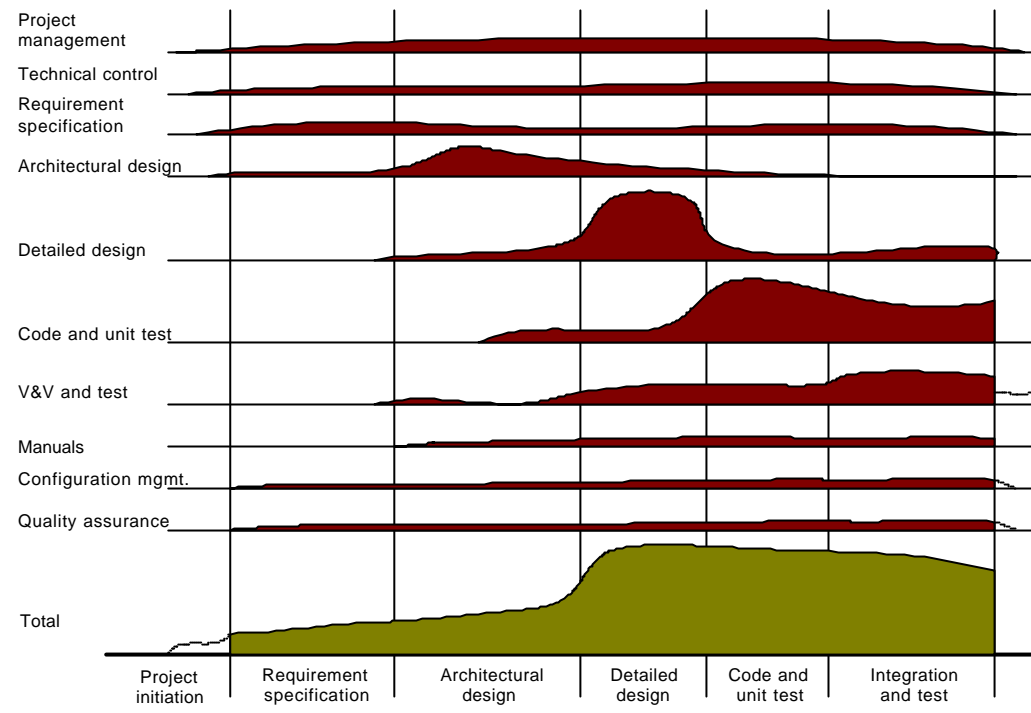


Figure 33: Classical Software Lifecycle Effort Profile (McDermid and Rook 1991)

This conventional perspective assumes that the software process has low levels of concurrency and iteration. The actual staffing profile might follow this distribution in organisations with dedicated project teams in which the same engineers follow the product through each lifecycle stage. In practice, however, the actual staffing profile rarely follows this distribution exactly because of constraints on resource supply.

The *supply* of resource tends in practice to be constrained and less flexible than demand requires. In the development of complex software, as in Rolls-Royce, it is generally not practical or cost-effective to add and remove experienced and qualified engineers according to short-term (weekly or even monthly) changes in demand. Rather, short-term demands are met by the use of overtime and longer-term demands by sub-contracting or recruitment.

The actual lifecycle effort profile for an individual project (*Project A*) is shown in Figure 34. This shows some general similarities with the conventional profile. For example, the peak in effort moves from (left to right) through the phases of the lifecycle over time.

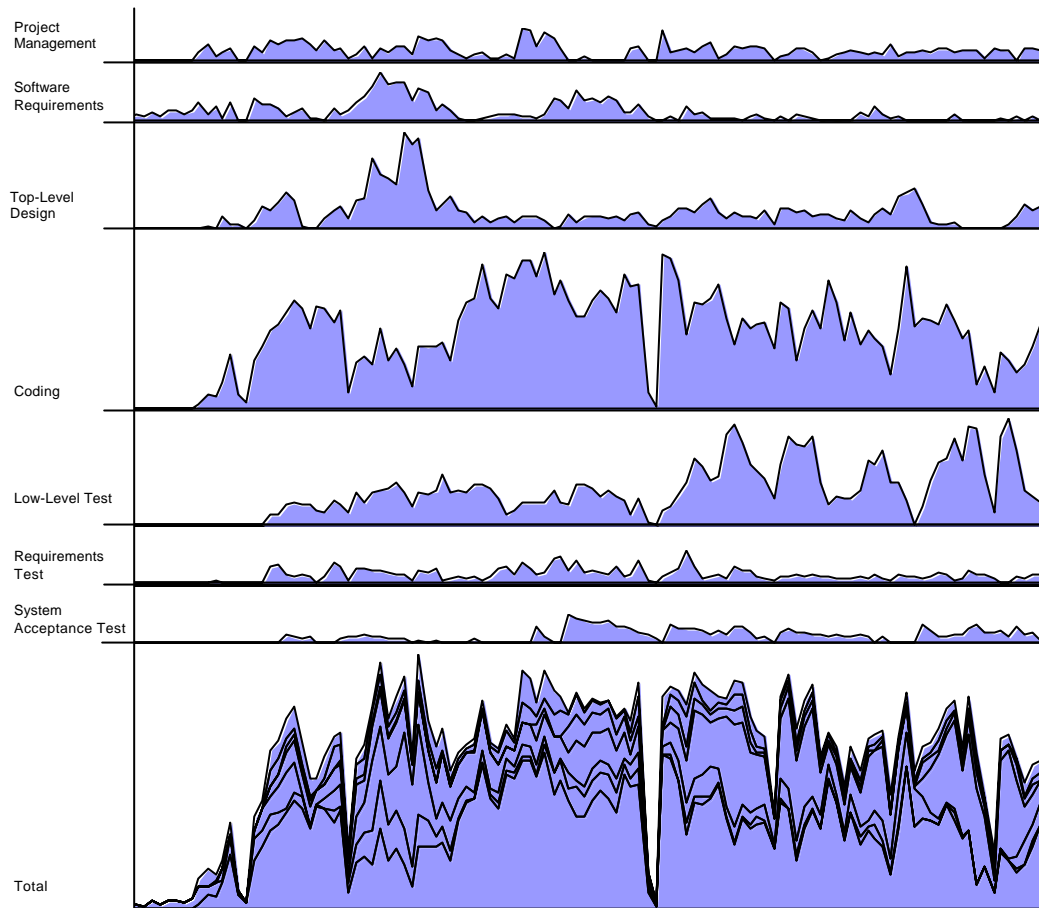


Figure 34: Time-Constrained Software Lifecycle Effort Profile

There are, however, some subtle but important differences between classical effort profiles and the ones we have observed (Figure 33 cf. Figure 34).

- (i) *The phase profiles are compressed to the left (i.e. 'left shift').* Downstream lifecycle phases, such as coding and testing, start very early in the process.
- (ii) *The phase profiles are flatter and longer.* There are significantly higher levels of concurrency between the lifecycle phases.
- (iii) *The phase profiles are jagged.* The peaks and troughs in each phase represent coinciding work demands of other concurrent projects.

These observations illustrate the effects of timescale compression on software development. They also indicate an important distinction in the way projects are planned. Managers are balancing resource levels to meet peaks in demand whilst avoiding the inefficiency of maintaining large teams through less busy periods. The project level concurrency therefore suggests that resource levels stay roughly the same over time (i.e. resource is a relatively known planning variable) but the way they are

allocated between the concurrent projects varies according to the relative demands of the competing activities taking place. To support this theory, it is necessary to look more closely at the lower levels of concurrency.

#### 4.4 Delivery-Level Concurrency

The next level in the hierarchy is that of delivery concurrency. A software delivery represents one-pass through a ‘waterfall’ lifecycle to supply a tested increment of functionality at planned intervals. This *iterative*, or *staged delivery*, lifecycle is used to reduce project timescales by giving a ‘flying-start’ to system and engine testing activities. It also gives faster feedback of problems by testing the software earlier and for longer on the engines themselves. The control system projects are based on six main deliveries:

- (i) Delivery 1.0, *Basic Engine Run*, aims to provide the minimum subset of software functionality (such as the *Starting* and *Control Laws* functions) required to begin engine rig testing;
- (ii) Delivery 2.0, *Full-up Functionality*, adds the remainder of control system functionality (such as the *ARINC* aircraft interface, *Thrust Reverser* and *Maintenance* functions) and solves problems found in the Basic Engine Run;
- (iii) Delivery 3.0, *Engine Test*, provides an opportunity to solve problems from full-up functionality whilst performing further unit, integration and static analysis testing of the software;
- (iv) Delivery 4.0, *Flight Test*, allows any remaining problems to be resolved and further software testing to prepare the control system for flight-testing;
- (v) Delivery 5.0, *Certification*, is used to resolve and test feedback from flight-testing whilst preparing evidence for formal certification of the control system;
- (vi) Delivery 6.0, *Entry-into-Service*, accommodates fine-tuning of the control system before the engine is finally to be used in commercial service.

The chart in Figure 35 shows that in practice, however, many more intermediate deliveries are needed to meet technical and schedule demands. The deliveries are identified by a project reference letter (*A*, *B*, *C*, *D*) and their delivery sequence number (*1.0*, *1.1* etc.). For example, the flight-testing for Project *C* actually used four additional

deliveries (*C4.1*, *C4.2*, *C4.3* and *C4.4*) and Delivery 4.3 was further split into three sub-deliveries (*C4.3.1*, *C4.3.2* and *C4.3.3*). Each additional delivery was used to (i) extend functionality, (ii) accommodate requirement changes, and/or (iii) resolve problems from earlier deliveries. Critically, the start and finish dates for each delivery must be negotiated with the engine development and testing programmes.

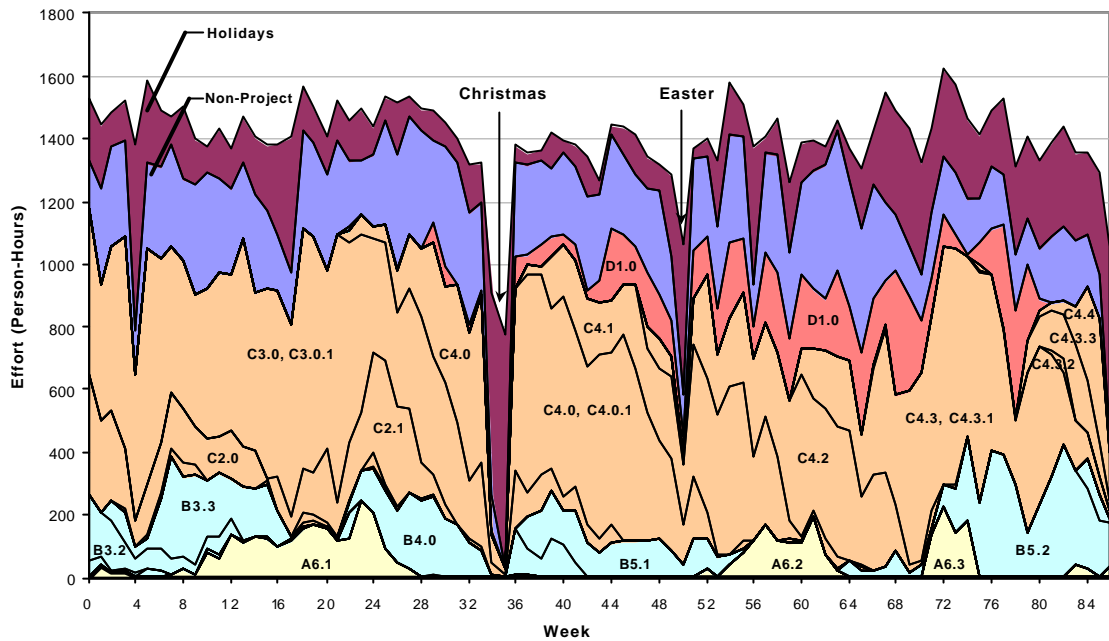


Figure 35: Rolls-Royce – Delivery Level Concurrency

The software deliveries shown in Figure 35 are interdependent due to their reliance on shared resources and products. Resources (e.g. the *Code* team) are typically split between four to six parallel deliveries, across a number of projects, at any point in time. Products are incrementally changed over successive deliveries (e.g. *C3.0* to *C3.1*) and shared between parallel projects (e.g. reuse from *C4.0* to *D1.0*). Hence, there are interacting effects over time between many parallel deliveries within, and between, projects.

The Rolls-Royce process is therefore made up of what are effectively numerous interacting sub-projects. The major implication is that *they cannot be managed independently because of their reliance on shared resource and common work-products*. Put simply, we need to manage the software projects as a portfolio rather than managing them in isolation.

## 4.5 Phase-Level Concurrency

The identification and analysis of project and delivery concurrency gives evidence of interactions between parallel activities, but it does not explain their nature. It is therefore necessary to understand what was happening at the phase level.

In order to reduce delivery timescales, the Rolls-Royce process exploits high levels of concurrency between phases in the software lifecycle. The early (upstream) lifecycle phases release intermediate draft versions of work-products to enable a ‘flying-start’ on subsequent (downstream) development stages. This phase concurrency is planned using a Delivery Template as illustrated in Figure 36.

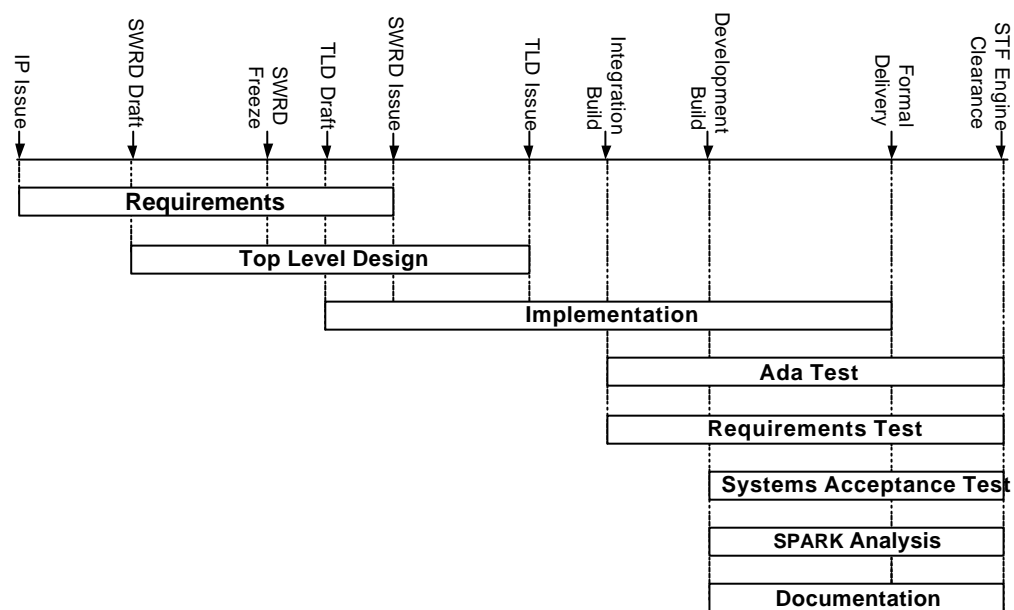


Figure 36: Rolls-Royce – Delivery Template

Each delivery starts with the issue of an Implementation Policy (IP) that controls the change requests (SANs) to be resolved. The Requirements Team then starts work on an initial draft of the *Software Requirements Document* (SWRD). Once this draft is issued, work starts on producing a *Top Level Design* (TLD) for the software, which includes module specifications and definitions of reusable components. For a while, the TLD team works to draft requirements, informally resolving issues with the Requirements team, before the requirements are finally frozen. The configured SWRD allows a draft TLD to be issued for a ‘flying-start’ to begin on code implementation. The coding proceeds in parallel with Requirements and TLD, once again providing a chance to

resolve smaller requirements and design problems, until the SWRD and TLD documents are finally issued.

Each delivery is therefore one pass through a software lifecycle and results in an effort profile like that in Figure 37 for example. This shows considerable concurrency between *Top-Level Design*, *Code* and *Low Level Testing* phases in *Delivery C4.3*.

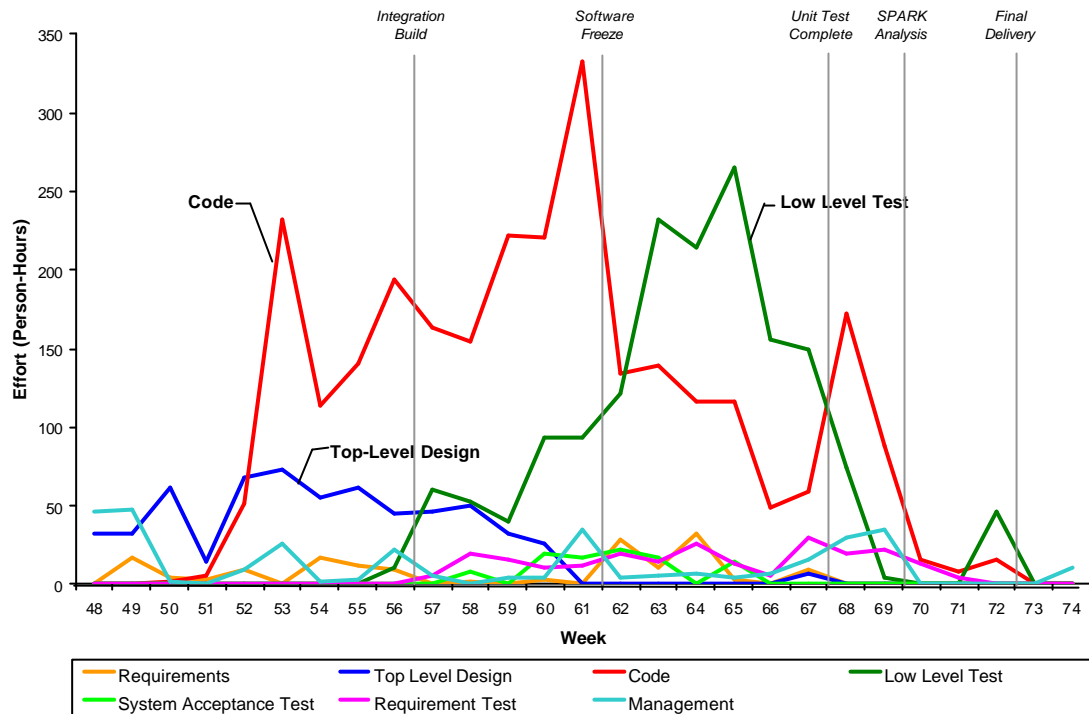


Figure 37: Rolls-Royce – Phase Level Concurrency (*Delivery C4.3*)

The profile in Figure 37 shows distinct peaks in effort before the phase milestones of *Integration Build*, *Software Freeze*, *Unit Test Completion*, and *SPARK Analysis*. This is not, in itself, surprising since we might expect pressure thus effort to increase as the team approaches a milestone (Brooks 1975). However, since the overall size of each resource pool is constrained (Figure 32, Page 89) these peaks in effort must therefore divert resources away from other parallel activities.

It is now possible to explain the complexity of the phase and delivery interactions for all sub projects. To do this, we take a simple example of concurrent deliveries being worked in just two consecutive phases. The result is shown in Figure 38 representing concurrency between a subset of the coding and low level testing phases across many parallel deliveries of the three projects (A, B, C).

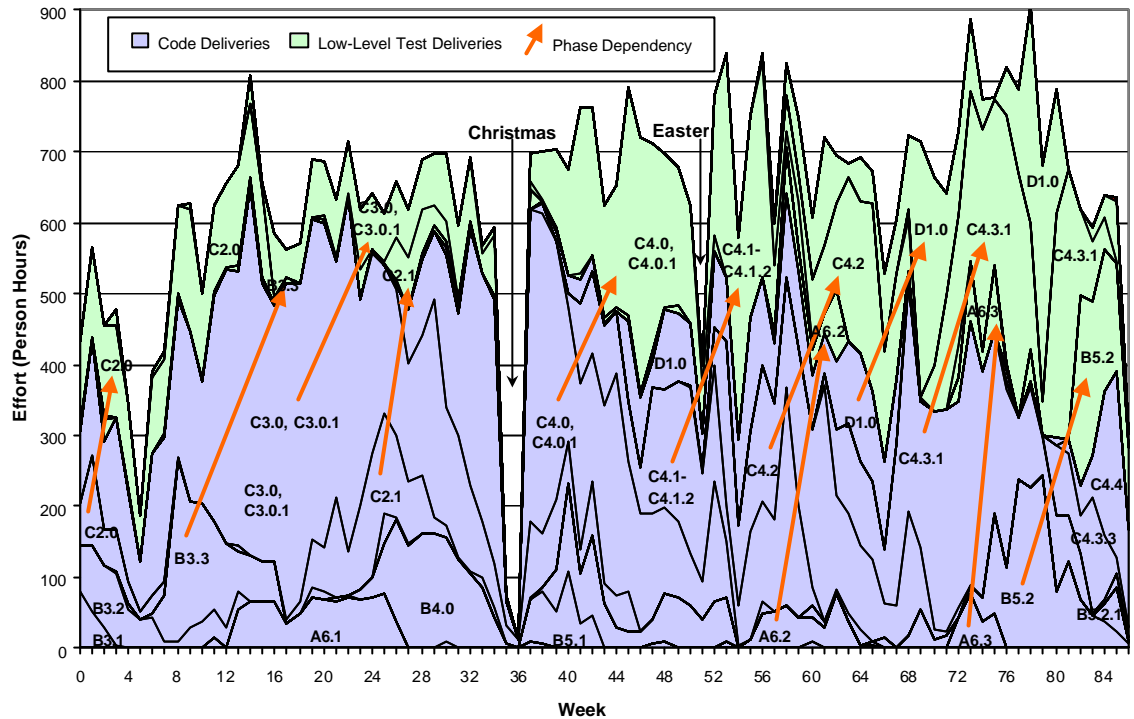


Figure 38: Rolls-Royce – Phase Concurrency between Deliveries

The code team is working on around four parallel deliveries at any one point in time. These deliveries appear to have peaks in their effort profile due to the deadline effect and troughs when resource is required by other parallel activities. The arrows in Figure 38 show the interactions between code and low level testing phases. The low level testing can't begin until an integration build has been completed and it can't finish until the software freeze has been tested. The parallel phases (code and test) across multiple deliveries therefore have numerous process interdependencies.

The process interdependencies within deliveries can have secondary effects between deliveries because of their reliance on shared resources and products. For example, if the code team is unable to meet their milestone the low level test process may be delayed. This can cause knock-on effects to (i) *parallel deliveries* as resources are diverted to help meet the milestone, and (ii) *subsequent deliveries* as attempts to recover schedule delays divert resources away from scheduled work. In the Rolls-Royce process, the timescales appear to have been compressed to such a point that even a minor schedule delay can have significant, even disproportionate, implications. We will return to this observation in the next chapter (Chapter 5).



## 4.6 Task-Level Concurrency

The previous sub-sections have described the Rolls-Royce process at the three levels of project, delivery and phase. The work itself is actually performed at the lowest level in our hierarchy, that of task concurrency. Task concurrency occurs when different SANs are allocated to different team members, and are worked in parallel. This allocation is made by the team leaders of each phase according to the *technical demands* of the task (e.g. engineers may specialise on certain functions of the product) and the *schedule demands* of the process. Furthermore, each task might have both a generative activity (e.g. *Produce Code*) and a complementary evaluative activity (e.g. *Review Code*). The introduction of PEL therefore gives us the fine-grained measures that we need to observe the interactions between these individual tasks. It also provides the connection between high-level policies and low-level behaviour.

## 4.7 Summary of Observations

In this chapter, we have observed the nature of time-constrained software development within Rolls-Royce. The observed process differs from conventional processes (described in published research) in a number of ways.

First, was the extent of the time-constraints imposed on the software process. In Rolls-Royce, this was due to the extreme commercial pressures on aeroengine development lead-times. Since the size, cost and quality of the Control System are largely fixed, the manager had little control over the high-level dynamics of the project.

Second, was the type of development process used to meet these time-constraints. In Rolls-Royce, the time-constraints had been addressed by the extensive application of reuse, concurrency and iteration.

Third, was the extent to which concurrency and iteration had been used to achieve lead-time reduction. In Rolls-Royce, concurrency had been used excessively at the levels of project, delivery, phase and task. Iteration had been used to give increments in software functionality and to fix problems found in testing. The levels of concurrency and iteration far exceed those seen in published accounts of software development.

Fourth, was that concurrency and iteration cause interdependencies between activities that might otherwise be unconnected. These occur because of shared resources, process

dependencies, and product reuse. As the constraints on timescales increase, the process seems to be more sensitive to their effects.

Fifth, was that concurrency and iteration change the way that resources were used. In Rolls-Royce, this meant a 'left-shift' in the effort profile compared to conventional effort models (Section 4.3), i.e. downstream phases started earlier and continued for longer. Resource levels stayed relatively consistent over time, but their allocation changed according to the demands of parallel activities.

Sixth, was that time-constraints changed the way that the process was managed. In Rolls-Royce, they had to manage the projects as a portfolio rather than in isolation.

These observations were the first steps towards understanding the specific problems of time-constrained software development that we discuss in the next chapter.

## Chapter 5:

# Problems of Time-Constrained Development

---

### 5.1 Introduction

In the previous chapter, we used PEL metrics to observe the nature of time-constrained software development in Rolls-Royce. In particular, we showed that concurrency and iteration cause interactions between activities that would otherwise be unrelated in conventional lifecycles. We have attempted to demonstrate the complex effects of these interactions, albeit from the limited perspective of effort. The presence of these interactions is most evident when things go wrong in the process.

In this chapter, we therefore try to isolate the specific problems of time-constrained development as a basis for their resolution. First, we use PEL effort, defect, and size metrics to identify three specific problems (Section 5.2): *Overloading* (Section 5.2.1), the *Bow-Wave* (Section 5.2.3) and the *Multiplier effect* (Section 5.2.4). Second, we compare data from Rolls-Royce and BAE SYSTEMS to see how the constraints in other organisations affect the way software is developed (Section 5.3).

### 5.2 Three Problems of Time-Constrained Development

#### 5.2.1 General Problems

The general problem seen in Rolls-Royce is represented in Figure 39. The pressures to reduce lead-times have been met by significant increases in the degree of process concurrency. This, in turn, introduces more product and process instability from working under pressure and to draft upstream work-products. This compression can cause more rework, demanding more deliveries to fix, and hence increasing the pressures on cycle-time that started the cycle in the first place.

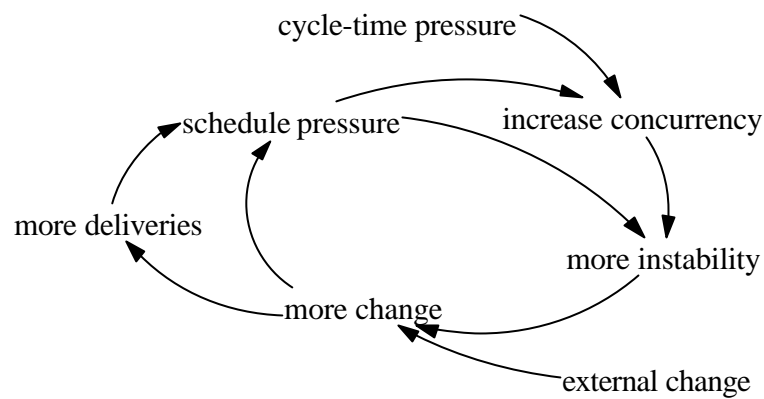


Figure 39: Problem – The Spiral of Process Compression

The problems of the deadline effect and rework cycles have individually been written about elsewhere (Brooks 1975). Rework cycles have been termed by Weinberg as the ‘*boomerang effect*’ since the effects of not getting it right first time ‘*eventually come back and hit you*’ (Weinberg 1992). The difference here is that the problems we describe occur *inside* the software process as a consequence of failing to balance the demands of concurrency and iteration.

### 5.2.2 Problem 1 - Overloading

The first problem is that, since resource levels are fixed and finite, a failure to consider the interacting demands of concurrency can result in resource bottlenecks. In certain situations, where the pressures on productivity are so high as to be unachievable, the process can become ‘*overloaded*.’

An example of the effects of overloading is shown in Figure 40. Three related deliveries are shown on the same timescale along with the original planned delivery date. Delivery C4.1 was scheduled to provide fixes to problems found in the engine and flight-testing programmes. However, unplanned rework from earlier deliveries had to be accommodated in the already over-ambitious timescale. Furthermore, the planned delivery date was fixed and any slippage would cause unacceptable delays in the engine-testing programme. As this date approached, it was clear to the managers that the plans were infeasible and something had to give.

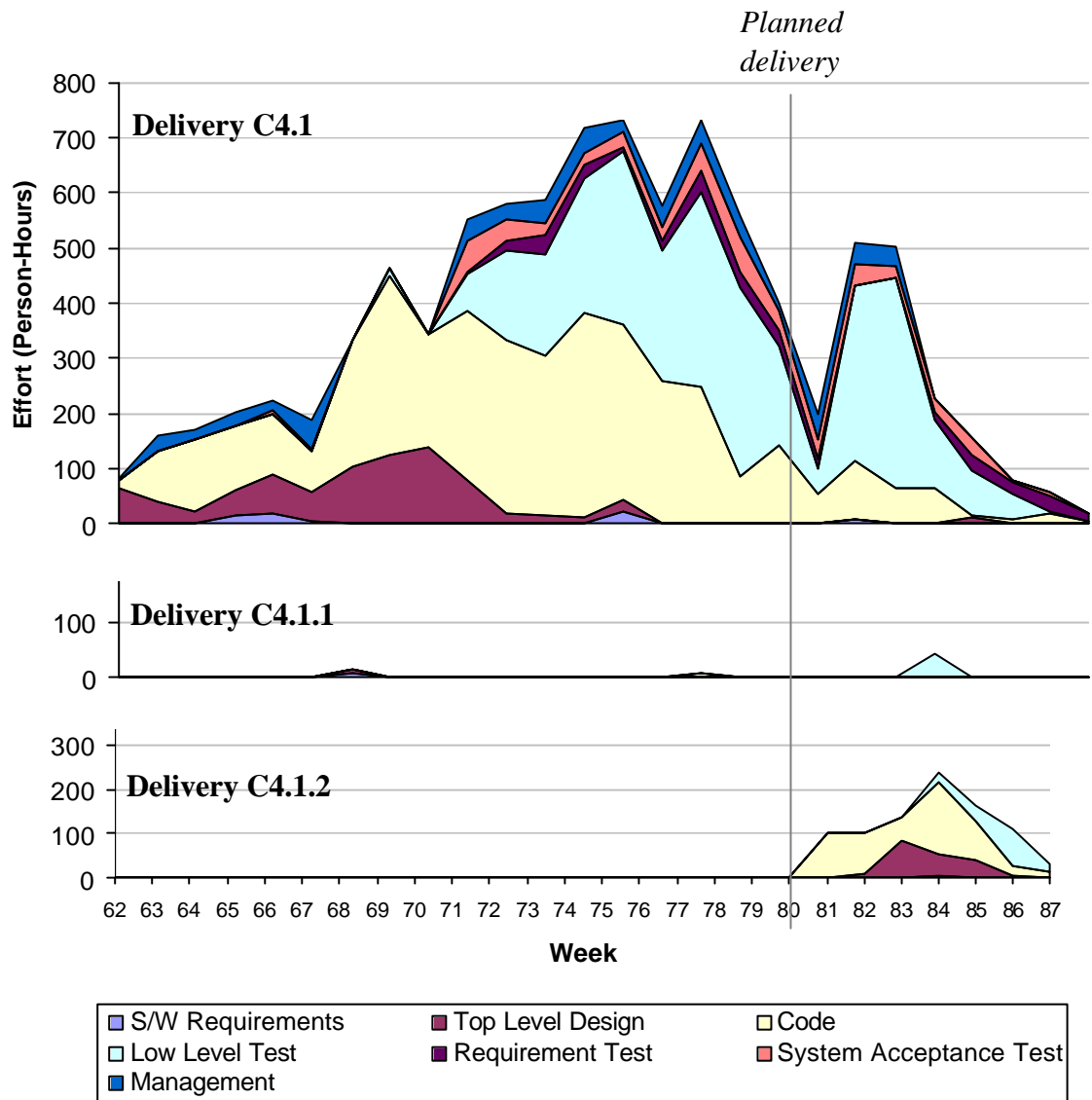


Figure 40: Problem 1 – Overloading

This ‘*snap-point*’ happened when the managers realised that one or more constraints had to be relaxed to cope with excess demands placed on the process. Faced with limited resource, and an unrealistic deadline, a decision was taken to split the planned delivery into three. The existing delivery, *C4.1*, was to provide the critical changes needed to keep the engine testing programme on track. The second delivery, *C4.1.1*, contained one small but important change that was delivered on time, but unit tested in parallel with engine rig testing. Finally, delivery *C4.1.2* addressed the remaining changes that could not be accommodated in the original planned delivery. The one planned delivery of control system software therefore turned out to be three.

In the next sections, we attempt to isolate the consequences of this overloading for parallel and subsequent activities. First, we consider how overloading can be diagnosed and avoided.

The chart in Figure 40 shows the effects of overloading but it does not reveal anything about the causes. An interpretation is that processes have natural limits on performance and any attempt to compress a process beyond these limits will inevitably result in overloading and snap-points. If we had suitable predictive models of time-constrained development, we could model when overloading is likely to occur (i.e. when our plans are unreasonable). We could then take early action to prevent its occurrence (i.e. by smoothing the plan or renegotiating constraints).

As we have stated, a plan is unreasonable when the required productivity of the team is so high as to be unachievable. The exact level at which productivity becomes unachievable is, however, difficult to identify and might vary depending on a particular situation. For example, a team's productivity will vary over time because of pressures and fatigue (Brooks 1975). An alternative is therefore to consider the level of *required productivity* relative to the *actual productivity* of the team. If, for example, the required productivity deviates significantly from the actual productivity then it might be a cause for concern.

In time-constrained development, however, calculating the level of *required productivity* is more difficult than it might first appear. In a simple single project lifecycle it is easy to identify the productivity assumed by a plan as: *planned work / available effort* (where *available effort* = *planned resource* x *planned time*). In a concurrent process the team works on several parallel deliveries each with different workloads and timescales. At any point in time, the required productivity will depend on the work and time remaining across all deliveries being performed. In the next chapter, we therefore propose a dynamic model of time-constrained development to help managers to predict where overloading might occur (Section 6.3).

### **5.2.3 Problem 2 – The Bow-Wave**

A second problem of time-constrained development is what we call the '*bow-wave*.' This occurs when a backlog or 'ripple' of changes move from delivery to delivery and results in increased pressure and risk towards the end of the project.

An example of the bow-wave in the Rolls-Royce projects (A, B and C) is given in Figure 41. This shows the sequence number of the delivery along with the number of change requests that were performed in that iteration. Deliveries 1 and 2 contain few change requests because they largely involve new work or the reuse of existing components. There is a peak at delivery 3 as further changes are raised and worked. As each project iterates through further deliveries, we might expect the number of changes to reduce as the product grows more mature. Instead, the number of changes increased as the team was spending time fixing a growing number of problems from earlier deliveries, i.e. the ripple of the 'bow-wave.' Critically, this bow-wave is internal to the process (i.e. caused as a consequence of a manager's planning decisions) as opposed to conventional change models that reflect the product being used and tested in an external environment.

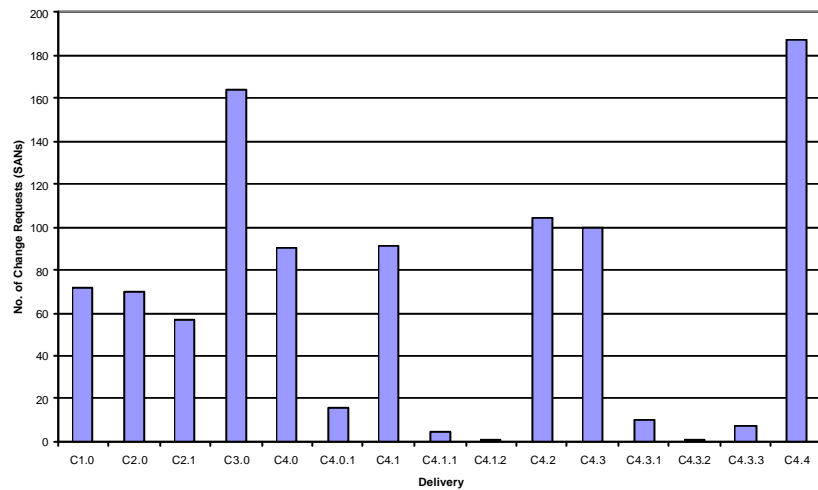
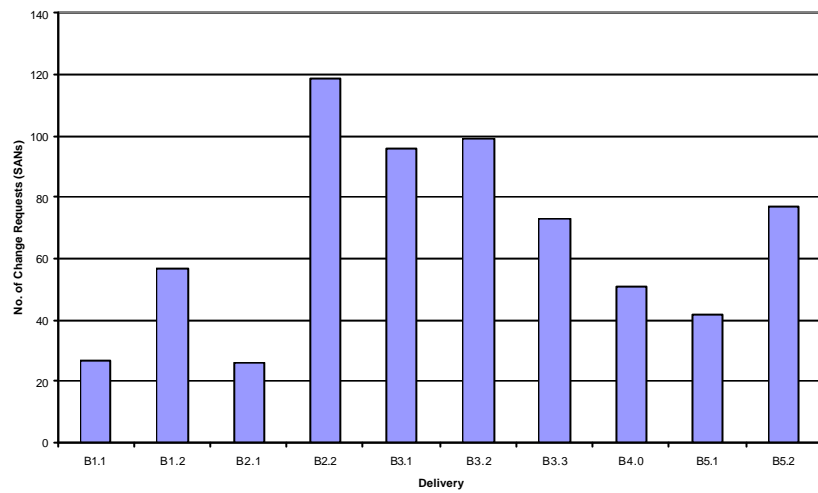
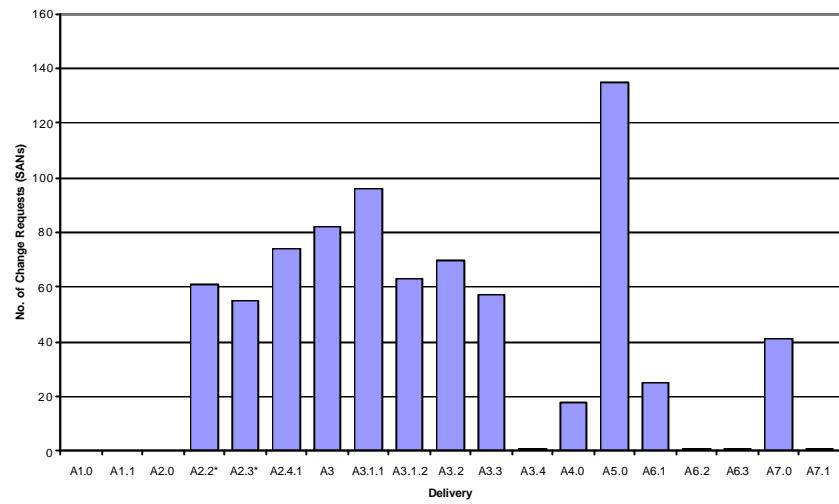


Figure 41: Problem 2 – Bow-Waves



There are a number of possible explanations for the profiles in Figure 41. They could be caused by:

- (i) Early deliveries containing a few large changes, but later deliveries containing large numbers of small ‘fine-tuning’ changes. However, the corresponding effort profiles share a similar profile, indicating that the average effort per change is relatively consistent.
- (ii) Inadequacies in reviews and other quality control activities within each delivery. The findings described in Section 3.6 (Page 67) suggest that this is partly the case (Bardill 1996).
- (iii) Product instability from working to unstable requirements and draft work-products from concurrent upstream phases. The findings described in Section 3.6 (Page 67) suggest that up to 70% of problems are introduced in requirements.
- (iv) Problems introduced by trying to accelerate the process too much, thus causing pressure and poor work quality. There is anecdotal evidence to suggest that the timescale pressures are leading to unnecessary changes.
- (v) Effects of software reuse being used to rapidly achieve full functionality (for engine testing to proceed) and then performing changes to correct known problems. The high levels of reuse have effectively turned development into a change-based process, blurring the distinction between new work and rework.
- (vi) Rapid feedback of problems from engine testing that might have gone undetected for longer had the software not been tested in its complex operational environment. The findings described in Section 3.6 (Page 76) suggest 20% of functions account for 70% of change. These real-time functions are hard to get right because they are complex, highly coupled and difficult to verify until late in the testing lifecycle. They also require fine trimming of calibrations and settings that can only be determined when the software is running in Engine, Airframe and Flight Testing.

In practice, the pressures for lead-time reduction had led to increased levels of concurrency and iteration. This, in turn, led to instability and large amounts of rework that spilled from one delivery to the next. The aim for managers, and this research, is

therefore to determine ways to balance timescale compression against the impact of rework. A logical step is to understand the consequences of these changes in the process, i.e. the *multiplier effect*.

#### **5.2.4 Problem 3 – The Multiplier Effect**

A third problem of time-constrained development is the *multiplier effect*. This occurs because change introduced in earlier (upstream) deliveries can have much greater knock-on effects on later (downstream) activities. If, for example, the process is not well planned or controlled, time pressure in one delivery can lead to large amounts of rework in another. This, in turn, can divert resources away from planned work and increase pressure on parallel and subsequent activities. The effects of change can be amplified by direct and indirect consequences.

The direct consequences of the multiplier are best seen in an example produced by Andrew Nolan of the BR700 software team (Nolan 1999). Nolan used PEL size metrics collected using the Comparator tool (Section 3.7, Page 76) to compare the levels of change between different software deliveries. The chart in Figure 42 shows the *cumulative change* and the *absolute change* in lines of code, files (i.e. Ada packages or modules) and functions (e.g. the *Thrust Reverser*) between deliveries *C3.0* and *C4.4*. The cumulative change is the sum of the individual changes or '*deltas*' in all the deliveries between *C3.0* and *C4.4* (i.e. the percentage change from *C3.0* to *C3.1* plus the percentage change from *C3.1* to *C3.1.1* and so on). The absolute change is the direct delta between *C3.0* and *C4.4*, i.e. the total change if all the changes had all been completed in one delivery.

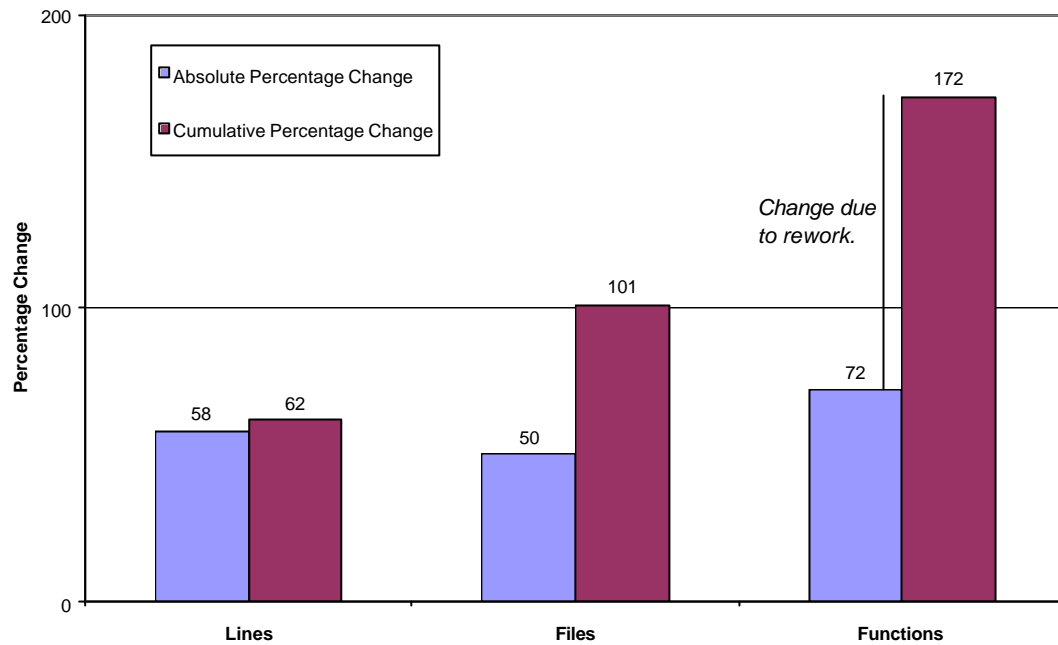


Figure 42: Problem 3 – The Multiplier (Change Between Deliveries)

The height difference between the bars illustrates the additional change that had to be performed because of process iteration. For example, had all the code changes between *C3.0* and *C4.4* been performed in a single release then 58% of the code would have been modified. However, through multiple releases of the software 62% of the code was modified. The additional 4% therefore reflects the amount of code reworked due to iteration.

On a function-by-function basis, we can see far greater differences. Whereas changes to lines of code reflect engineering and design effort, the change to functions represents the amount of additional tests required to verify these changes. For example, changing a single line in the *Thrust Reverser* function would require the whole function to be re-tested. This shows that, had all the changes been performed between *C3.0* and *C4.4* in a single delta, 72% of functions would have been affected and required re-testing. However, through multiple releases of the software 172% of functions had to be tested. That is, between *C3.0* and *C4.4*, for 4% of lines changed an additional 100% of testing was required (Nolan 1999).

The relative profile of changes over the intermediate deliveries is shown in Figure 43. This clearly illustrates the impact of piecemeal change in many small deliveries. Each

software change must go through the software lifecycle and appropriate unit, integration and engine tests must be repeated.

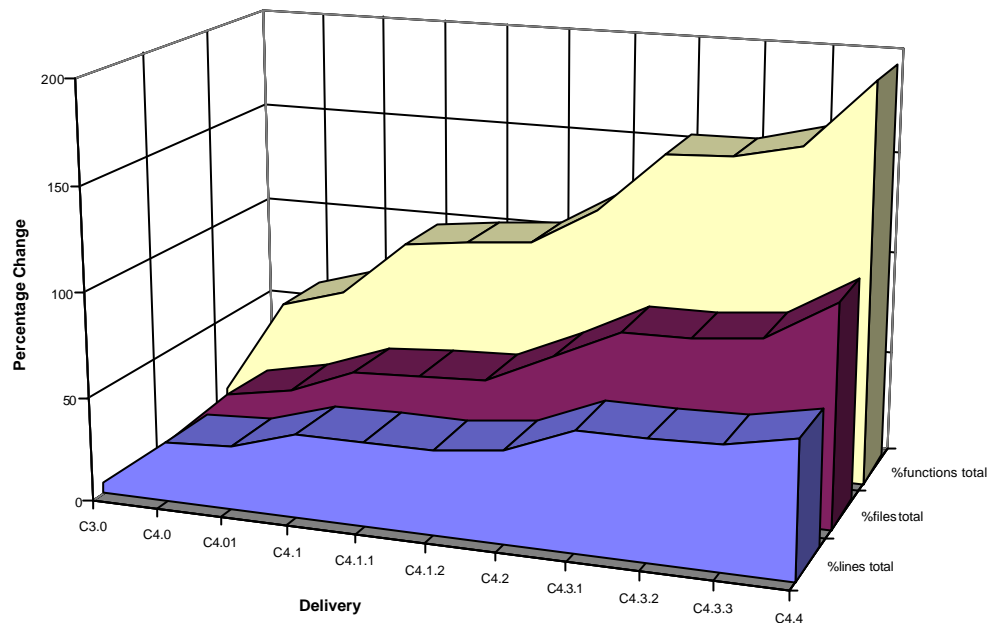


Figure 43: Problem 3 – The Multiplier (Product Change over Time)

The costs of change are therefore significant. This observation is illustrated in Figure 44, which shows the relative effort per software delivery in *Project C*. The high levels of software reuse mean that 6% of project effort is required to get the engine running (C1.0) and 20% of project effort to provide full functionality (C2.0 and C2.1). This means that the engine testing can begin very early in the programme, cutting lead-times considerably. The 27% of effort spent in delivery C3.0 reflects many planned changes for engine rig testing. The remainder illustrates the impact of the bow-wave of change as around 53% of project effort. This represents the cost of performing, re-testing, managing, configuring and documenting changes to the system.

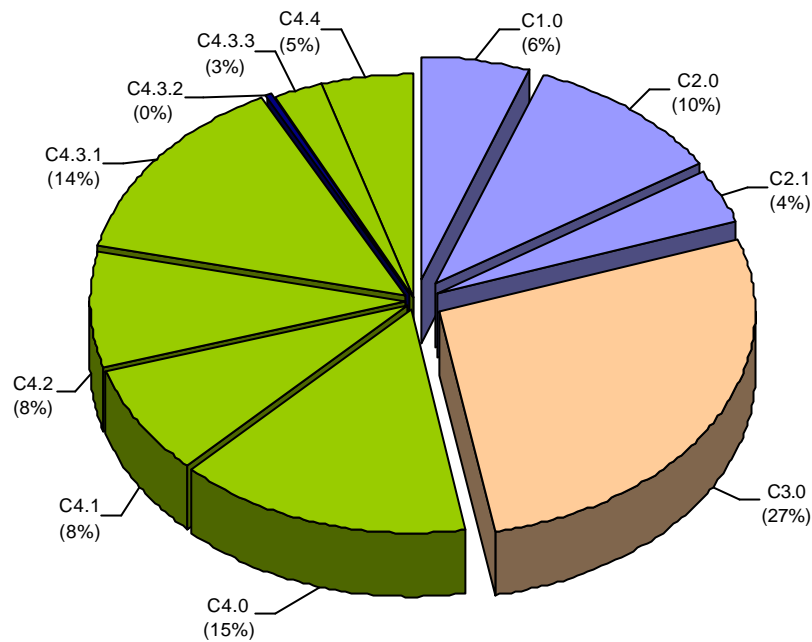


Figure 44: Problem 3 – The Multiplier (Relative Effort per Delivery)

It is therefore clear that the attempts to accelerate the development schedules have resulted in very high levels of change; the effects of which are propagating explosively (McDermid 1981).

The costs and time-delays of performing this direct (primary) rework has indirect (secondary) effects on other parallel activities. These are, however, more difficult to observe. For example, indirect effects include (i) the opportunity cost of engineering specialists performing rework instead of new work, (ii) the cost of quality problems due to the instability in shared work-products and reused components, or (iii) the impact of pressure on productivity and quality in parallel activities which share a common resource pool. Even using the detailed data from PEL, it is difficult to represent these indirect effects because of the complexity of phase interactions that we described in Section 4.5. In the next chapter, we therefore present models to help us describe, analyse and predict the consequence of these effects (Chapter 6).

## **5.3 Generality of Findings: A Comparison with BAE SYSTEMS**

### **5.3.1 Background**

The analysis presented so far has been based on one software development dataset within Rolls-Royce. In order to understand how general our findings are, we have performed some preliminary studies of development in other organisations. The dataset chosen was the Fuel Computer project for the European Fighter Aircraft (EFA) at BAE SYSTEMS.

The Fuel Computer project was selected for two reasons. First, was the author's collaborative work with Clark (Clark and Powell 1999) to share a common measurement approach (i.e. PEL) between BAE SYSTEMS and Rolls-Royce (Chapter 3). Second, the Fuel Computer project is similar to those in Rolls-Royce in that it is concerned with the development of high-integrity, real-time, embedded software for the aerospace industry. The similar nature of the applications means that the development tools, techniques, and standards, are generally common between the two organisations.

In this section, we analyse data collected from the Fuel Computer project and compare the results with our findings from Rolls-Royce (Chapter 4). The data was gathered using a shared PEL language set across to the two organisations (Clark and Powell 1999).

### **5.3.2 Project and Delivery Concurrency**

The levels of Project and Delivery concurrency in BAE SYSTEMS are shown in Figure 45. The Fuel Computer data covers a period of 20 months, matching that of the Rolls-Royce dataset. During this time, two projects (*A* and *B*) were being work on, each consisting of two major increments in functionality (e.g. *A1* and *A2*).

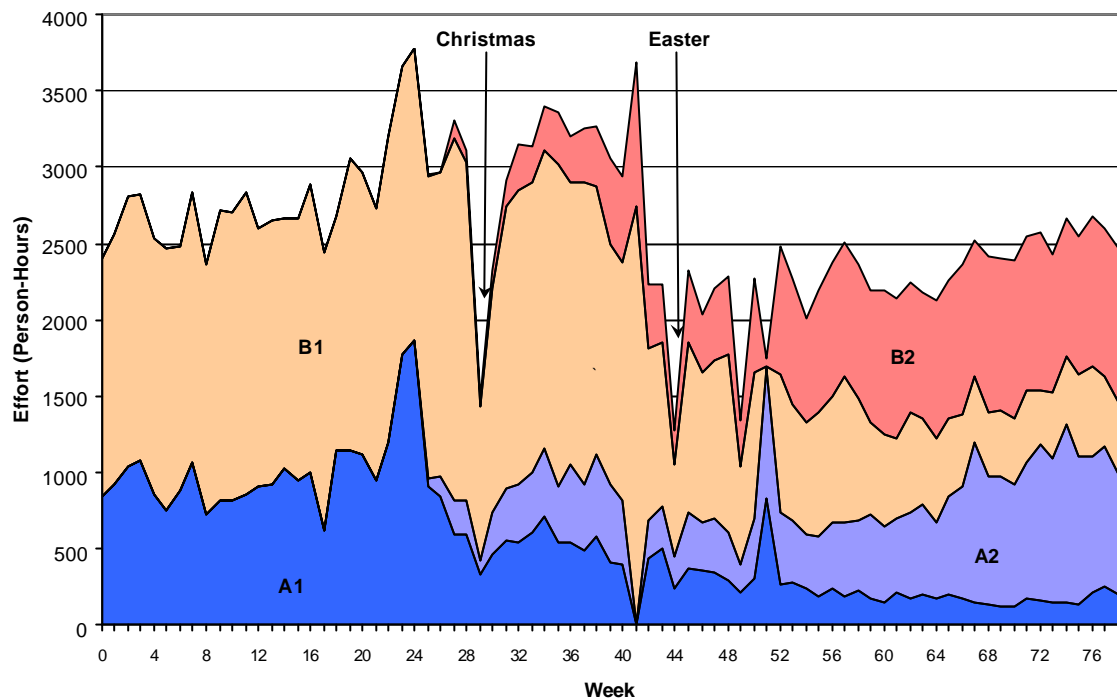


Figure 45: BAE SYSTEMS – Project and Delivery Concurrency

It is immediately clear that the levels of project and delivery concurrency are nowhere near the levels seen in Rolls-Royce (Figure 35, Page 92). This reflects the relatively longer timescales of the military aerospace projects in BAE SYSTEMS, which involve research and development of specific solutions. By contrast, the shorter timescales of civil aerospace projects in Rolls-Royce consist of tailored product-line developments.

The much lower levels of iteration (fewer deliveries) in BAE SYSTEMS is markedly different to that of Rolls-Royce (Section 4.4). The strategy pursued by BAE SYSTEMS is one of ‘right-first-time’ due to the difficulty of testing the Fuel Computer on anything other than the end system (i.e. a military aircraft using the Fuel Computer to circulate fuel around the aircraft body to dynamically maintain its stability during high-speed manoeuvres). This contrasts with the Rolls-Royce ‘right-on-time’ process where the software is produced using many staged-increments to get the engine-testing programme running as early as possible. Hence, different schedule and technical constraints have resulted in different lifecycle strategies in BAE SYSTEMS and Rolls-Royce.

### 5.3.3 Phase Team Concurrency

In the Rolls-Royce dataset, we observed how total resource levels were relatively stable over time (Figure 31, Page 70). The equivalent levels of Phase Team concurrency for BAE SYSTEMS are shown in Figure 46.

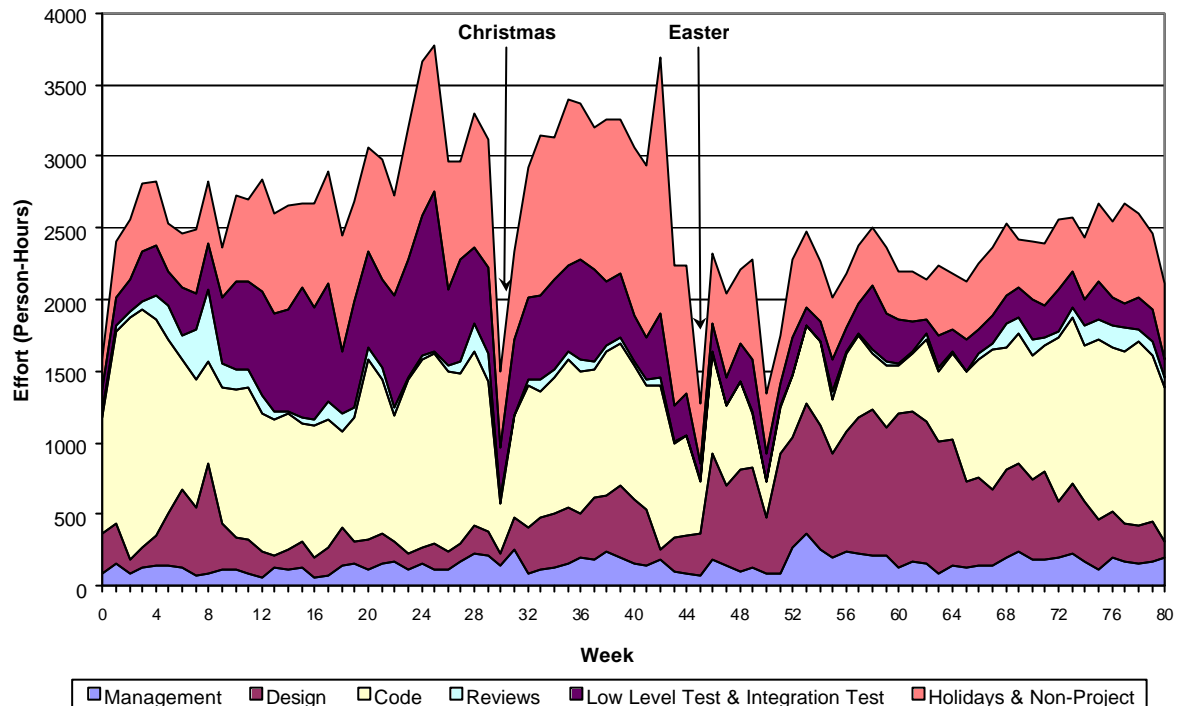


Figure 46: BAE SYSTEMS – Phase Team Concurrency

The chart in Figure 46 shows some overall stability in effort for each phase team. At a high level, this represents organisational stability and the need to maintain a capability of skilled engineers. It also shows that resource is organised around phase teams rather than following the product through all phases. However, there is a clear move of resource from Code to Low Level Testing between weeks 45 and 65. This contrasts with the Rolls-Royce process that showed greater stability with each phase team over time (Figure 32, Page 89). Hence, we can observe how the constraints on the process can influence the resourcing strategies that may be appropriate.

### 5.3.4 Delivery Phase Concurrency

In Section 4.5, we observed how Rolls-Royce had used very high levels of concurrency between lifecycle phases within each software delivery (Figure 37, Page 95). The level



of phase concurrency within an individual software delivery for BAE SYSTEMS is shown in Figure 47.

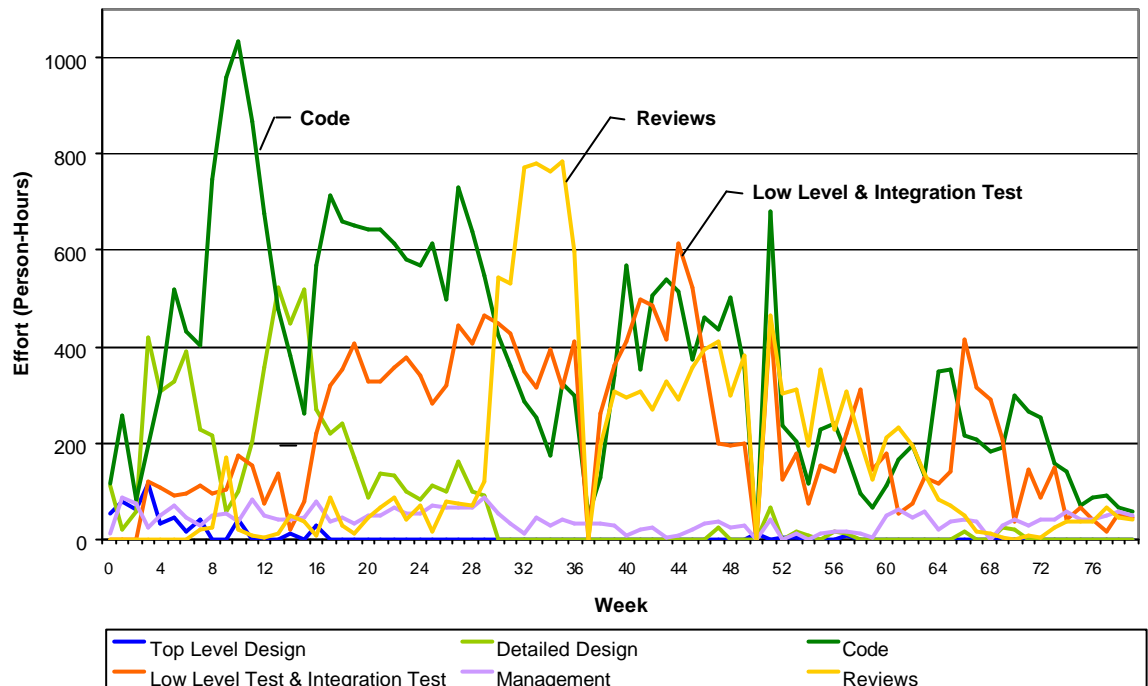


Figure 47: BAE SYSTEMS – Delivery Phase Concurrency

The chart (Figure 47) shows that the BAE SYSTEMS process does use high levels of concurrency across lifecycle phases (e.g. Code and Low Level Testing). However, compared to Rolls-Royce, this concurrency is over much longer timescales and does not show the ‘flying-start’ approach (Section 4.4) that Rolls-Royce used to reduce delivery lead-times. Hence, the BAE SYSTEMS process is more reminiscent of an intermediate evolutionary lifecycle (Mode 2 in Figure 1, on Page 16) than the extreme evolutionary lifecycles observed within Rolls-Royce (Mode 3, Figure 1).

### 5.3.5 Discussion

The results show that, whilst the BAE SYSTEMS and Rolls-Royce processes are similar in the lifecycle activities that they need to perform (same standards, techniques etc.), the way they are actually performed is quite different. In BAE SYSTEMS, a major first release is followed by still reasonably large deliveries of the same target product, increasing functionality and improving performance. In contrast, Rolls-Royce has a rapid first release (using large levels of reuse) followed by multiple incremental releases to meet engine testing requirements. The strategies reflect underlying differences in the

lead-time pressures in military and civil aerospace projects and the nature of the product being developed. An important research question is therefore: what strategies are appropriate in different situations?

A first step in answering such a question has been to introduce common measurement approach so that we can compare strategies and performance in different organisations. The use of PEL has been essential in making the comparisons that we have already described. It is also supporting analysis like that in Figure 48 which shows the relative percentage spend per lifecycle phase of the BAE SYSTEMS and Rolls-Royce processes.

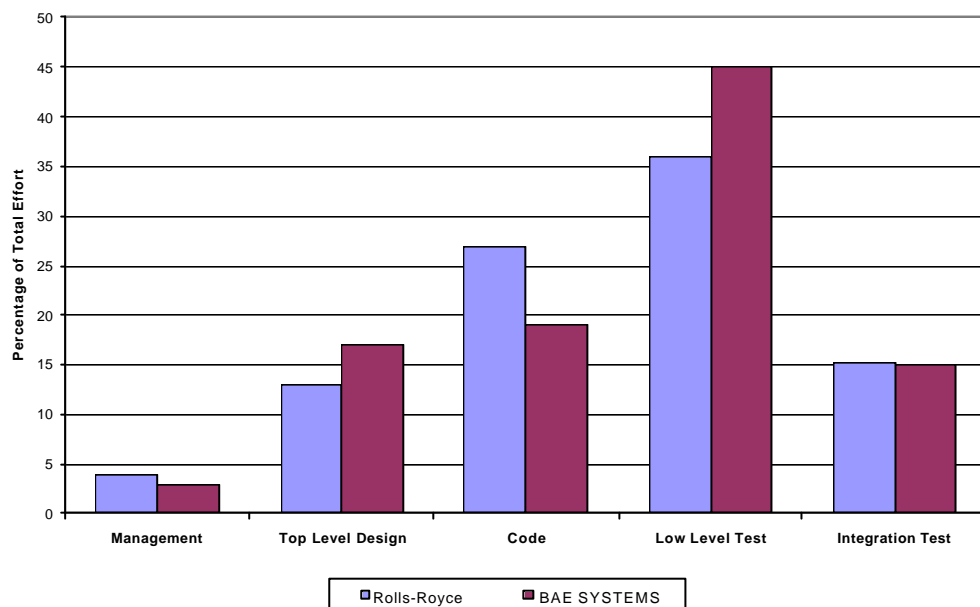


Figure 48: Comparison of Phase Effort in Rolls-Royce and BAE SYSTEMS

The chart in Figure 48 shows how, despite different strategies, the overall percentage spend per lifecycle phase is relatively similar in the two organisations. There are interesting differences, for example, BAE SYSTEMS' use of autocode generation and Rolls-Royce's emphasis on static analysis. More generally, in terms of benchmarking we need to take into account the very different pressures on the organisations in order to make valid comparisons. For example, comparing the two processes on their levels of rework could be misleading as it will not take into account the benefits of accelerating the development process. It therefore shows how we need to understand the relationships between different process constraints (including time-constraints) and the objectives of the processes. The measurement and modelling frameworks that we introduce in this research are therefore an important step forward.

## **5.4 General Implications for Measurement**

In theory, the problems of measuring time-constrained development are no different to conventional processes. In practice, however, we observe that measurement is made significantly more difficult due to the increased number and degree of interrelationships between development activities. This significantly increases the complexity of process behaviour, with corresponding implications for metrics collection and analysis.

The problems of metrics collection arise from the need to be more rigorous in the way data is recorded. In incremental and concurrent development, effort is constantly being allocated among many parallel phase activities within different deliveries across a number of projects. This in-turn causes changes in the qualities of many work-products. There is an inherent trade-off between level of accuracy and collection overhead since engineers must record shorter time slices to reflect what was done. The result is either greater inaccuracies or higher costs of more rigorous data collection.

The problems of metrics analysis arise from the difficulty of isolating the interacting effects between parallel activities over time. It becomes difficult to correlate even basic relationships, such as size and effort, because of the number of interacting variables. For example, is the variation in productivity due to problem complexity or pressure due to parallel activities? Given the complexity of process interactions it becomes difficult to observe clear trends or patterns of behaviour without more detailed measurements of the process.

The introduction of PEL has helped to alleviate these problems by enabling more fine-grained measures to be made without increasing the cost of collection. This allowed observations that would have been much more difficult to obtain, or even obscured, by conventional approaches to effort collection.

## **5.5 Summary of Problems of Time-Constrained Development**

In this chapter, we have isolated three specific problems of time-constrained software development and compared data from Rolls-Royce and BAE SYSTEMS to see how constraints in other organisations affect the way software is developed. There were a number of findings.

First, the application of concurrency and iteration are highly interrelated, and the degree and manner in which they should be applied is far from clear. A failure to manage them correctly can result in overloading, multiplier and bow-wave effects.

Second, *overloading* occurs when the pressures on productivity become so high as to be ‘unachievable.’ At this ‘snap-point’ one or more constraints must be relaxed in order to cope with the demands placed upon the process. It therefore implies that managers need to understand the limits of process capability.

Third, the *bow-wave* occurs when a backlog or ‘ripple’ of changes move from delivery to delivery and result in increased pressure and risk towards the end of a project. It implies that managers need to balance timescale compression against the impact of rework.

Fourth, the *multiplier* effect occurs because changes introduced in earlier (upstream) deliveries can have much greater knock-on effects on later (downstream) activities. For example, a schedule delay, or a change in the quantity of work performed at any point in any lifecycle, has the potential to cause increased allocation of resource that can delay or impact the performance of later phases. It implies that managers need to understand the consequences of rework in terms of costs, time-delays and thus risk.

Fifth, the comparison of metrics from Rolls-Royce and BAE SYSTEMS show distinctly different profiles of concurrency and iteration. Whilst both organisations produce aerospace systems using similar techniques, the constraints and strategies in software development have resulted in very different development processes. The frameworks that we describe in this research are the starting point for investigating what strategies are most appropriate in different contexts.

In the next chapter, we introduce a new modelling approach that enables managers to understand and avoid the problems of time-constrained software development.

# Chapter 6:

## Modelling Time-Constrained Development

---

### 6.1 Introduction

In the previous two chapters, we have investigated the nature and specific problems of time-constrained software development. An important observation was that a schedule delay, or a change in the quantity of work performed at any point in any lifecycle, has the potential to cause increased allocation of resource that can affect the performance of later phases. This dynamic behaviour has significant implications for the way in which software development is modelled (Abdel-Hamid and Madnick 1991; Lehman 1995).

In this chapter, we explore the limitations of current predictive models and propose a new approach to modelling time-constrained development. We start by identifying the problems of conventional models and an alternative model produced by Rolls-Royce (Section 6.2). We then propose a new approach called Capacity-Based Scheduling to help managers to plan and control risk across a portfolio of projects (Section 6.3). Finally, we describe an implementation of CBS using a primitive model to explain its operations by way of example (Section 6.4).

### 6.2 Approaches to Modelling

#### 6.2.1 *Conventional Predictive Models*

Conventional predictive models take the form shown in Figure 49. For example, COCOMO takes the user's predictions of system size and, using 15 productivity drivers (e.g. programming languages, programmer experience), estimates the level of effort required in person-months (Boehm 1981). A planning tool, such as Microsoft Project, is then used to arrange schedules and level resource demands in the context of other parallel projects.

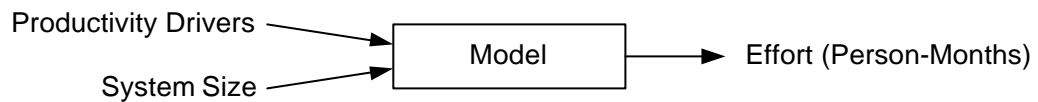


Figure 49: Conventional Predictive Models of Software Development

In practice, the predictive accuracy of these models is poor even for simple development projects. As Kitchenham observes *“There is no evidence that estimation models can do much better than get within 100% of the actual effort during requirements specification and 30% of the actual effort prior to coding”* (Kitchenham 1998). The models are even less appropriate for managing time-constrained development for a number of reasons.

First, in time-constrained development (i) the timescales of the project, and its major increments, are largely fixed and non-negotiable, and (ii) the total level of resource remains, on the whole, quite stable and predictable. Since time and resource (and thus available effort) are ‘known’ variables, having them as outputs of a predictive model is therefore of little benefit to planners.

Second, as time-constrained development is a dynamic feedback system, estimating and planning cannot be treated as separate activities. The ‘point-estimates’ given by conventional estimation models therefore give little indication of the risks involved in accepting these constraints because process behaviour is dynamically dependent on the way the process is planned and controlled, i.e. *“a different estimate creates a different project”* (Abdel-Hamid 1989).

The intrinsic problems of prediction mean that we need to rethink its rôle in the management of software development. As Kitchenham observes: *“senior managers and project managers need to concentrate more on managing estimate risk than looking for a magic solution to the estimation problem”* (Kitchenham 1998).

### **6.2.2 The Rolls-Royce Lead-Time Model**

The limitations of current predictive models, such as COCOMO, led engineers in Rolls-Royce to formulate their own model of time-constrained development. Their goal was not to predict the cost of future projects but to understand if a set of plans were feasible given the constraints placed upon the team.

Their solution was the ‘*Lead-Time Model*’ as shown in Figure 50. The ‘model’ simply shows the *lead-time* of past software deliveries (i.e. the elapsed time in weeks from Top-Level Design to Formal Delivery) against the *size* of the delivery (i.e. the number of changes or SANs performed). By plotting an exponential curve of best fit, the managers could read between the axes to estimate (i) the lead-time needed to complete a given number of changes (e.g. 20 SANs would take 12 weeks), or (ii) the number of changes that can be performed given the available timescales (e.g. given 12 weeks they could perform around 20 SANs).

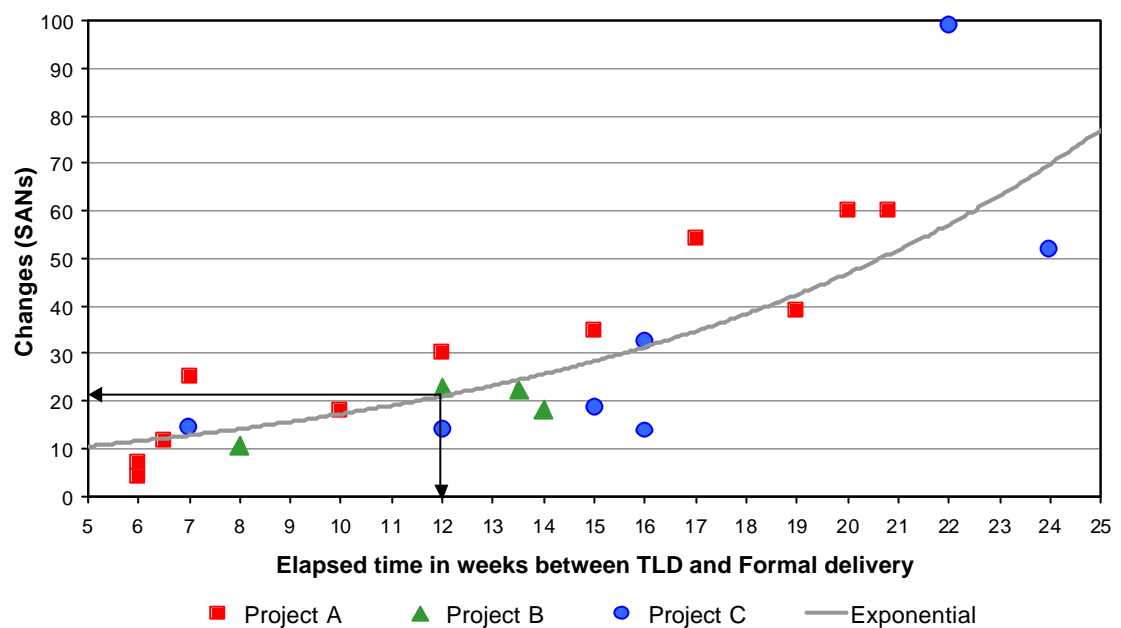


Figure 50: Rolls-Royce – Lead-Time Model

The lead-time model is thus a crude planning tool for determining if a schedule is both challenging and realistic. The more a plan deviates above or below the line then the more over- or under- ambitious the plan might become. For example, a plan that requires 40 changes to be performed in 12 weeks would therefore be cause for concern.

The lead-time model is, however, naïve for a number of reasons.

- (i) It makes no allowance for the level of resource available (if we add more people what would happen to the lead-time?)
- (ii) It does not take into account the impact of concurrent activities on productivity and quality (how did pressure affect the team’s performance?)

- (iii) It ignores the impact of process improvements or other extraneous factors on performance (is productivity increasing over time?)
- (iv) It assumes that all changes are of an equal ‘normalised’ size and complexity (what is the effect of product reuse?)
- (v) It can lead to self-fulfilling prophecies (if we allow 14 weeks, how long will it take?)

Despite these problems, and in the absence of alternatives, the model has been of considerable practical value by enabling managers to judge if a schedule is reasonable.

## 6.3 A Model of Time-Constrained Software Development

### 6.3.1 Overview of Capacity Based Scheduling

We have developed a new model of time-constrained development, called *Capacity Based Scheduling (CBS)*, which addresses the problems of the lead-time approach. The basic concept of Capacity-Based Scheduling is to reverse the inputs and outputs of current predictive models (Figure 51).

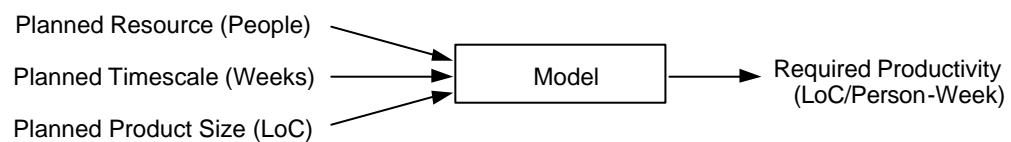


Figure 51: Capacity-Based Scheduling – Concept

By making the constraints on resources and timescales explicit inputs, we can reason about the nature of the capabilities required for their achievement. Critically, we are modelling the constraints on the process in order to compare the *Required Productivity* against the *Actual Productivity* of past projects, and thus indicate the relative ‘risk’ of alternative plans.

The principles behind the model are best explained by example. Consider two planned deliveries of code,  $A_1$  and  $A_2$ . Delivery  $A_1$  needs 1000 LoC to be written and delivery  $A_2$  needs 500 LoC. The work can start on each delivery immediately and the delivery



deadline for each is 10 weeks from now. Supposing we have a staff of six programmers, how should we allocate staff between the parallel deliveries?

At time 0, delivery  $A_1$  requires 100 LoC to be produced each week until the deadline and  $A_2$  requires 50 LoC to be produced each week until the deadline. An obvious and well-motivated approach to allocating staff to the tasks would be to allocate four programmers to  $A_1$  and two programmers to  $A_2$ . This balances the load between the parallel activities; no programmer is under-stressed whilst another is overstressed. In other words, this is *proportional allocation according to need*.

Thus, after the first week with the proportional allocation above, the four programmers working on  $A_1$  deliver (together) 100 LoC and the two on  $A_2$  deliver 50 LoC. If there are no other deliveries starting after the first week then the required rates of progress for  $A_1$  and  $A_2$  are now  $900/9=100$  LoC per week and  $450/9=50$  LoC per week, i.e. the same as before. The same approach can likewise be used for any number of concurrent deliveries; if, for example, another delivery,  $A_3$ , started after the first week then we would need to take it into account in deciding proportional allocations for Week 2.

The *Required Productivity* at any stage can be compared with historical performance and judgements made. For example, requiring programmers to have a very high productivity now could affect the amount of work to be carried out in a later delivery since stressed programmers might introduce more defects. By modelling planning constraints and assumptions, the CBS approach highlights what meeting the deadlines actually means for the staff carrying out the tasks.

### **6.3.2 A Formal Description of the Primitive Model**

The basic structure of our primitive model is presented in Figure 52 that shows the main variables and their causal relationships as arrows. The variables are calculated by dynamic (time-based) equations that describe the relationship between inputs and outputs.

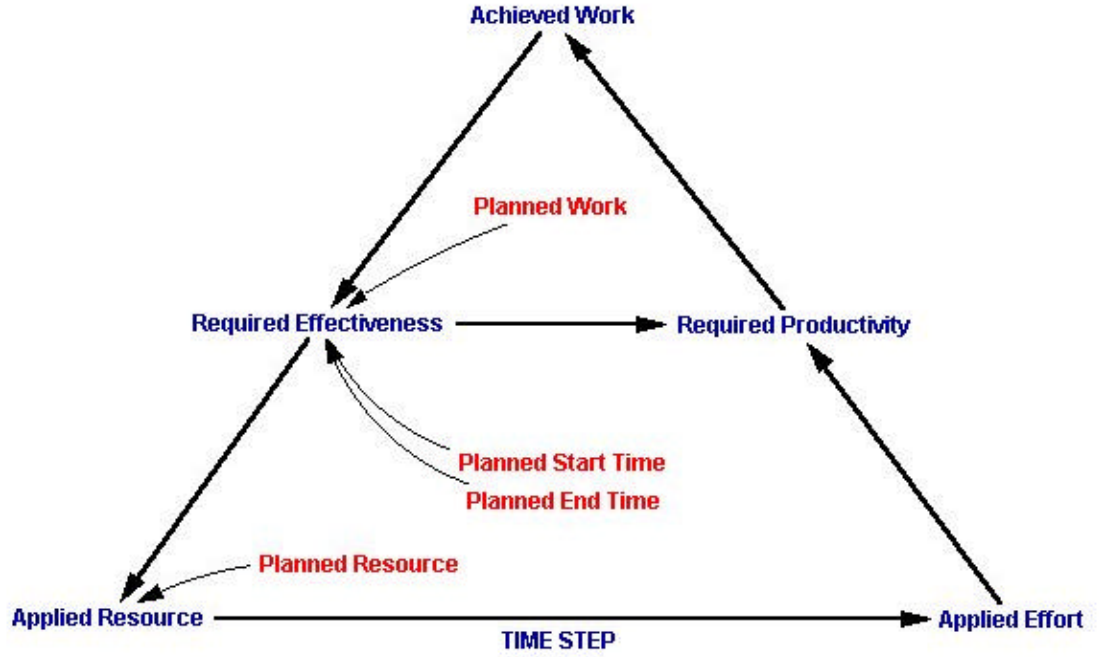


Figure 52: Capacity-Based Scheduling – Primitive Model

The inputs to the model are the *Planned Work* (LoC) to be produced, the *Planned Resource* (people) in each phase team, the *Planned Start Time* (time), and the *Planned End Time* (time) of each phase delivery. The outputs are the calculated values of *Required Effectiveness* (LoC/week), the *Applied Resource* (people), the *Applied Effort* (person-weeks), the *Required Productivity* (LoC/person-week), and the *Achieved Work* (LoC), over time.

The following Sections (6.3.2.1 to 6.3.2.7) formally describe one cycle of the model's operation (corresponding to one unit of time). The full model, implemented using the VenSim simulation environment, is described in Appendix A. Here we concentrate on the operations of the primitive model introduced to explain the modelling approach.

#### 6.3.2.1 Step 1 – Input the Planned Schedule

There are a number of planned deliveries (e.g.  $D_1, D_2$ ).

- $Deliveries = \{D_1, ..., D_n\}$

Each delivery comprises several phases of the software lifecycle (e.g. *Code*, *Unit Test*).

- $Phases = \{P_1, ..., P_n\}$

The planning window comprises a set of time points of from  $0$  to  $T$  with the current time being  $t$ .

- $Time = 0 \dots T$
- $time = t$

Each delivery involves planned work required for its completion (e.g. 1000 LoC for  $D_1$  and 500 LoC for  $D_2$ ).

- $Planned\ Work : Phase \times Delivery \textcircled{R} R$   
(where  $R$  is the set of non-negative numbers)

Each delivery phase has a planned start and end time.

- $Planned\ Start\ Time : Phase \times Delivery \textcircled{R} Time$
- $Planned\ End\ Time : Phase \times Delivery \textcircled{R} Time$

Finally, there is a fixed resource capability available for each phase:

- $Planned\ Resource\ Phase : Phase \textcircled{R} R$  (i.e. a natural number of people)

### 6.3.2.2 Step 2 – Calculate the Required Effectiveness

After  $t$  time steps a certain amount of work has been performed (*Work Achieved*) for a specific *Phase* ( $p$ ) of each *Delivery* ( $d$ ).

- $Achieved\ Work : Phase \times Delivery \times Time \textcircled{R} R$

Each product is associated with the remaining required effectiveness at time  $t$ .

- $Required\ Effectiveness : Work \times Time \textcircled{R} R$  (LoC/time) given by
- $Required\ Effectiveness(p,d,t) =$   

$$(Planned\ Work(p,d,t) - Achieved\ Work(p,d,t)) / (Planned\ End\ Time(p,d) - t)$$

if  $Planned\ End\ Time(p,d) > t$  and  $0$  otherwise.

The *Required Effectiveness* shows the total stress on the development team.

### 6.3.2.3 Step 3 – Calculate the Applied Resource

At each time interval ( $t$ ), the *Planned Resource* for each phase ( $p$ ) is allocated among parallel deliveries ( $d$ ) relative to the *Required Effectiveness*.

- *Applied Resource: Phase x Delivery x Time @ R*

- *Applied Resource (p,d,t) =*

$$\text{Planned Resource}(p) \times (\text{Required Effectiveness}(p,d,t) / \sum_{j=D_1}^{j=D_n} \text{Required Effectiveness}(p,j,t))$$

#### 6.3.2.4 Step 4 – Calculate the Applied Effort

The *AppliedEffort* is the *AppliedResource* over a time interval (*t*).

- *Applied Effort : Phase x Delivery x Time @ R*
- *Applied Effort (p,d,t) = Applied Resource (p,d,t) x t*

#### 6.3.2.5 Step 5 – Calculate the Required Productivity

We are interested in how much the *Required Productivity* might have to deviate from known performance in order to meet the deadline.

- *Required Productivity : Phase x Delivery x Time @ R* given by
- *Required Productivity(p,d,t) = Required Effectiveness (p,d,t) / Applied Effort (p,d,t)*  
if Applied Effort (p,d,t) > 0 and 0 otherwise.

#### 6.3.2.6 Step 6 – Calculate the Achieved Work

The *Achieved Work* in each time-step is assumed to be the amount required.

- *Achieved Work : Phase x Delivery x Time @ R* given by
- *Achieved Work (p,d,t) = Required Productivity (p,d,t) x Applied Resource (p,d,t)*

Finally, the *Achieved Work* to-date for a delivery is given by summing all the achievement for that product to date.

- *Achieved Work To-Date (p,d,t) =  $\sum_{j=1}^{j=t} \text{Achieved Work (p,d,j)}$*

#### 6.3.2.7 Step 7 – Iterate

The model iterates between Steps 2 and 6 until time equals the end time (*T*).

## 6.4 Model Worked Examples

In this section, three examples are shown to demonstrate the operation of the basic model. The first example, Section 6.4.1, revisits our example from Section 6.3.1 to model two concurrent deliveries for a single lifecycle phase (*Code*). The second example, Section 6.4.2, considers the impact of staging our two concurrent deliveries to start at different points in time. The final example, Section 6.4.3, uses the model to study the effect of having staged concurrent deliveries across two lifecycle phases (*Code* and *Low Level Test*). In all cases, the goal is to evaluate whether the planned schedule is feasible given external constraints and internal planning assumptions.

### 6.4.1 Model Example 1: Two Deliveries, One Phase

The input to the model is the planned delivery schedule consisting of the: *Planned Resource*, *Planned Work*, *Planned Start Time* and *Planned End Time*, for each delivery.

The *Planned Resource* is the number of available resource (people) for each phase. We assume that each phase (e.g. *Code*) contains a fixed pool of staffing resources over the time-period under consideration. In this example, the *Planned Resource* is 6 people.

The *Planned Work* is the total amount of work (LoC) to be performed. We assume that this plan includes both planned new work and an allowance for planned rework. In this example, the values of *Planned Work* are 1,000 LoC and 500 LoC respectively.

The *Planned Start Time* and *Planned End Time* are the planned start and end times (week) of each phase delivery. In this example, both *A1 Code* and *A2 Code* start at Week 0 and end at Week 10 (Figure 54a).

The model then iteratively calculates the values of *Required Effectiveness*, *Applied Resource*, *Applied Effort*, *Required Productivity* and *Achieved work*.

The *Required Effectiveness* calculates the average work rate (LoC/week) required to meet the deadline. We assume that the coders are 100% effective, i.e. our estimates of size are correct and there is no rework. In this example, the *Required Effectiveness* of delivery *A1* is therefore 100 LoC per week and delivery *A2* is 50 LoC per week for the duration under study (Figure 54b).

The *Applied Resource* is allocated according to the *Required Effectiveness* over time of the two parallel deliveries. Our strategy for allocating resources does not take into

account other deliveries with start times in the future but only with the active tasks in the next time step (week). We assume that deliveries share a common resource team that is allocated by managers according to the relative size of the task. In this case, the ratio of *Required Effectiveness* is 100:50, or 2:1, so the *Planned Resource* of 6 people is allocated in the ratio of 4 coders on *A1* to 2 coders on *A2* (Figure 54c).

The *Applied Effort* is the product of *Applied Resource* and *Time*. We assume a standard working week with no overtime such that total effort is fixed, therefore the *Applied Effort* mirrors the level of *Applied Resource*. In this example, the *Applied Effort* is consistent at 4 person-weeks for *A1* and 2 person-weeks for *A2* (Figure 54d).

The total *Required Productivity* is equivalent to the *Required Effectiveness* since we assume that progress is actually made evenly at the required rate (i.e. all deadlines are met). In this example, if 25 LoC/person-week is the average *Required Productivity* calculated for a programmer after proportional allocation has been carried out, then it is assumed that they will deliver 25 LoC in the next week (Figure 54e).

The *Achieved Work* is simply the sum of productivity over time. The simple model therefore approximates to linear growth in the product (Figure 54f).

The model is simulated over the period of ten weeks (i.e.  $t=0$  to  $t=10$ ). A manager can then make assessments to see if the loads on each delivery (*A1* 100 LoC/week and *A2* 50 LoC/week), and on the code team as a whole ( $A1 + A2 = 150$  LoC/week), are realistic and achievable given the available resources.

		Week									
		1	2	3	4	5	6	7	8	9	10
Inputs	PlannedProduct[D1,Code]	1000									
	PlannedProduct[D2,Code]	500									
	PlannedStartTime[D1,Code]	1									
	PlannedStartTime[D2,Code]	1									
	PlannedEndTime[D1,Code]	10									
	PlannedEndTime[D2,Code]	10									
	PlannedResource[Code]	6									
Outputs	RequiredEffectiveness[D1,Code]	100	100	100	100	100	100	100	100	100	100
	RequiredEffectiveness[D2,Code]	50	50	50	50	50	50	50	50	50	50
	AppliedResource[D1,Code]	4	4	4	4	4	4	4	4	4	4
	AppliedResource[D2,Code]	2	2	2	2	2	2	2	2	2	2
	AppliedEffort[D1,Code]	4	4	4	4	4	4	4	4	4	4
	AppliedEffort[D2,Code]	2	2	2	2	2	2	2	2	2	2
	RequiredProductivity[D1,Code]	25	25	25	25	25	25	25	25	25	25
	RequiredProductivity[D2,Code]	25	25	25	25	25	25	25	25	25	25
	AchievedProduct[D1,Code]	100	200	300	400	500	600	700	800	900	1000
	AchievedProduct[D2,Code]	50	100	150	200	250	300	350	400	450	500

Figure 53: Model Example 1 – Inputs and Outputs

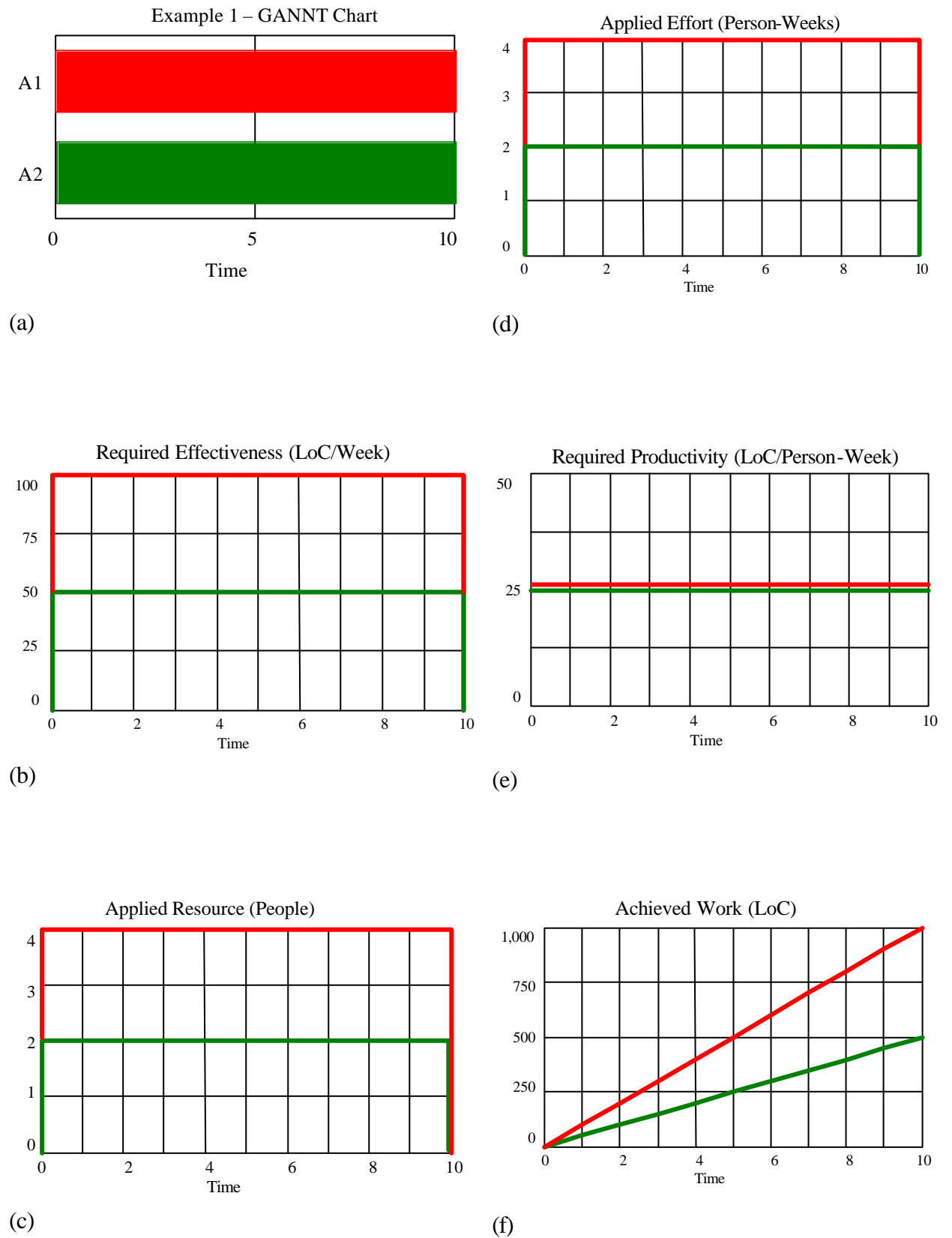


Figure 54: Model Example 1 – Results (Two Deliveries, One Phase)



#### 6.4.2 Model Example 2: Two Staged Deliveries, One Phase

The first example considered two deliveries operating in parallel for the entire duration of the simulation (Example 1). In practice, software deliveries tend to be staged over time in response to the constraints from upstream phases (e.g. the supply of systems requirements), the needs of downstream phases (e.g. the availability of code for engine testing), and to balance the pressure on the development team.

In this example, we consider the impact of staging our two deliveries by delaying the start of *A2* until Week 5, whilst finishing *A1* earlier at Week 8 (Figure 55a). We therefore amend the *Planned Start Time* and *Planned End Time* of the deliveries and run our model again to observe the effects.

The immediate impact of staging the deliveries can be seen in Figure 55b. Having reduced the available timescale for each delivery, whilst keeping the *Available Resource* and *Planned Work* at the same levels, there is a consequent increase in the *Required Effectiveness* of both deliveries.

The effect of the staged deliveries can be seen when we look at the way the *Planned Resource* is allocated across the concurrent deliveries as *Applied Resource* (Figure 55c). All the available resources are applied to delivery *A1* until Week 5 at which time it is split between *A1* and *A2* proportionally relative to the required effectiveness of the deliveries. When delivery *A1* finishes, in Week 8, all the available resource is switched to delivery *A2*. In simple terms, we are therefore modelling the way a manager might allocate their resources and the resulting effort profile (Figure 55d).

The key outputs of the model are the *Required Effectiveness* and *Required Productivity* necessary to perform the given schedule successfully (Figure 55b and Figure 55e). If we compare these results with our earlier example, Figure 54b, we can see that the *Required Effectiveness* has increased from 100 to 125 LoC/Week for *A1* and from 50 to 100 LoC/Week for *A2*. The corresponding demands on *Required Productivity* (Figure 55e) may be relatively trivial and absorbed by a short-term increase in team performance (due to pressure). If the deviation is more significant then there will be a higher risk of schedule problems and overloading. The manager might therefore choose to (i) adjust the profile of *Planned Resource*, (ii) adjust the *Planned Start Time* and *Planned End Time* of schedule activities, or (iii) adjust the levels of *Planned Work* to be

performed in each delivery. The manager can therefore investigate the potential impact of alternative schedules and constraints.

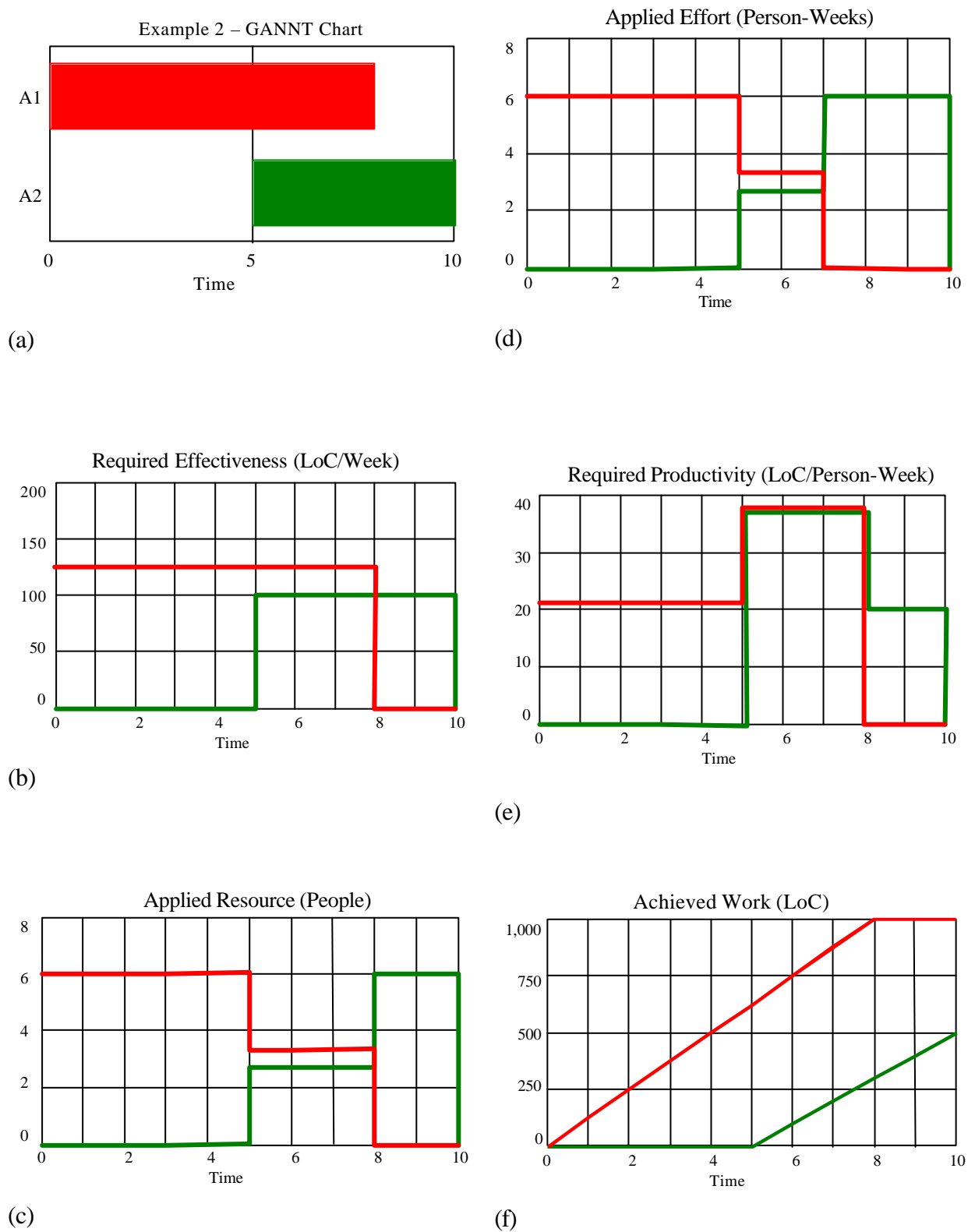


Figure 55: Model Example 2 – Results (Two Staged Deliveries, One Phase)

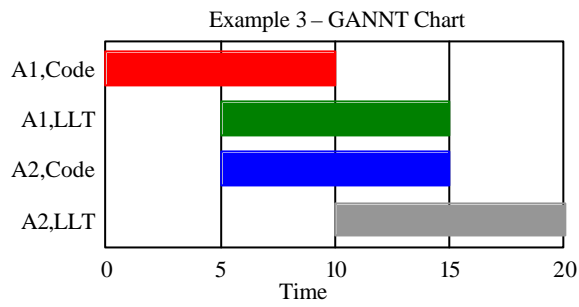
### 6.4.3 Model Example 3: Two Staged Deliveries, Two Phases

Until now, we have only been concerned about one lifecycle phase (i.e. *Code*). In practice, we want to model the many levels of concurrency and iteration that we have observed in industry (Chapter 4). To do this we need to expand our model to consider all phases in the development lifecycle.

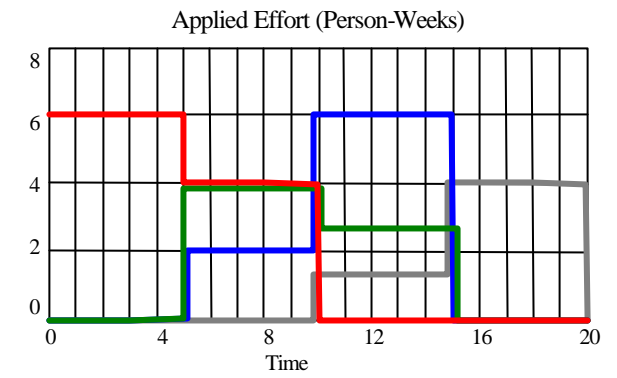
In this example, we consider a planned schedule consisting of two deliveries (*A1* and *A2*) each with two lifecycle phases, *Code* and *Low Level Test (LLT)*, over a period of 20 weeks as shown in Figure 56a. This allows us to model the hierarchy of concurrency.

The operation of the model is the same as we have described in our previous examples. The *Required Effectiveness* (Figure 56b) calculates the work rate needed to complete each individual combination of delivery and phase (e.g. *A1 Code*, *A1 Low Level Test*) on time. For simplicity, we assume in the primitive model that resource cannot move between code and test phases, reflecting the skills restrictions observed in Rolls-Royce (Section 4.2). The level of *Applied Resource* thus uses independent resource pools for code and test phases, i.e. *Planned Resource (Code)* = 6 and *Planned Resource (LLT)* = 4. At any point, the available coders are allocated among parallel coding phases and the available testers are allocated among parallel testing phases, giving the profile in Figure 56c.

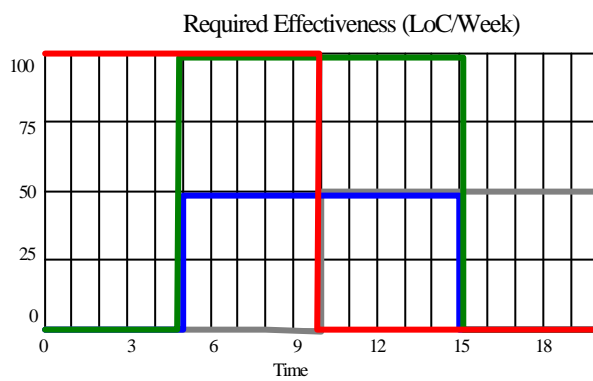
This example illustrates one of the main problems of planning concurrent software development. The manager must ensure that the work-products of upstream phases (e.g. *Code*) are sufficiently mature for downstream phases (e.g. *Low Level Test*) to begin. If the upstream work-products are not sufficiently mature then there may be delays in the schedule. Conversely, if the upstream work-products are “too mature” then we may be missing an opportunity to further compress the schedule. In this example, the code for both products is 50% complete when low level testing begins (as shown by the arrows in Figure 55f). The manager then has the option to determine if this level is reasonable or not based on their observations on past performance.



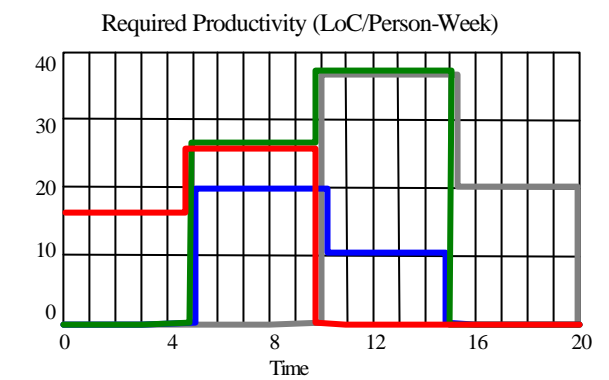
(a)



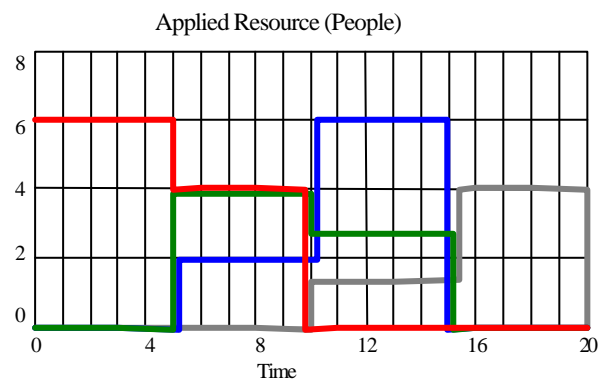
(d)



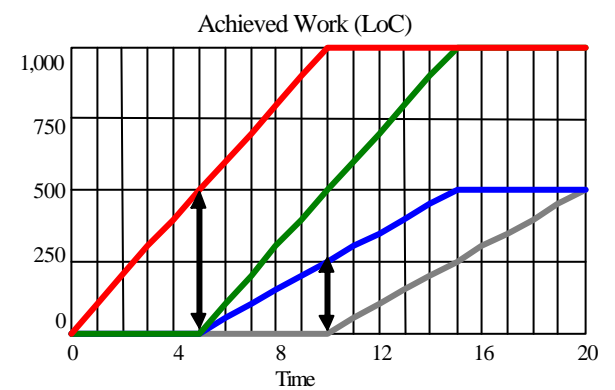
(b)



(e)



(c)



(f)

Figure 56: Model Example 3 – Results (Two Staged Deliveries, Two Phases)

## 6.5 Summary of Modelling Time-Constrained Development

In this chapter, we have identified limitations of current predictive models and propose a new approach to modelling time-constrained development.

First, the problems of current predictive models occur because they treat estimation and planning as separate activities. Whereas, in chapters 4 and 5, we have shown that the way a project is planned and controlled has a significant effect on its performance, i.e. *‘a different estimate creates a different project’* (Abdel-Hamid 1989).

Second, the problems of current predictive models led Rolls-Royce to develop their own ‘lead-time’ model. Whilst this model was naïve, it was of considerable practical value as it enabled managers to roughly judge if a schedule was feasible.

Third, we have developed a new approach to modelling time-constrained development called *Capacity-Based Scheduling (CBS)*. The approach differs from current models in that our input parameters are the planning variables of *Resource*, *Work* and *Timescales*. The planned schedule is modelled to predict the *Required Effectiveness* and *Required Productivity* of concurrent activities. The deviation between the *Required Effectiveness* and the team’s measured *Actual Effectiveness* is then used to tell if the plan is feasible.

Fourth, we used a primitive CBS model to explain the principles of the CBS approach. By modelling just two phases of two deliveries, we were able to explain the complexity of interactions in time-constrained projects. A manager must consider simultaneously the effects of planning decisions on concurrent deliveries in the same phase, and subsequent phases of the same delivery, whilst trying to minimise the overall lead-time.

In the next chapter, we evaluate the validity and utility of the CBS approach using real data from Rolls-Royce and explain how it can be used to improve the management of time-constrained development.

## Chapter 7:

# Managing Time-Constrained Development

---

### 7.1 Introduction

In the previous chapter, we proposed a new approach to modelling time-constrained development, called Capacity-Based Scheduling (CBS). The principles of CBS were explained using a primitive model that, whilst naïve, was sufficient to explain why time-constrained (iterative and concurrent) projects need to be managed as a portfolio. It follows that we need to reconsider our present approaches to management and their suitability in time-constrained software development.

In this chapter, we evaluate the Capacity-Based Scheduling approach and explain how it could be applied to improve the management of time-constrained development. We start, in Section 7.2, by describing our preliminary work to evaluate the utility and validity of the CBS approach using the primitive model and our PEL data from Rolls-Royce. Then, in Section 7.3, we explain how CBS could be used in practice to give improvements in the management of time-constrained development.

### 7.2 Model Evaluation

#### 7.2.1 Evaluation Method

The evaluation method consists of two tests. The first test (Section 7.2.3) evaluates the *validity* of the model by comparing its predictions against the actual performance of the development process, i.e. *can the model describe process behaviour?* The second test (Section 7.2.4) evaluates the *utility* of the modelling approach by investigating the model's ability to indicate the relative risk of a set of plans and thus influence the decision maker in a positive way, i.e. *can the model help managers make better planning and control decisions?*

The evaluation was performed using the primitive model described in the previous chapter (Chapter 6). The plans, consisting of delivery timescales, work and overall

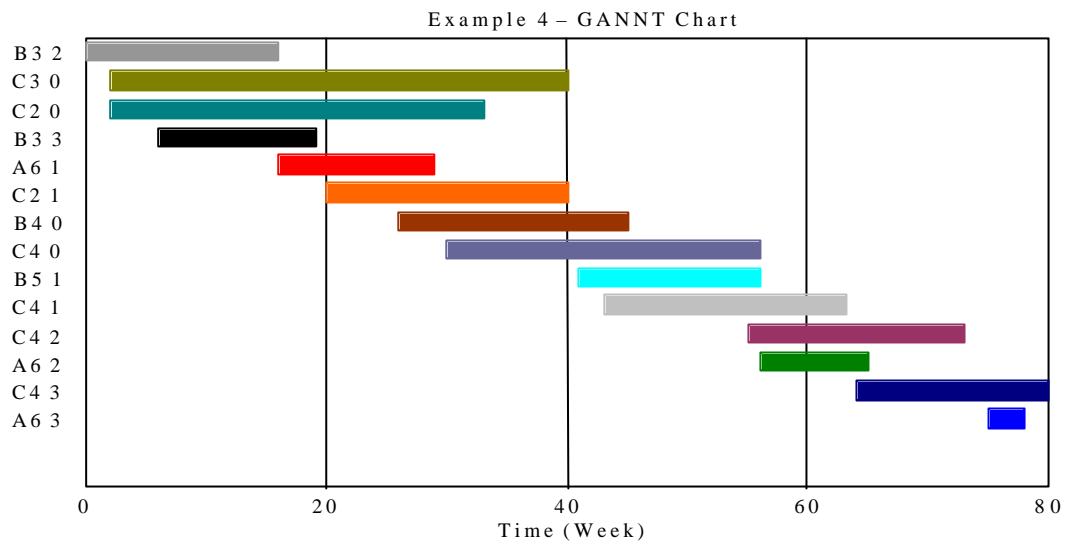
resource levels, were extracted from the Rolls-Royce dataset (described in chapters 3, 4 and 5).

### **7.2.2 Results of the Model Run**

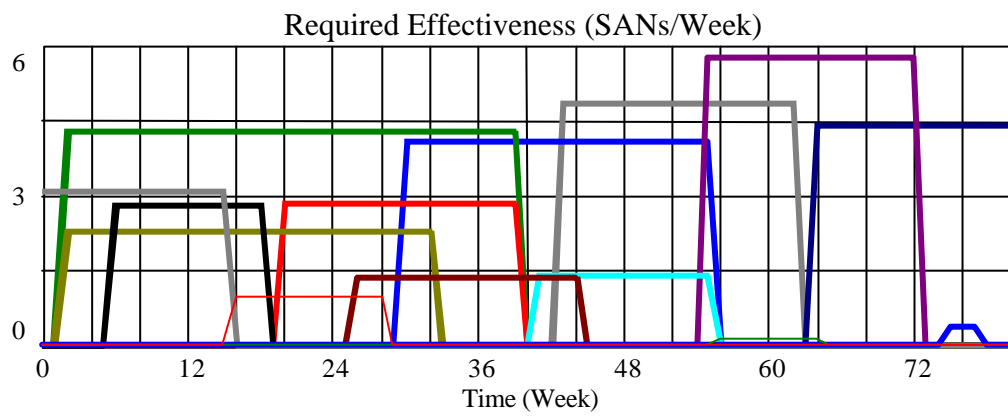
The results of running the model on the Rolls-Royce dataset are shown in Figure 57 and Figure 58. They follow the same pattern as our examples in Chapter 6 (Section 6.4).

- (i) The *Planned Start Time* and *Planned End Time* of each parallel code delivery is shown in the GANNT chart in Figure 57a. The data covers 14 software deliveries over a period of 20 months (80 weeks).
- (ii) The *Required Effectiveness* (Figure 57b) represents the relative size of each delivery and the time available for their completion. Clearly not all deliveries are equal and some are significantly more demanding than others.
- (iii) The *Applied Resource* (Figure 57c) models the manager's resourcing decision by allocating the *Planned Resource* of 15 coders across the concurrent deliveries according to their relative *Required Effectiveness*.
- (iv) The *Applied Effort* (Figure 58a) is the same as that of *Applied Resource* since we are assuming a fixed effort per week and a perfect allocation of resources.
- (v) The *Required Productivity* (Figure 58b) is applied at the level of *Required Effectiveness*, assuming that the work is performed at the required rate to achieve the plan.
- (vi) The *Achieved Work* (Figure 58c) grows linearly over time to meet the target of *Planned Work*.

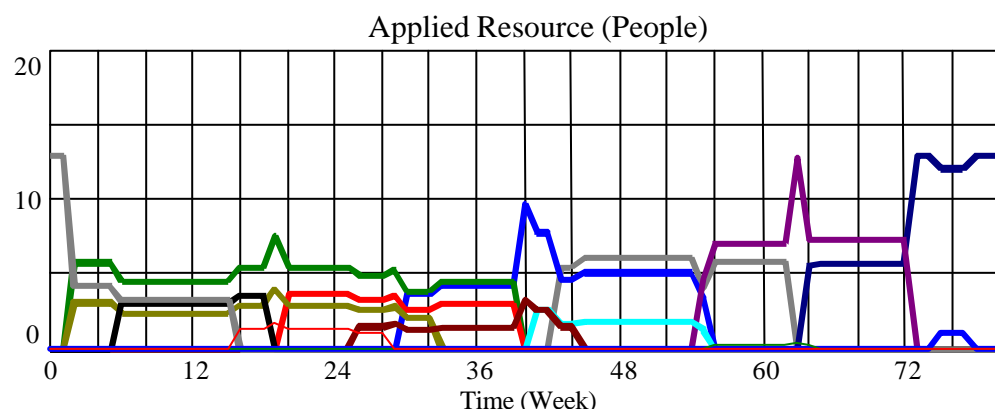
The results were exported to a spreadsheet for comparison against the actual project data collected using PEL. The data was then used to test the model's *validity* (section 7.2.3) and *utility* (section 7.2.4).



(a)



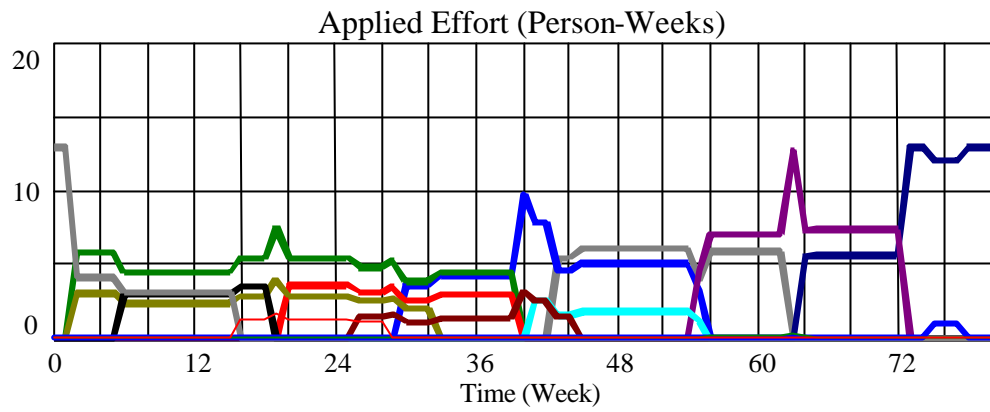
(b)



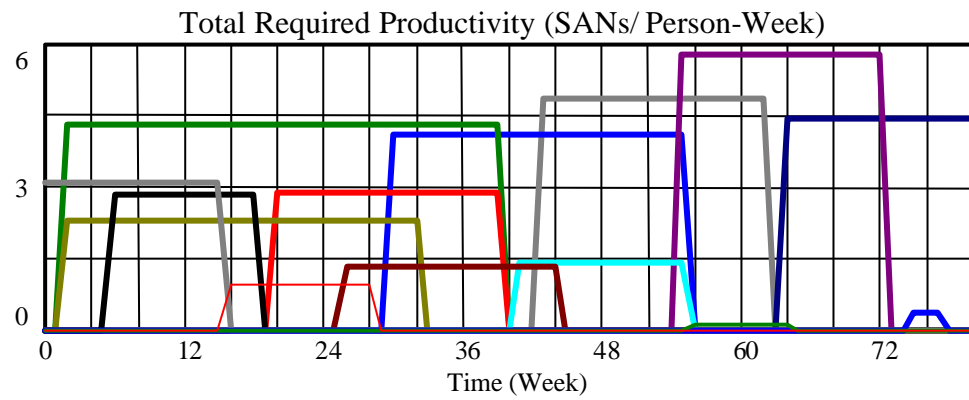
(c)

Figure 57: Model Example 4 – Rolls-Royce Results

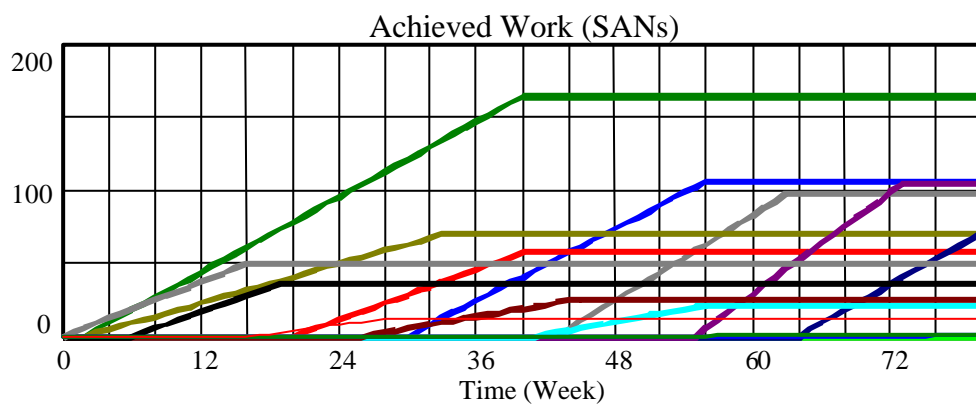




(a)



(b)



(c)

Figure 58: Model Example 4 – Rolls-Royce Results (continued)

### **7.2.3 Model Validity - Analysis of Applied Resource**

The model's validity was tested by comparing the levels of *Applied Effort* predicted by the model (Figure 59) with the *Actual Effort* measured using PEL (Figure 60). Since the input to the model was the planned schedule, the most important output is not the duration of each task but the way in which the available resources are allocated among the parallel deliveries in each time-step. If the model captures a manager's likely resourcing decisions for a given plan, the two profiles should show a similar pattern of effort allocation over time.

A comparison of the two profiles does reveal some similarities in the overall pattern of *Applied Effort* (Figure 59 cf. Figure 60). These are most apparent where the demands of several deliveries coincide to give peaks in resource allocation. For example, the peaks at Week 25 show coinciding demands of deliveries A6.1, B4.0, C2.0 and C2.1, as do the peaks of delivery B4.0 in weeks 26 and 40.

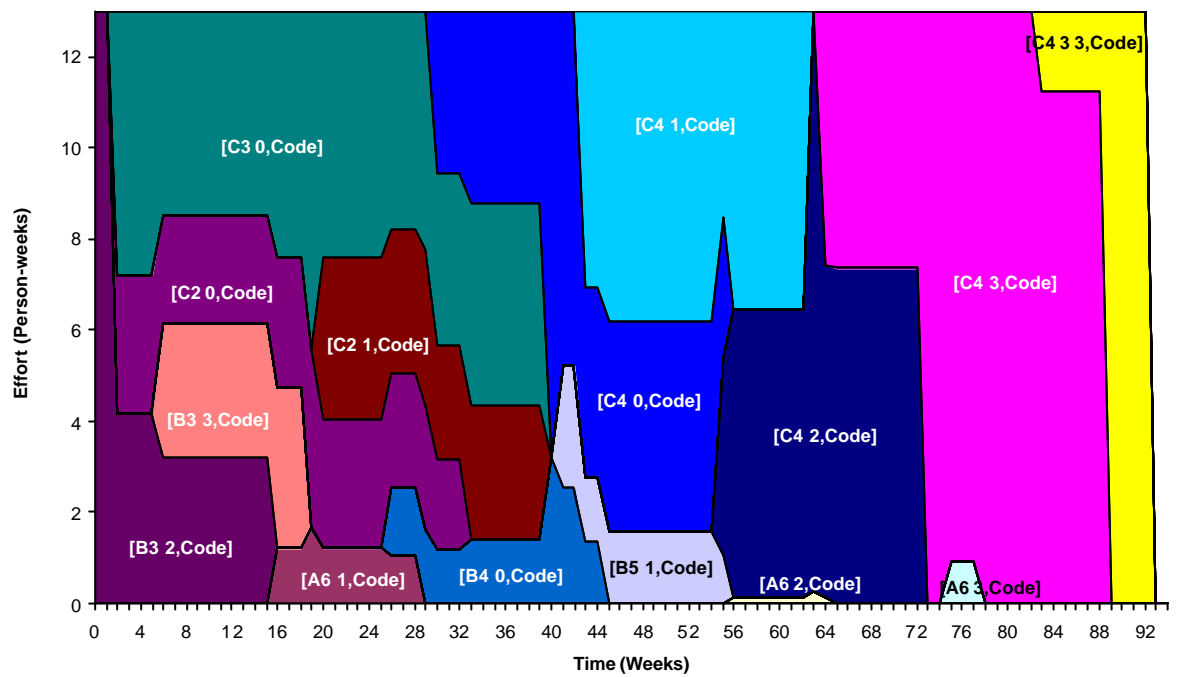


Figure 59: Model Analysis – Model's Allocation of Code Effort

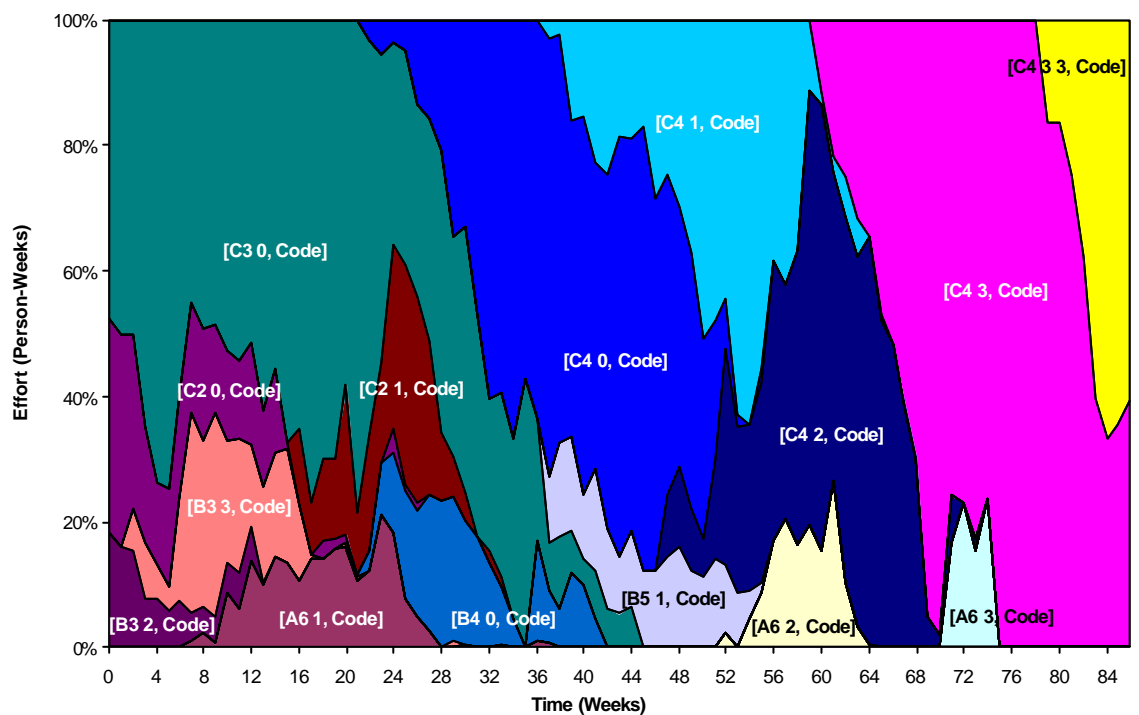


Figure 60: Model Analysis – Actual Allocation of Code Effort

The overall performance of the model can be evaluated by comparing the total predicted and actual allocations of *Applied Effort* between the concurrent deliveries. The results are shown in Figure 61 with the actual and predicted values being expressed as a percentage of the overall effort for the period. Ideally, the predicted and the actual levels would be identical, but an over-allocation for one delivery will cause an under-allocation elsewhere (and vice versa) so errors in the model will appear magnified.

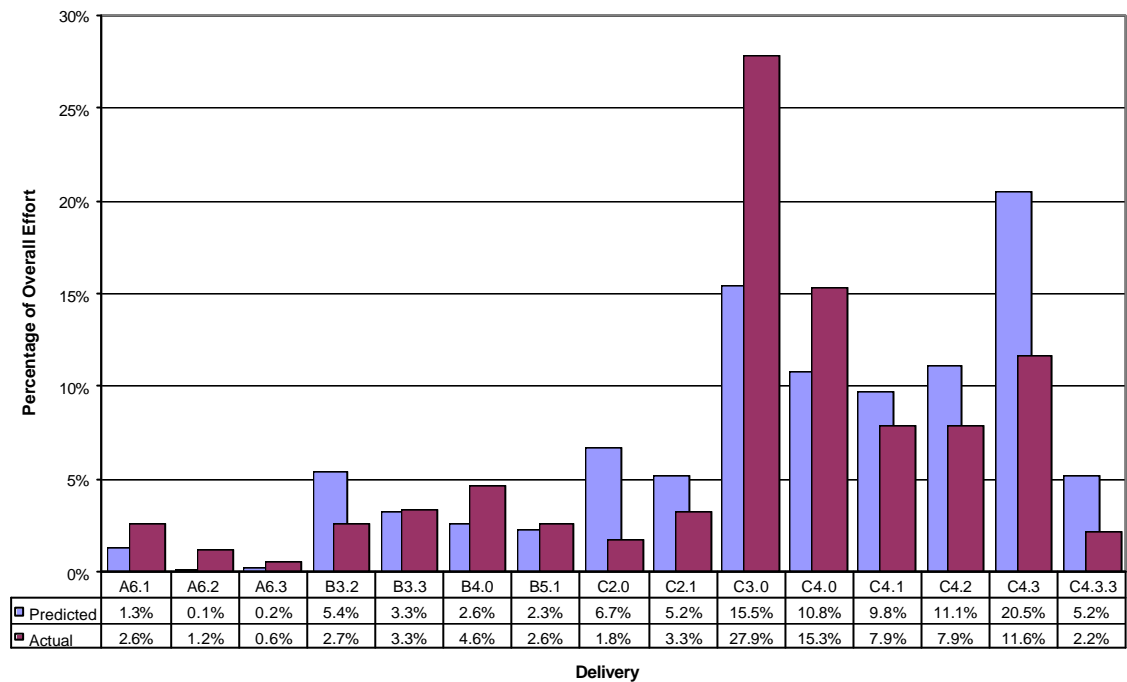


Figure 61: Model Analysis – Actual versus Predicted *Applied Effort*

The results shown in Figure 61 show a reasonable accuracy in predicting the overall allocation of effort between deliveries. This is due in part to the strong correlation between the size of the delivery and the effort required (a principle exploited by the lead-time model). The difference with our model is that it takes into account the time-constraints on, and interactions between, concurrent deliveries. The analysis of resource allocation suggests we can build constraint-based models that, with some calibration, may be of use in decision-making.

## 7.2.4 Model Utility - Analysis of Required Effectiveness

The main output of the model is the predicted level of *Required Effectiveness* that, we propose, can be used to indicate the relative risk of a set of plans. To test this hypothesis, and the utility of our model, we examined the predicted level of *Required Effectiveness* to see if the model could have given insights of use to planners. In Figure 62, we show more clearly the values of *Required Effectiveness* per delivery predicted by the model. The difference in height shows that deliveries are not equal in terms of the demands or ‘pressures’ placed upon them by the plan.

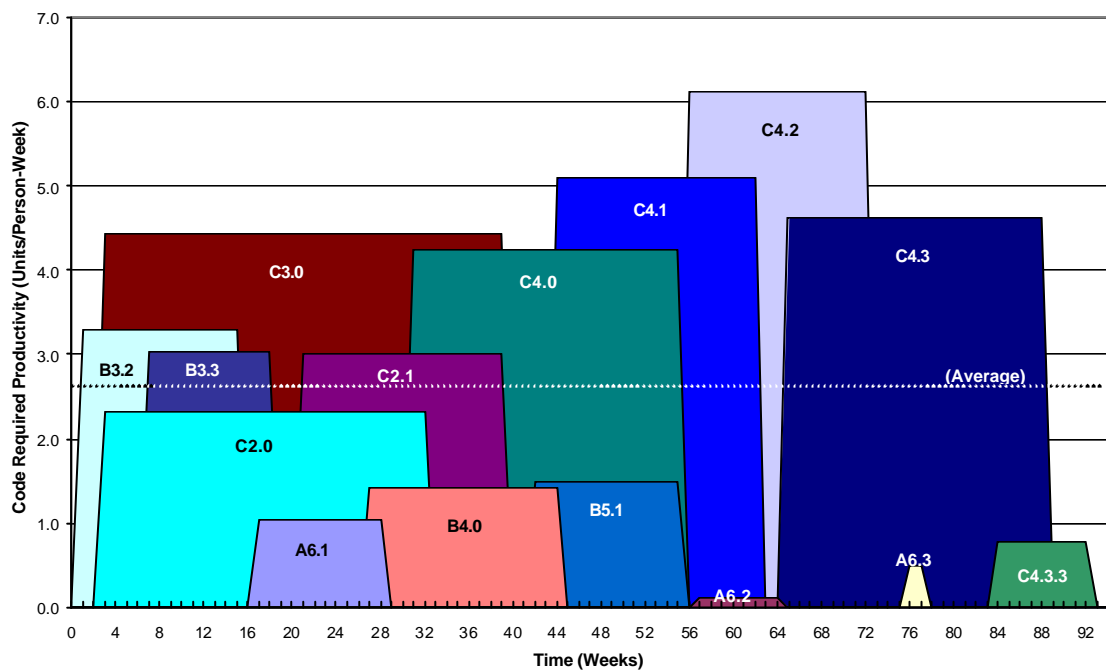


Figure 62: Model Analysis – Predicted Levels of Code Phase *Required Effectiveness*

The levels of *Required Effectiveness* were interesting in that the two most pressured deliveries, C4.1 and C4.2, were the ones we observed in our example of overloading from Chapter 5 (Section 5.2.1). This overloading had occurred where the project pressures appeared to exceed some ‘snap point,’ after which the delivery had to be split in order to meet the unreasonable schedule constraints. Could the model have forewarned that the schedule was unrealistic?

The results showed that deliveries C4.1 and C4.2 were individually demanding, but if overloading had occurred we needed to look at the pressure on the code team as a whole. The results are shown in Figure 63 that gives the total (stacked) *Required*

*Productivity* for all code deliveries. By our definition, to observe overloading we would expect the overall pressure on the code team to be significantly higher as to be unachievable. In fact, the expected peak at *C4.1* or *C4.2* was nowhere to be found and, despite *C4.1* and *C4.2* being individually demanding deliveries, the model did not predict the existence of overloading at the code level.

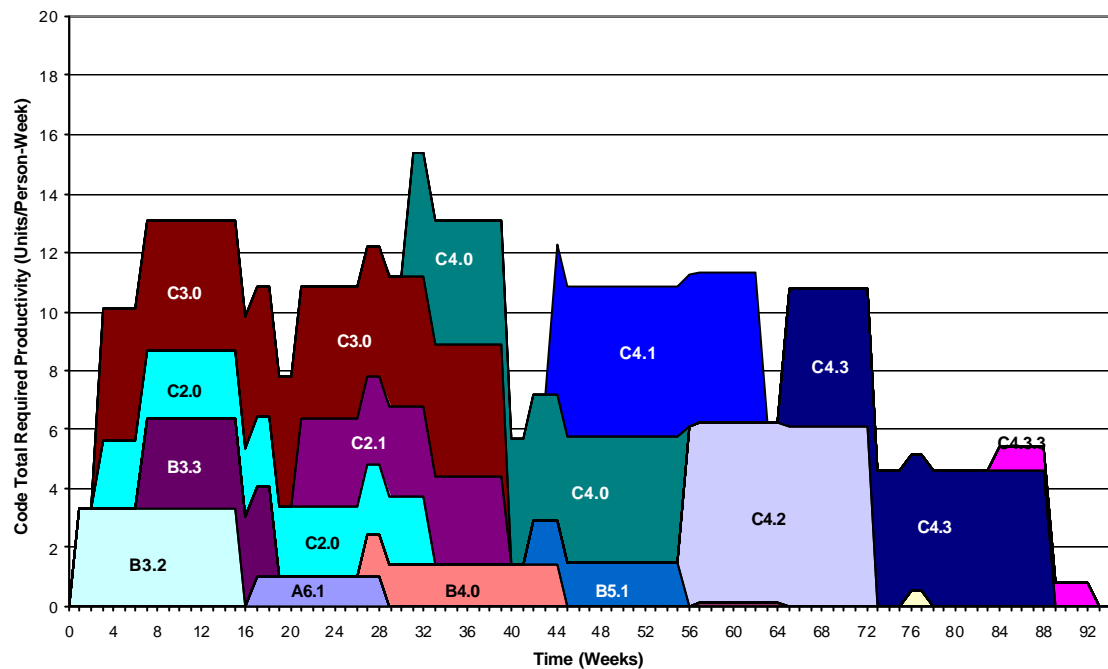


Figure 63: Model Analysis – Total Predicted Code Phase *Required Productivity*

The results at the code level were surprising but our experiment also considered the effects of concurrency in the Low Level Test phase. The model's *Required Productivity* for the Low Level Testing Team is shown in Figure 64. The results showed a significant peak toward the end of Low Level Testing for *Delivery C4.1*.

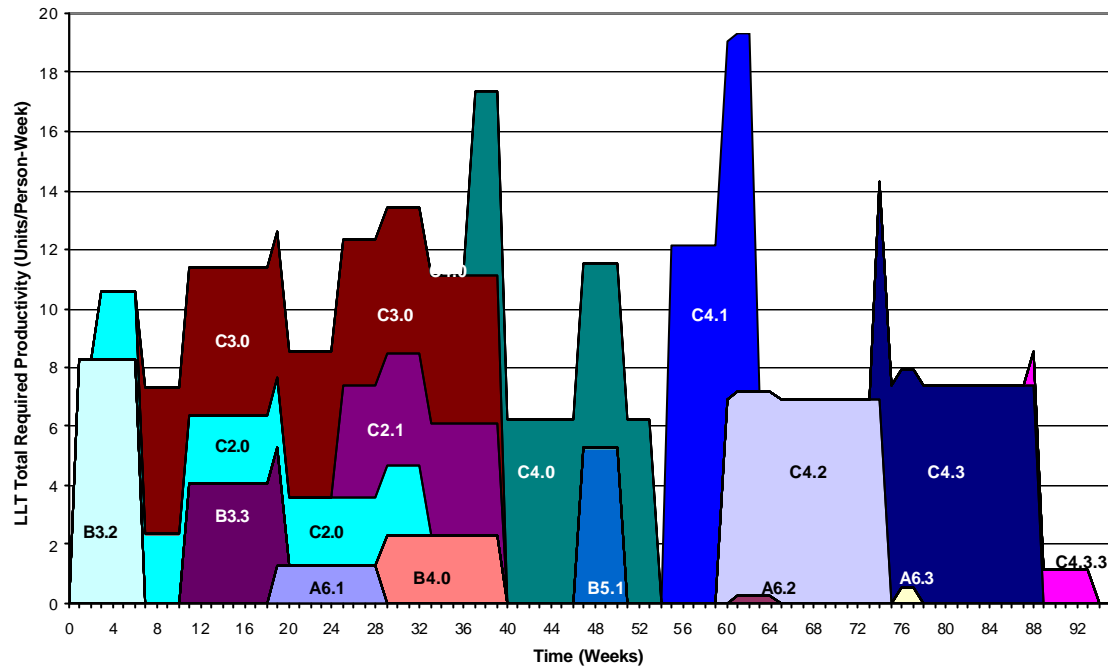


Figure 64: Model Analysis – Total Low Level Testing Phase *Required Productivity*

The peak in *Delivery C4.1* suggested that the overloading was caused by excessive pressure on the Low Level Testing phase and not on the Code phase as had been assumed. To confirm this, we re-examined our previous example of overloading shown in the delivery profile in Figure 40 (Section 5.2.1). On closer inspection, the overloading was a consequence of having insufficient time in which to perform low level testing of the code changes. In a follow-up discussion, the manager confirmed that the decision to split *Delivery C4.1* into *C4.1.1* and *C4.1.2* was a consequence of excessive pressure on low level testing. Faced with an over ambitious schedule, it had been infeasible to perform the bulk of low level testing on the full delivery in the time available (i.e. just four weeks).

The results were surprising but highly informative. In practice, scheduling of the required code changes had been the driver for the whole planning process, much to the detriment of Low Level Testing activities. This can be observed in the stability of total *Required Productivity* for code (Figure 63) compared with the more inconsistent demands on the low level testing team (Figure 64). The Low Level Testing phase traditionally had a bad reputation for causing schedule delays, but the model results actually showed that unreasonable planning constraints right at the outset of the project were to blame.

Whilst not conclusive, the results demonstrate the potential to predict the effects of planning decisions on parallel deliveries in the same phase (e.g. ensuring that coinciding deliveries don't cause overloading) and subsequent phases in the same delivery (e.g. balancing the schedule durations for coding and low level testing). The existence of overloading, indicated by a step-change in the demands on the process, shows what happens when we fail to get the balance right. If the managers had used our model to assess their planning constraints and assumptions, it is possible that this overloading could have been avoided.

### **7.2.5 Conclusions of the Evaluation**

We have used real planning data from Rolls-Royce in our primitive model to evaluate the Capacity-Based Scheduling approach. Two main tests were performed.

1. The *validity* of the modelling approach was tested by comparing the *Actual Effort* allocation made in practice, against the model's *Predicted Effort* (Section 7.2.3). The observed similarities between the predicted and actual patterns suggest we can build constraint-based models that, with some calibration, may be of sufficient accuracy for use in decision-making.
2. The *utility* of the modelling approach was tested by determining if the predicted level of *Required Productivity* could indicate the relative risk of a set of plans and thus give insights for planners (section 7.2.4). The results were sufficient to predict and explain the actual occurrence of overloading from a set of plans.

With further research, it seems possible that we could build more advanced models that are sufficiently reliable to help managers to rank alternative schedules according to their perceived risk. If the results are very close, the relative ranking may be of little importance. If, however, the results are an order of magnitude different then we can have a much higher degree of confidence in our ranking decisions.

### **7.2.6 Extensions to the Model**

We have introduced and evaluated the principles of Capacity Based Scheduling using a primitive model. A significant benefit of the CBS approach is that it provides a framework on which to build more realistic models of time-constrained development. In Appendix B, we present a table summarising some of the main assumptions behind the



model, the reasoning behind the assumptions, and example actions or extensions to address them. Whilst our modelling work has progressed beyond the models described in this thesis, our work to validate them is ongoing. Hence, these assumptions and proposed extensions must be seen as ‘future work.’ In the mean time, the following section describes how the CBS approach could be applied in practice.

## 7.3 Model Application

### 7.3.1 Overview

The results of our evaluation suggest that Capacity-Based Scheduling is a promising approach for the management of time-constrained software development. In this section, we bring together our strands of research by explaining how the CBS approach allows planners to evaluate different schedules in order to generate a plan that meets programme objectives within acceptable bounds of performance. The planning process consists of the following stages:

#### 7.3.1.1 Step 1 – High-Level Planning

In engine control systems development, critical milestones of the planned programme are identified by working backwards from the target date for the engine’s Entry-Into-Service to determine the dates that Certification, Flight-Testing, Engine Rig Testing, and First Engine Run, must commence. The high-level plan identifies the number of iterations necessary to meet the technical objectives, along with early predictions of the number of SANs in each delivery.

#### 7.3.1.2 Step 2 – Detailed Planning

A conventional planning tool, such as Microsoft Project, is then used to produce a set of low-level delivery and phase schedules. This detailed plan identifies (i) a *Planned Start Time* and *Planned End Time* for each phase delivery (and implicitly their level of concurrency), (ii) a *Planned Work* content in order to meet delivery objectives, and (iii) a *Planned Resource* for each phase team based on the assessment of current resource levels and future requirements.

Traditionally this was left to the judgement of the systems and software managers using past project plans and estimates of potential productivity from the lead-time model

(Section 6.2.2). The introduction of PEL, however, has provided managers with more information on which to base their planning decisions (Chapter 3).

#### 7.3.1.3 Step 3 – Model Plans

The model is then run with the inputs from the planned schedule, in conjunction with existing data for other planned and ongoing projects. The aim is to study two types of constraints: (i) *external constraints* that are imposed by the engine programme and need to be negotiated, and (ii) *internal constraints* that are imposed by the manager's planning decisions.

The model is then run to iteratively evaluate and refine (i) *productivity* (Section 7.3.1.4), (ii) *work* (Section 7.3.1.5), (iii) *resource* (Section 7.3.1.6), (iv) *effort* (Section 7.3.1.7), and the (iv) *schedule* (Section 7.3.1.8).

#### 7.3.1.4 Step 4 – Evaluate Productivity (*Required Productivity*)

The profile of *Required Productivity* is used to evaluate the relative risk of the plan. This is done using a stacked plot of required productivity (Section 7.2.4) or by using a control chart (Figure 65). The level of *Required Productivity* in Figure 65 is expressed as a deviation from the past *Actual Productivity* (measured using PEL). The control limits indicate the manager's acceptable bounds of productivity deviation (i.e.  $\pm 50\%$ ).

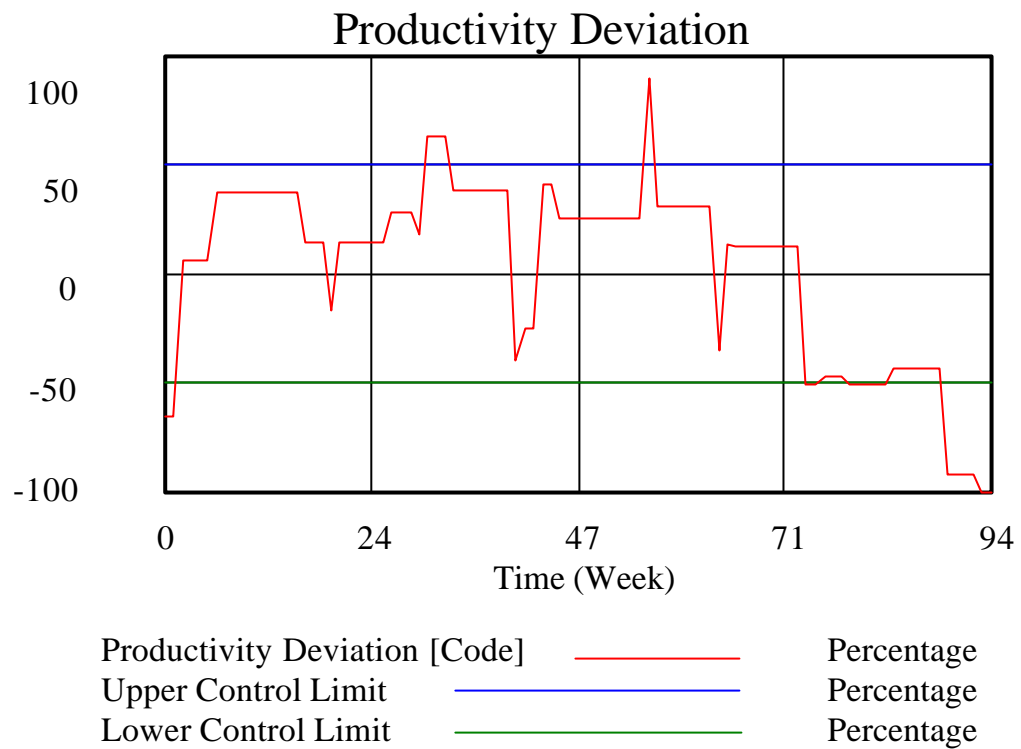


Figure 65: Model Application – Evaluation of Code *Required Productivity*

The *Required Productivity* for each development phase (*Code*, *Low Level Test* etc.) is then evaluated to see if the expected deviation is cause for concern. For example, in Figure 65, the required Code Team productivity has periods of high and low *Required Productivity* but these may not be sufficiently long for slack, overloading, or fatigue to be a problem. By contrast, the required Low Level Test Team productivity (Figure 66) shows the inconsistent schedule demands, and overloading, that we observed in our evaluation of CBS (Section 7.2.4).

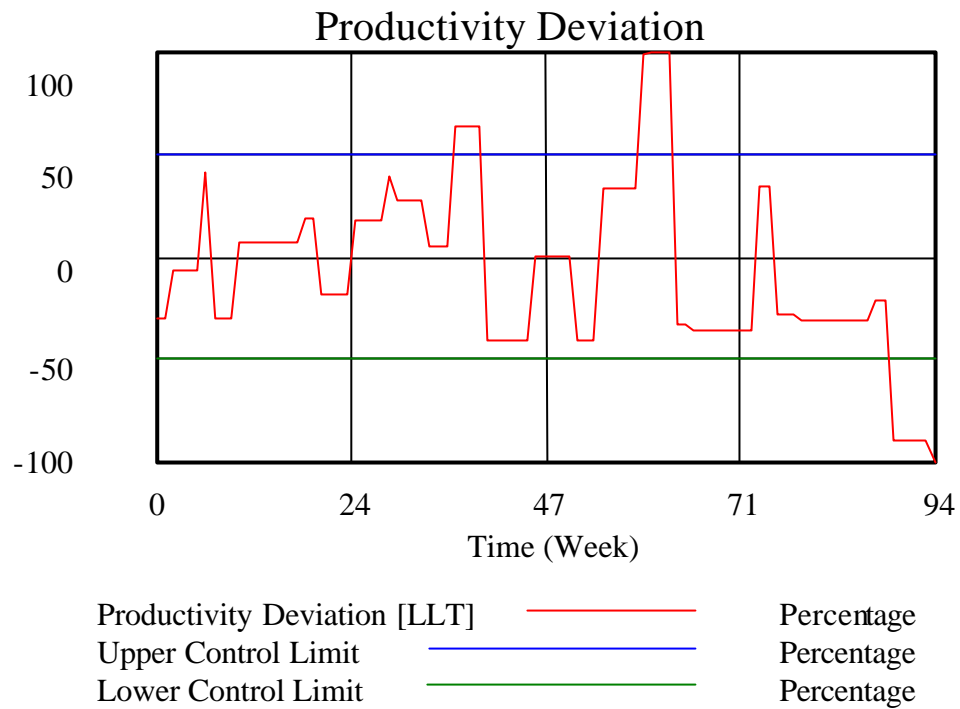


Figure 66: Model Application – Evaluation of LLT *Required Productivity*

If the deviation for any given phase is judged ‘too risky’ then the manager has a number of options: adjust the workloads, adjust the scheduling, adjust the resourcing, or accept the consequences of committing to this set of plans.

#### 7.3.1.5 Step 5 - Evaluate Work (*Planned & Achieved Work*)

The levels of *Achieved Work* are evaluated to ensure that upstream phases (e.g. *Code*) are sufficiently mature for related downstream phases (e.g. *Low Level Testing*) to begin. For example, Figure 67 shows the ‘completeness’ (*Achieved Work* as a percentage of *Planned Work*) for two phases (*Code*, *LLT*) of two deliveries (*C4.2*, *C.4.3*). The arrows show that *C4.2 Code* is 25% complete, and *C4.2 Code* is 30% complete, before their corresponding Low Level Testing phases are scheduled to begin.

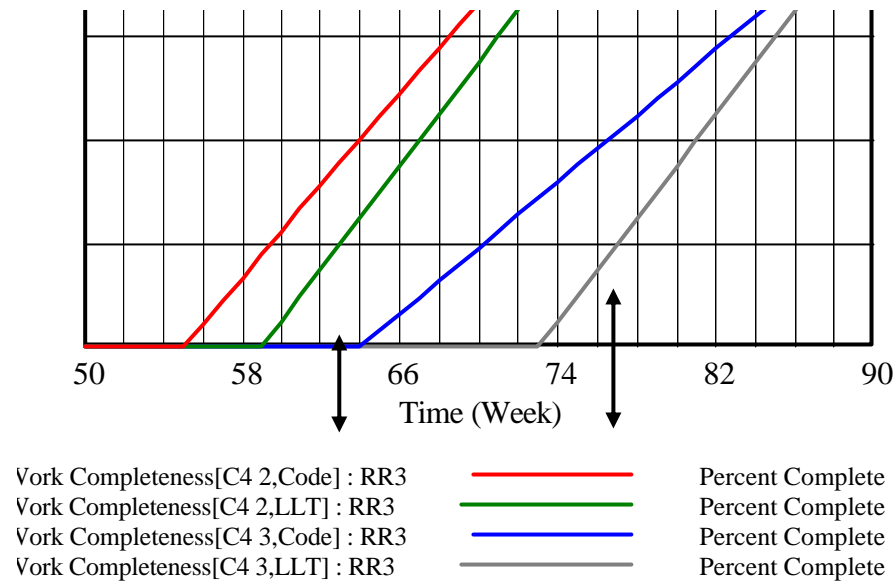


Figure 67: Model Application – Evaluation of Achieved Work

If the upstream phase is not sufficiently complete (e.g. 0 to 10%) then the level of concurrency is too high and the downstream phase could be delayed or, possibly worse, may receive unstable work-products. Equally, if the upstream phase is too complete (e.g. >50%) then the level of concurrency is too low and the project may be missing an opportunity to achieve further lead-time reductions.

#### 7.3.1.6 Step 6 - Evaluate Resources (*Planned & Applied Resource*)

The predicted profile of *Planned Resource* can be modelled using the STEP function of VenSim that enables the user to introduce a step increase or decrease the value of *Planned Resource* at any point in the simulation. For example, in Figure 68, the manager has increased the level of resource in the early stages of the plan, but decreased it as the *Planned Work* starts to tail off towards the planning horizon. The model therefore makes resourcing constraints and assumptions an explicit part of the planning process.

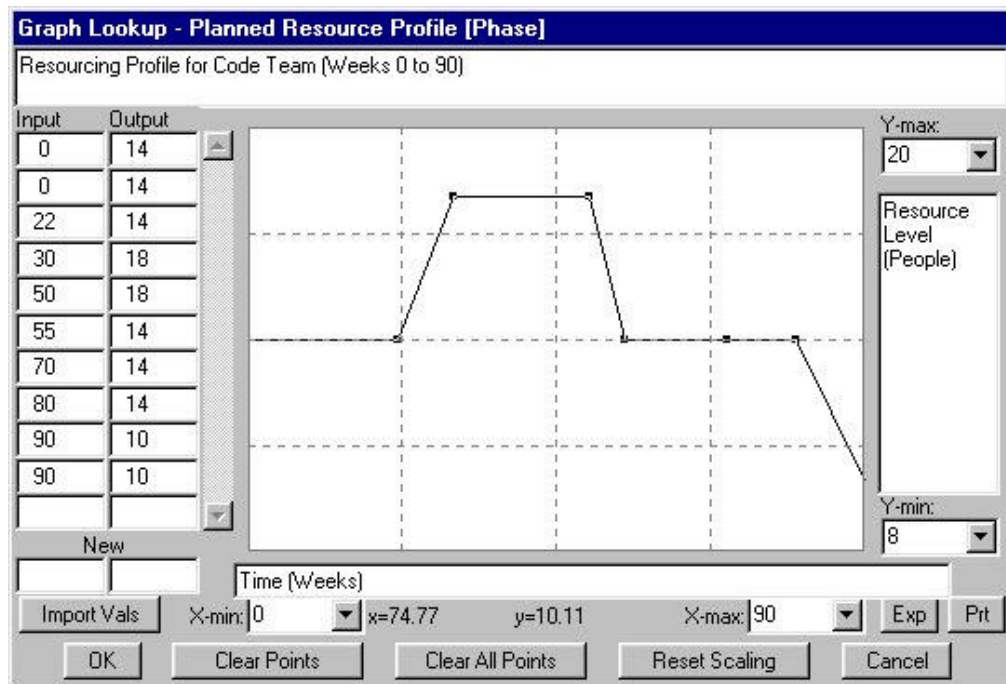


Figure 68: Model Application – Evaluation of *Planned Resource*

#### 7.3.1.7 Step 7 - Evaluate Effort (*Applied Effort*)

The overall effort and cost profile of the proposed plan must also be considered. For simplicity, the primitive model assumes a fixed weekly effort from the development team. In practice, overtime can be used to handle peaks in required productivity. Like for resource levels, the model can be extended to include a user-defined STEP profile of overtime to adjust the *Applied Effort* of the phase team. The level of *Achieved Cost* (= *Applied Effort* x *Planned Cost Rates*) can then be compared against budgets for acceptability.

#### 7.3.1.8 Evaluate Schedule (*Planned Start & End Time*)

The planning process is therefore a progressive and iterative refinement of these steps to produce a schedule that looks reasonable given past performance, current constraints, and future targets. It helps managers to balance risk across a portfolio of projects, and keeps plans within bounds of acceptable performance. If a manager does allow a deviation, they must be prepared to accept the consequences.

## 7.4 Summary of Managing Time-Constrained Development

In this chapter, we have evaluated the Capacity-Based Scheduling approach, and demonstrated how it can be applied to aid the management of time-constrained software development.

The software cost estimation field is coming to terms with the inherent limitations on the precision that can be achieved. As Kitchenham observes, “*Industry needs to come to terms with the fact that it cannot have the estimate accuracy it wants. Senior managers and project managers need to concentrate more on managing estimate risk than looking for a magic solution to the estimation problem*” (Kitchenham 1998). In response, we have proposed the Capacity-Based Scheduling approach to help managers to understand the risks inherent in their decisions.

It has a number of advantages over present management techniques.

- (i) *It makes planning assumptions explicit.* It encourages users to refine their understanding and decision-making, rather than following relying on the point estimates of cost models. The CBS approach therefore elicits previously implicit planning assumptions and renders explicit the expected consequences of a planning decision.
- (ii) *It helps investigate trade-offs between planning variables.* It prompts users to consider the consequences of alternative plans. For example, the impact of adding an extra coder, or balancing timescales for Code and Low Level Test phases.
- (iii) *It helps optimise plans according to some objective criteria.* We have suggested that different plans have different consequences. If managers do not have criteria for choosing between alternatives, then they are in trouble. CBS allows users to make judgements about alternative plans according to some goodness criteria (e.g. the deviation in *Required Productivity*).
- (iv) *It allows plans (assumptions) to be progressively refined over time.* It treats estimation and planning as a continuing rather than one-off activity.

Capacity-Based Scheduling therefore appears to be a promising approach for the management of time-constrained software development.

## Chapter 8: Conclusions

---

### 8.1 Thesis Contribution

In this thesis, we have investigated the proposition:

*“It is difficult to measure, model and manage time-constrained (iterative and concurrent) software development using conventional techniques. Instead, new fine-grained approaches to measurement and modelling are required in order to help plan, control and improve time-constrained software development”* (Section 1.4).

The reasons why conventional approaches are unsuitable in time-constrained situations are explored in chapters 2 to 5. In retrospect, their unsuitability is not surprising as they are failing in single-project waterfall lifecycles, let alone the highly iterative and concurrent lifecycles seen in industry. However, the tendency for published research to focus on the former has limited, if not hindered, the search for practical solutions for the latter (Parnas and Clements 1986). The potential costs of this oversight to both academia and industry imply that our definition of the problems is in itself a valuable contribution to knowledge.

In response, we have given foundations for fine-grained measurement and modelling approaches to aid the management of time-constrained software development. Our starting point was the introduction of a measurement technique, called the Process Engineering Language (PEL), to capture fine-grained information that was being lost by previous approaches to measurement (Chapter 3). The application of PEL was essential to understanding the specific nature of time-constrained projects within Rolls-Royce and how they differ from conventional projects (Chapter 4). Furthermore, it enabled us to isolate three specific problems that occur when managers fail to adequately balance the demands of concurrency and iteration: overloading, bow-waves, and multiplier effects (Chapter 5). This led us to propose a new modelling approach called Capacity-Based Scheduling (CBS) to help managers to avoid these problems and control risk across a portfolio of projects (Chapter 6). Finally, we evaluated the CBS approach before explaining how it can be used to improve the management of time-constrained software development (Chapter 7).



In Section 8.2, we examine in more detail the contribution made by each chapter towards the thesis proposition. After which, Section 8.3 presents areas for future research. Finally, in Section 8.4, we present our final conclusions.

## **8.2 Work Done, Claims and Evidence**

### ***8.2.1 Literature Review (Ch. 2)***

In Chapter 2, we gave a general overview of the research literature on the measurement, modelling, and management of software change. Whilst the specific nature and problems of time-constrained software development had not been studied in detail, we identified three authors whose work was highly influential in our thinking. First, was Lehman who argued that the (E-type) software process forms a dynamic feedback system and needs to be managed as such (Lehman 1998). Second, was Abdel-Hamid who used system dynamics to study software process feedback and illustrate the limitations of static models of software development (Abdel-Hamid and Madnick 1991). Third, was Kitchenham who argued that, since the software process is a complex non-linear adaptive system, there are inherent limits in the potential accuracy of our predictions. As a result, we need to adjust our methods and expectations accordingly (Kitchenham 1998). This research, and our observations of industrial practice, helped us to formulate a set of research questions to aid our understanding of time-constrained software development (Section 2.6). We concluded from this literature review that there was a lack of published literature and, hence, understanding of the nature and issues of time-constrained software development.

### ***8.2.2 Measuring Time-Constrained Software Development (Ch. 3)***

In Chapter 3, we identified specific problems and solutions in the measurement of time-constrained software development. First, we showed how conventional measurement frameworks, such as Work Breakdown Structures (WBS), fail to capture information needed to understand the detailed behaviour of software development processes. Instead, we introduce the Process Engineering Language (PEL) approach to process description and measurement, originally devised by Clark (Clark 1995). PEL exploits a taxonomy and simple language to collect measurements in a consistent and concise

manner that, unlike WBS, allows flexibility in the types of questions that can later be answered.

The specific contribution of the author was to extend and apply PEL to time-constrained projects within Rolls-Royce. We define a Goal Question Metric set for time-constrained development and our approach to the measurement of effort, defects and size. A cornerstone of this work was the introduction of three tools to collect PEL-compliant measures in a low-cost and unobtrusive manner: CoBS, RoCC, and the Comparator. Each tool used PEL to ensure consistency of description and connect our measures of effort, defects and size. We therefore showed by example how the resulting capability gave “*significant benefits to [...] identify and drive business improvements*” (Cann 1999). The introduction of a full-scale measurement programme was a significant undertaking but an essential basis for the research that followed.

### **8.2.3 Observations on Time-Constrained Software Development (Ch. 4)**

In Chapter 4, we used PEL to help understand the nature of time-constrained development, and how it differs from more conventional development described in the research literature. To do this, we used PEL effort data from Rolls-Royce to observe concurrency and iteration at the nested levels of project, delivery, phase, and task. In doing so, we could begin to explain the effects of time-compression on the process structure and behaviour. The Rolls-Royce process was driven by the extreme time-constraints of the engine development programme. This had led to the application of reuse, staged-delivery, concurrency and iteration, to levels far beyond those reported elsewhere in the literature. In particular, we observed how these techniques led to interdependencies between activities due to their reliance upon shared resources, processes, and work-products. The implication is that we need to rethink the techniques by which we manage these lifecycles. They must be managed as a portfolio rather than in isolation.

### **8.2.4 Problems of Time-Constrained Software Development (Ch. 5)**

In Chapter 5, we used our PEL effort, defect, and size measures to isolate three specific problems of time-constrained software development: overloading, bow-wave and the

multiplier effect (Section 5.2). Overloading occurs when the pressures on productivity are so high as to become unachievable. Bow-waves occur when a backlog or ‘ripple’ of changes move from delivery to delivery and result in increased pressure and risk towards the end of a project. Multiplier effects occur because changes introduced early in the process can have much greater knock-on effects later on. These three problems occur because of unreasonable internal and external planning constraints. In contrast, our comparison with BAE SYSTEMS reveals a very different set of constraints and resulting process (Section 5.3). It therefore seems that we need new models capable of showing if our constraints are reasonable and what strategies are appropriate in different situations.

### **8.2.5 Modelling Time-Constrained Software Development (Ch. 6)**

In Chapter 6, we identified the limitations of current predictive models and proposed a new approach to modelling time-constrained development, called *Capacity-Based Scheduling (CBS)*. The problem with conventional models is that they treat estimation and planning as separate activities. This contrasts with our observations of time-constrained projects where increased interdependencies between activities means that the way a project is planned and controlled can have a significant effect on its performance. We observed how, in the absence of existing models, Rolls-Royce formulated their ‘lead-time’ model that, whilst naïve, was of considerable practical benefit in judging the feasibility of delivery timescales.

In response, we therefore developed the Capacity-Based Scheduling (CBS) approach to model concurrency and iteration in a planned schedule. A primitive model was used to explain the principles behind the CBS approach. By comparing the levels of required productivity predicted by the model against the actual productivity measured using PEL, it is possible to evaluate the relative feasibility of a plan. The CBS approach therefore allows managers to make much more effective decisions as they can investigate the effect of their choices prior to committing to them.

### **8.2.6 Managing Time-Constrained Software Development (Ch. 7)**

In Chapter 7, we brought together the various strands of research by evaluating the validity and utility of the CBS approach and explained how it can be applied in practice.

The evaluation showed that it is possible to build CBS models of sufficient accuracy for use in decision-making and, even with our primitive model, we could predict and explain the actual occurrence of overloading from a set of plans. We ended by describing how CBS could be applied in practice and the benefits it can bring for the management of time-constrained software development.

### 8.3 Future Work

The work presented in this thesis therefore raises a number of strategic areas for future research. These include:

*Definition and refinement of the PEL approach.* The Process Engineering Language provides a significant improvement over present approaches for collecting valid software engineering data. However, more research is required to formally define and refine the method such that it can be adopted by, and validated between, different development environments (Clark and Powell 1999).

*Validation of the problems of time-constrained development.* The problem of managing software development under constrained timescales is a critical business issue for Rolls-Royce, but to what extent are these problems unique? Early evidence from projects within BAE SYSTEMS reveals the dual application of concurrency and staged-delivery in projects but with relatively more relaxed time constraints. More work is required to investigate the relationship between lead-time pressures and different strategies for software development. The joint application of PEL in Rolls-Royce and BAE SYSTEMS is providing a growing high-quality set of empirical data from which to investigate these problems (ibid.).

*Further development and validation of the CBS approach.* The principles of Capacity-Based Scheduling have been introduced using a primitive model. More work is required to build, calibrate and validate models that are suitable for application on live projects.

*Technology Transfer.* This research has been motivated by the practical problems of software engineering faced by industry. The introduction of the PEL measurement philosophy and toolset into industrial practice, and resulting improvements in process maturity (Nolan 1999), have been a major success of this research. The work to apply the techniques described in this thesis is ongoing.

*Wider Application.* Finally, whilst the focus of this research has been software development, the insights are not limited to this specific domain. The techniques introduced have wider significance for systems engineering and project management in many areas of endeavour.

## **8.4 Final Conclusions**

Considerable research work remains before we can fully understand the nature and problems of time-constrained software development. Even then, we are bound by both pragmatic and natural limits to the level of process understanding possible in these “*complex, non-linear, adaptive systems*” (Kitchenham 1998).

In the mean time, we have identified the problems and some solutions for the measurement, modelling, and management, of time-constrained software development. More pragmatically, we have developed and deployed methods and tools that have proved of value to managers responsible for developing software ‘*right on time.*’

# Glossary of Terms

---

Action	In <i>PEL</i> , the generative, evaluative and supportive verbs that describe what we do, e.g. <i>Produce</i> , <i>Review</i> (Section 3.3.2).
Actuals	Actual values are direct measures of the project.
Aircraft Interface	The physical connections (power, mechanical links, electrical signals, bus protocols), and control software, that connect the engine to the aircraft.
BAE SYSTEMS	A global systems, defence and aerospace company who act as a prime contractor and systems integrator in air, land, sea and space market sectors. This thesis includes data collected by BAE SYSTEMS' Fuel Computer team from the European Fighter Aircraft project.
Baseline	A <i>baseline</i> refers to the <i>release</i> of software from which subsequent changes have been made.
Behaviour	The way in which the elements or variables interact to stimulate changes over time.
Boomerang Effect	The effect, described by Weinberg, when a failure to get the product right first time will " <i>eventually come back and hit you</i> " (Weinberg 1992).
Bow-Wave	A ripple of changes (work and rework) that slip from delivery to delivery resulting in increased pressure and risk towards the end of a project.
BR700	A series of aeroengines developed jointly by Rolls-Royce and BMW to serve the mid-range market of business, cargo and passenger aircraft (up to 130 seats).
Capability Maturity Model (CMM)	A five-level model of a software development organisation's process maturity developed by the Software Engineering Institute.
Capacity-Based Scheduling (CBS)	A project scheduling technique, developed by the author, that starts with known constraints on the software process and reasons about the capabilities required to achieve them.
Certification	The process by which an engine product is shown to be flight-worthy.
Change Control Board (CCB)	A committee formed of senior engineers responsible for managing the change process including authorising, scheduling, and assuring, <i>SANs</i> .

Change Request	See: <i>Systems Anomaly Note (SAN)</i>
COCOMO	The COConstructive COst Model is a regression-based software cost estimation model, developed in 1981 by Barry Boehm, whilst at TRW (Boehm 1981).
Comparator Tool	A tool developed during the course of this research (as part of the ESSI PRIME project) to measure the size in <i>lines of code</i> , <i>files</i> and <i>functions</i> , of a configured version and the difference (delta) between two software <i>baselines</i> .
Concurrency	The simultaneous performance of development activities between projects, product deliveries, development phases and individual tasks. This exploits the benefits of performing activities in parallel against a risk that changes in an upstream activity may cause higher rework costs downstream.
Control	A means or device to regulate a process or sequence of events.
Control Laws	The result of applying Control Theory to a control problem; a software function to maintain control over an aeroengine.
Conventional Measures & Models	Traditional approaches to measurement and modelling that are predicated on linear and sequential lifecycles.
Conventional Software Lifecycle	The stages of software development performed in a sequential or ‘Waterfall’ manner with little or no overlap between them.
Cost Booking System (CoBS)	A tool developed during the course of this research to collect engineer cost (effort) bookings using the <i>Process Engineering Language (PEL)</i> approach to data collection.
Defect	An error or fault in a work-product.
Delivery	A release of a system or component to its customer or intended user). In the case of EEC development, the ‘customer’ is the wider engine development and testing programme.
Delivery Sequence Number (DSN)	A sequential numbering system for software deliveries in the form <i>Ab.c</i> where: <i>A</i> is the project reference (e.g. Project <i>A</i> ), <i>b</i> is the major delivery (e.g. 1 = <i>First Engine Run</i> , 2 = <i>Full-up Functionality</i> ), and <i>c</i> is the sub-delivery. For example, <i>D2.1</i> is the first full-up functionality delivery of the Douglas project.
Dynamic System	A system whose present output depends on past inputs.

Electronic Engine Controller (EEC)	The <i>Electronic Engine Controller (EEC)</i> is the microprocessor hardware and software that implements the real-time control functionality of a <i>Full Authority Digital Engine Controller (FADEC)</i> .
Embedded Software	Software that is physically incorporated into a larger system whose primary purpose is not data processing.
Empirical	Established by observation.
Engine Run	A delivery of control systems hardware and software containing the minimum of functionality needed to run an engine on a test rig. The <i>First Engine Run</i> delivery is made very early in the software lifecycle so that the long engine test programme can begin.
Entry Into Service (EIS)	The first delivery of control systems hardware and software that is used in service on an aircraft.
E-type System	<i>E-type</i> programs are required to solve a problem or implement an application in a real-world domain but they themselves change that domain. Correctness here is determined by the program's behaviour under operational conditions (Lehman 1996). See also: <i>P-type system</i> .
Evaluation	The process of determining whether an item or activity meets specified criteria.
Evolutionary Development	A lifecycle in which the system's complete functionality is enhanced in each successive delivery.
FADEC	The <i>Full Authority Digital Engine Controller</i> , or <i>FADEC</i> , is the system of hardware and embedded software responsible for the control of engine thrust, performance, monitoring and cockpit communication.
FEAST	The <i>Feedback Evolutionary And Software Technology (FEAST)</i> hypothesis suggests that the software process forms a dynamic feedback system and therefore needs to be managed as such (Lehman 1998).
Feedback	Transmission of information about the actual system performance to an earlier stage in order to modify its operation



Feedback (System)	Any actor in a system will eventually be affected by its own action (system dynamics). Negative feedback reduces the difference between actual and desired performance. Positive feedback induces instability by reinforcing a modification in performance (Senge 1990).
File	A <i>File</i> , measured by the <i>Comparator Tool</i> , is typically an Ada function, package body, package specification or module. Many <i>lines</i> comprise a file.
Function or Functional Area	A <i>Function</i> , or <i>Functional Area</i> , is a set of files that form a high-level abstraction of the system functionality and requirements. Many <i>files</i> comprise a function.
Goal Question Metric (GQM)	The Goal Question Metric (GQM) is a top-down goal-oriented approach to measurement developed by Basili and colleagues at the University of Maryland, USA (Basili 1988)
Grammar	The rules, inflections and syntax of a language. The basic PEL grammar combines the four dimensions of the lexicon – action, stage, product and representation – to form a statement of activity
Hierarchy of Concurrency	Four nested levels of concurrency within the software development (identified during this research). At the highest level is project concurrency where the software team works concurrently on software for different engine projects. Each project uses delivery concurrency to iterate through the software life-cycle and provide successive increments of software for engine testing. Within each delivery, phase concurrency is used to overlap the development phases, e.g. code and testing. Finally, each phase uses task concurrency by allocating tasks to be worked in parallel by different engineers.
Implementation Policy (IP)	An <i>Implementation Policy (IP)</i> schedules and controls the change requests ( <i>SANs</i> ) to be performed in a particular software delivery. The <i>RoCC</i> tool provides a convenient mechanism for scheduling <i>SANs</i> to <i>IPs</i> .
Incremental Development	A software development technique in which requirements definition, design, implementation and testing occur in an overlapping, iterative (rather than sequential) manner, resulting in incremental completion of the overall software product.
Integration Testing	Testing the emergent properties of a system element when combined with other elements, e.g. a single control component with other control components.

Iteration & Iterative Development	The repetition of development activities to deliver increments of product functionality at pre-planned intervals.
Lead-time	The time between the initiation of a product development and its market launch. Significant lead-time reduction in aeroengine development has been achieved by the simultaneous development of EEC hardware and software, in parallel with the development and testing of engines themselves.
Lexicon	The PEL lexicon is a constrained vocabulary of terms (verbs and nouns) used to describe a process. The basic lexicon is divided into four dimensions common to any organisation: <i>Actions, Stages, Products</i> and <i>Representations</i> .
Lifecycle	All the steps or phases an item passes through during its useful life.
LoC	<i>Line(s) of Code</i> . In the <i>Comparator Tool</i> , LoC is a physical line of source code, excluding comments, annotations and blank lines.
Low Level Testing	A testing process that exercises a procedure or module with data with the aim of ensuring that the code inside the unit implements its specification (McDermid 1991).
Measure	An empirical, objective assignment of a number (or symbol) to an entity (object or event) to characterise an attribute.
Measurement	The process of empirical, objective assignment of numbers (or symbols) to properties of entities (objects and events) in the real world in such a way as to describe them.
Metric	A quantitative measure of the degree to which a system, component or process possesses a given attribute.
Model	A representation of an artefact or activity intended to explain the behaviour or some aspects of it.
Multiplier effect	The multiplier effect occurs because change introduced in earlier (upstream) deliveries can have much greater knock-on effects on later (downstream) activities. If, for example, the process is not well planned or controlled, time pressure in one delivery can lead to significant amounts of rework in another. This in-turn can divert resources away from planned work and increase pressure on parallel and subsequent activities.

Overloading	Since resource levels are fixed and finite, a failure to consider the interacting demands of concurrency can result in resource bottlenecks. In certain situations, where the pressures on productivity are so high as to be unachievable, the process becomes ‘overloaded.’
Phase	A characteristic period in the sequence of events that comprise a lifecycle, e.g. Design, Coding, Low Level Testing.
Pivot Table	A feature of Microsoft Excel that uses interactive tables to summarise and manipulate large amounts of data.
Planning	The activities concerned with the specification of the components, timing, resources, and procedures of a project (ISO 2382-20).
Process	A ‘set of interrelated activities which transform inputs into outputs’ [ISO95].
Process Engineering Language (PEL)	A language-based approach to process description and measurement.
Product	The configurable physical or functional components or systems that are produced by a process. For example, <i>Aircraft, Engine Fuel Systems, Starting Systems</i> .
Product Representation	In <i>PEL</i> , the intermediate outputs or work-products of a process. For example, <i>Designs, Code, Plans</i> etc.
Productivity	The ratio of work done to effort expended.
P-type System	P-type programs are required to form an acceptable solution to a stated problem in the real world. Correctness of P-type programs is determined by the acceptability of the solution (Lehman 1996). See also: <i>E-type System</i> .
Quality	The degree to which a system, component, or process meets specified requirements.
Quality Assurance (QA)	A planned and systematic pattern of actions to provide adequate confidence that a given product conforms to established technical requirements.
Release	A defined set of many code modules or files (consisting of many functions) released for operation.
Representation	Product representations are the (intermediate) outputs or work-products of a process, for example: <i>Designs, Code</i> .

Requirements Testing	A verification process that generates acceptance tests, using software requirements, to determine whether a system meets user requirements.
Resource	The people, equipment and raw materials that are used to perform a process. They are usually limited in supply and therefore have an associated cost.
Resource Team/Pool	A team of people involved in the same phase of product development. In the case of safety critical systems, the need to maintain independence between development and testing activities, places restrictions on the movement of staff between lifecycle phases.
Reuse	The use of software components that already exist when building a new software product.
Rework	Action taken on a non-conforming product so that it will fulfil the specified requirements (ISO 8402). Internal and external
Rework Cycle	The sequence of rework and retesting activities performed in order to remove a defect.
Right-First-Time	The assumption of conventional life-cycles, typified by the Waterfall model, that imply a sequential once-through approach to development with cycle-times that are sufficiently long to proceed in a stepwise manner
Right-on-Time	The recognition that right-first-time may not be a cost effective strategy but that software artefacts should evolve over time in a controlled risk-driven manner; as typified by evolutionary models such as Boehm's Spiral model (Boehm 1988).
Ripple Effect	The impact of one change that can be seen to propagate through a system through a series of consequential changes that need to be made. See also: <i>Bow-Wave</i> .
Risk (schedule)	We define schedule risk as the deviation of a plan from actual capability.
Rolls-Royce	Rolls-Royce plc is a global company providing power generation systems for civil aerospace, defence, marine and energy markets. This thesis includes data collected by the BR700 Engine Controls team.

Rolls Change Control (RoCC)	A change control tool developed by Rolls-Royce to control the process by which SANs are authorised, raised, performed and closed. A key feature of RoCC is its ability to schedule changes to be performed ‘now or later’ by allocating SANs to <i>Implementation Policies (IPs)</i> .
Scheduling	The process of allocating scarce resources to activities over time.
Sensitivity Analysis	A procedure to determine the sensitivity of the outcomes of an alternative to changes in its parameters
Snap-Point	A snap-point occurs when managers realise that one or more constraints have to be relaxed to cope with excess demands placed on the process.
Software Freeze	A configured baseline of software to be released to downstream phase in the software lifecycle.
Software Lifecycle	The sequence of processes performed when developing and maintaining software.
Spiral Model	A model of the software development process in which the constituent activities, typically requirements analysis, preliminary detailed design, coding, integration and testing are performed iteratively until the software is complete.
Stage	In <i>PEL</i> , the time-based (milestone or cost) breakdown of logically sequenced partitions of work. For example, <i>Project A</i> or <i>Delivery D2</i> .
Staged Deliveries	The performance of several iterations of the software lifecycle to supply a tested increment of functionality at planned intervals.
Starting	The starting system (hardware and software) of an aeroengine.
S-type System	S-type programs are required to satisfy a pre-stated specification. Correctness is the absolute relationship between the specification and the program (Lehman 1996).
System Dynamics	A methodology used to understand how systems change over time (Forrester 1961).
Systems Anomaly Note (SAN)	The <i>Systems Anomaly Note (SAN)</i> is a generic problem report used by Rolls-Royce to report the existence of an anomaly in work-product(s) and control the consequent change activities.

Thrust Reverser	A software controlled device to slow an aircraft on landing by reversing the direction of an engine's thrust.
Time-Constrained Development	The development of systems and software where project deadlines are very tight and often non-negotiable (i.e. the costs of failing to meet deadlines are very high).
Top Level Design (TLD)	Top Level Design is lifecycle phase that uses software requirements to define the basic software structure and the environment in which the software operates. This includes defining the code structure, architecture, iteration rates, dataflow and data-hiding.
VenSim	The Ventana Simulation Environment is a tool that uses causal loop diagrams and continuous equations to build, and simulate, a system dynamics model (VenSim 1997).
Waterfall lifecycle	A simple model of software development that treats the process as a sequential set of stages dealing with requirements analysis, design, implementation, testing, and so on.
Work	A unit of output.
Work Breakdown Structure (WBS)	A product-oriented family tree composed of hardware, software, services, and other work tasks, which results from project engineering effort during the development and production of an item, and which completely defines the project or programme.
Work-products	The intermediate and final outputs of a process, for example: software requirements, designs, code, test specifications etc.

# Appendix A:

## Model Implementation

---

### A.1 Background

The Capacity-Based Scheduling model is implemented using the VenSim modelling tool (VenSim 1995). This was chosen for its ability to understand how structure affects behaviour by modelling systems using causal loop diagrams (Figure 69).

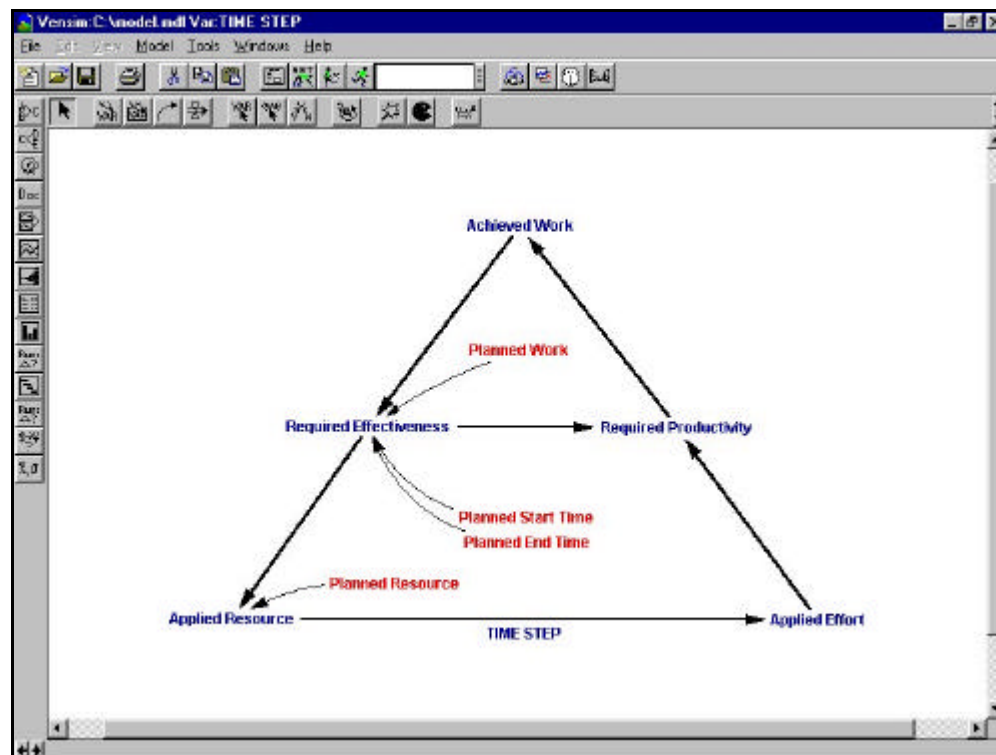


Figure 69: The VenSim Model

Each variable is represented by a mathematical equation expressed in the VenSim modelling language. For example, the equation for Applied Resource is shown in the top panel of the VenSim Equation Editor in Figure 70.

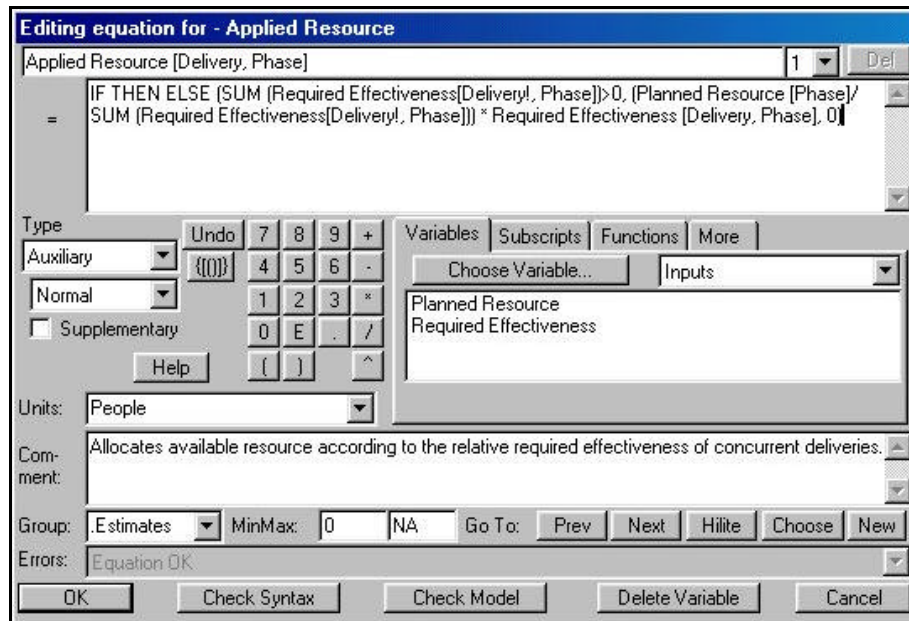


Figure 70: The VenSim Equation Editor

The model source code is provided in Section A.2.



## A.2 Model Source Code

```
*****
.Plans
*****
```

Planned Resource [Phase]= 0

```
~
~      Initialises the Planned Resource variable for user input.
~      People
|
```

Planned Start Time [Delivery, Phase]= 0

```
~
~      Initialises the Planned Start Time variable for user input.
~      Week
|
```

Planned End Time [Delivery, Phase]= 0

```
~
~      Initialises the Planned End Time variable for user input.
~      Week
|
```

Planned Work [Delivery, Phase] = 0

```
~
~      Initialises the Planned Work variable for user input.
~      DANs
|
```

```
*****
.Arrays
*****
```

~ Arrays are used to define the structure of the process being modelled

Projects: A, B, C

```
~
~      Declares the array of Projects to be simulated.
~      Array
|
```

Deliveries: A15, A16, A17, B6, B7, B8, B9, C11, C12, C15, C16, C2, C3, C4, C6, C8

```
~
~      Declares the array of Deliveries to be simulated.
~      Array
|
```

Phases: TLD, LLD, LLT

```
~
~      Declares the array of Phases to be simulated.
~      Array
```

\*\*\*\*\*  
 .Mappings  
 \*\*\*\*\*

Project Delivery Map [Project, Delivery] = 0

~  
 ~      *Project Delivery Map* describes the hierarchy of deliveries and parent  
 ~      projects, i.e. value is 1 if the delivery belongs to the project, otherwise 0.  
 ~      Binary flag.  
 |

Delivery Delivery Map [Delivery, Delivery] = 0

~  
 ~      *Delivery Delivery Map* describes the precedence of deliveries [A, B]  
 ~      i.e. if A is before B (B follows A) then 1, otherwise 0.  
 ~      Binary flag  
 |

Phase Phase Map [Phase, Phase] = 0

~  
 ~      *Phase Phase Map* describes the precedence of phases [A, B]  
 ~      i.e. if A is before B (B follows A) then 1, otherwise 0.  
 ~      Binary flag  
 |

\*\*\*\*\*  
 .Equivalences  
 \*\*\*\*\*

Project <-> Projects

~  
 ~      Mapping of *Project* to *Projects*.  
 ~      Equivalence  
 |

Delivery <-> Deliveries

~  
 ~      Mapping of *Delivery* to *Deliveries*.  
 ~      Equivalence  
 |

Phase <-> Phases

~  
 ~      Mapping of *Phase* to *Phases*.  
 ~      Equivalence  
 |

\*\*\*\*\*  
 .Calculations  
 \*\*\*\*\*

Achieved Productivity [Delivery, Phase] = INTEG (IF THEN ELSE(Achieved Work [Delivery, Phase]>0,( Achieved Effort [Delivery, Phase] / Achieved Work [Delivery, Phase]),0),0)

~  
~ Calculates the cumulative *Achieved Productivity* on the delivery phase  
~ from the *Achieved Effort* per *Achieved Work*.  
~ PersonWeeks/Unit  
~ :Supplementary  
|

Achieved Time [Delivery, Phase] = INTEG (Applied Time [Delivery, Phase], 0)

~  
~ *Achieved Time* records the total *Applied Time* on (i.e. duration of)  
~ each delivery phase.  
~ Weeks  
|

Achieved Work [Delivery, Phase] = INTEG (Required Productivity [Delivery, Phase], 0)

~  
~ *Achieved Work* is the cumulative *Required Productivity* over time.  
~ SANs  
|

Applied Effort [Delivery, Phase] = Applied Resource [Delivery, Phase] \* Applied Time [Delivery, Phase]

~  
~ The level of *Applied Effort* by the *Applied Resource* (people) over  
~ the *Applied Time* (Weeks).  
~ PersonWeeks  
|

Applied Project Effort [Project, Phase] = SUM (Applied Effort [Delivery!,Phase] \* Project Delivery Map[Project,Delivery!])

~  
~ Total *Applied Project Effort* for each project phase (e.g. A, Code),  
~ calculated from the total *Applied Effort* in each of its deliveries.  
~ :Supplementary  
|

Applied Resource [Delivery, Phase]= IF THEN ELSE (Required Phase Effectiveness [Phase]>0, (Planned Resource [Phase]/ Required Phase Effectiveness [Phase]) \* Required Effectiveness [Delivery, Phase], 0)

~  
~ Simulates a manager's allocation of the available *Planned Resource*  
~ for each phase (e.g. the Code team) between the parallel competing  
~ deliveries. The level of *Applied Resource* is proportional to the  
~ *Required Effectiveness* of each delivery.  
~ People  
|

Applied Time [Delivery, Phase] = Task Active[Delivery, Phase] \* TIME STEP

~  
~ The *Applied Time* increments one *TIME STEP* if *task active* = 1.  
~ Weeks  
|

Available Effort [Delivery, Phase] = Planned Effort [Delivery, Phase] – Applied Effort [Delivery, Phase]

- ~
- ~ Calculates the remaining *Available Effort* from the *Planned Effort* less the total *Applied Effort* (used). It is used as a reporting
- ~ variable to indicate the performance of the simulated plan against
- ~ the project planning constraints.
- ~ PersonWeeks
- ~ :Supplementary
- |

Available Time [Delivery, Phase] = Planned Time [Delivery, Phase] – Achieved Time [Delivery, Phase]

- ~
- ~ The remaining *Available Time* for each delivery phase, calculated
- ~ from the *Planned Time* less the *Achieved Time* (used).
- ~ Weeks
- |

Planned Time [Delivery, Phase] = Planned End Time[Delivery, Phase] – Planned Start Time [Delivery, Phase]

- ~
- ~ Calculates the duration or *Planned Time* for each delivery phase from
- ~ the *Planned Start Time* and *Planned End Time*.
- ~ Weeks
- |

Required Productivity [Delivery, Phase] = Required Effectiveness [Delivery, Phase] / Applied Effort [Delivery, Phase]

- ~
- ~ The *Required Productivity* is equivalent to the *Required Effectiveness*
- ~ since, for this model, we are interested in how much the required
- ~ productivity might have deviate from our known productivity to
- ~ meet the planned schedule.
- ~ SANs/Week
- |

Required Effectiveness [Delivery, Phase] = IF THEN ELSE (Task Active[Delivery, Phase] =1 :AND: Available Time [Delivery, Phase] >0, (Required Work [Delivery, Phase] / Available Time [Delivery, Phase]),0)

- ~
- ~ The average *Required Effectiveness* by each delivery phase to complete
- ~ the *Required Work* in the remaining *Available Time*.
- ~ SANs/Week
- |

Required Phase Effectiveness [Phase] = SUM (Required Effectiveness [Delivery!, Phase])

- ~
- ~ *Required Phase Effectiveness* (backlog) for each phase (e.g. code) team
- ~ calculated from the sum of the *Required Effectiveness* for each of its
- ~ deliveries.
- ~ SANs/Week
- |

Required Work [Delivery, Phase] = Planned Work [Delivery, Phase] – Achieved Work [Delivery, Phase]

- ~
- ~ *Required Work* (remaining) to complete the delivery, calculated by

~ deducting the *Planned Work* from the *Achieved Work*.  
 ~ SANs  
 |

Task Active [Delivery, Phase]= IF THEN ELSE(Time >= Planned Start Time [Delivery, Phase] :AND: Time <= Planned End Time [Delivery, Phase],1,0)

~  
 ~ Indicates the delivery phase is *Task Active* (i.e. the current *time* is  
 ~ between *Planned Start Time* and *Planned End Time*)  
 ~ Binary flag (1 active, 0 not active)  
 |

\*\*\*\*\*

.Control

\*\*\*\*\*

~ Simulation Control Parameters

FINAL TIME = 100

~  
 ~ The final time for the simulation.  
 ~ Weeks  
 |

INITIAL TIME = 0

~  
 ~ The initial time for the simulation.  
 ~ Weeks  
 |

SAVEPER = TIME STEP

~  
 ~ The frequency at which the model output is stored.  
 ~ Weeks  
 |

TIME STEP = 1

~  
 ~ The time step for the simulation.  
 ~ Weeks  
 |

(End of model code)

## **Appendix B:**

### **Model Assumptions and Extensions**

---

A basic list of model assumptions is provided in the table overleaf (Page 175), along with an example set of actions and/or extensions to the model.

<b>Variable</b>	<b>Assumptions</b>	<b>Reason</b>	<b>Example Actions / Extensions</b>
Resource	Resource levels are constrained (fixed) for each phase.	This is because resource is a relatively known planning variable in time-constrained projects.	Re-run the model to study the effects of adding/removing staff.
	Resources cannot move between lifecycle phases.	This models the Rolls-Royce process where engineers are specialists and independence has to be maintained between development and testing activities.	The model can be extended to allow resources to be manually, or automatically, reallocated.
	Resources allocation is 'perfect'	This is an initial simplification of the primitive model.	The model can be extended to allocate resources in chunks and after time-delays.
Effort	Effort is level with no overtime or lost-time (e.g. holidays)	This is an initial simplification of the primitive model.	The model can be extended to allow overtime either manually (as a STEP function) or as a function of, for example, productivity deviation.
Productivity	Deadlines are met (i.e. productivity is 100% of that which was required)	This is fundamental premise of the CBS approach in order to study the deviation of our plans from known capability.	The model can be re-run to perform sensitivity analysis on different schedules.
	Progress is made evenly from the current point until delivery.	This is an initial simplification of the primitive model.	Include dynamic behaviour (e.g. deadline effect) by defining relationships between, for example, required productivity and effectiveness.
	Past performance is a good guide of future performance.	The focus the CBS approach is to learn from past performance when making future decisions.	Allow users to develop and test their own CBS models.
Work	Work size reflects complexity and can be measured using a single property for each lifecycle phase.	This is an initial simplification of the primitive model.	The model can be extended to use multiple size and complexity measures.
Effectiveness	Products are completed (i.e. the process is 100% effective)	The primitive model does not simulate the 'bow-wave.'	The model can be extended to have dynamic effectiveness causing rework to move between deliveries.
	Model does not take into account deliveries with start times in the future	The model only deals with active tasks in the next time step.	The model can be extended to include anticipatory behaviour.

## References

- Abdel-Ghaly, A. A., P. Y. Chan, et al. (1986). "Evaluation of Competing Software Reliability Predictions." IEEE Transactions on Software Engineering **SE-12**(9): 950-967.
- Abdel-Hamid, T. and S. Madnick (1986). "Impact of Schedule Estimation on Software Project Behaviour." IEEE Software **3**(4): 70-75.
- Abdel-Hamid, T. K. (1993). "Adapting, Correcting, and Perfecting Software Estimates: A Maintenance Metaphor." Computer(March): 20-29.
- Abdel-Hamid, T. K. and S. E. Madnick (1991). Software Project Dynamics: An Integrated Approach. Englewood Cliffs, N.J., Prentice Hall.
- Ajila, S. (1995). "Software Maintenance: An Approach to Impact Analysis of Objects Change." Software - Practice and Experience **25**(10): 1155-1181.
- Albrecht, A. J. and J. E. Gaffney, Jr. (1983). "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." IEEE Transactions on Software Engineering **SE-9**(6): 639-48.
- Andersen, O. (1992). "Industrial Applications of Software Measurements." Information and Software Technology **34**(10): 681-693.
- Aoyama, M. (1993). "Concurrent Development Process Model." IEEE Software **10**(4): 46-55.
- Azuma, M. and D. Mole (1994). "Software Management Practice and Metrics in the European Community and Japan: Some Results of a Survey." Journal of Systems and Software **26**(1): 5-18.
- Baker, A. L., J. M. Bieman, et al. (1990). "A Philosophy for Software Measurement." Journal of Systems and Software **12**(3): 277-81.
- Bandinelli, S., A. Fuggetta, et al. (1995). "Modeling and Improving an Industrial Software Process." IEEE Transactions on Software Engineering **21**(5): 440-453.
- Bardill, M. (1996). Process Improvement Recommendations based on Analysis of the RoSEC Change Control Information. Derby, Rolls-Royce plc.
- Basili, V. R. (1990). "Viewing Maintenance as Reuse - Orientated Software Development." IEEE Software(January): 19-25.



- Basili, V. R. and J. D. Musa (1991). "The Future Engineering of Software: A Management Perspective." IEEE Computer **24**(9): 90-6.
- Basili, V. R. and H. D. Rombach (1988). "The TAME Project; Towards Improvement Oriented Software Environment." IEEE Transactions on Software Engineering **14**(6): 758-773.
- Basili, V. R. and D. Weiss (1984). "A Methodology for Collecting Valid Software Engineering Data." IEEE Transactions on Software Engineering **SE10**(6): 728-738.
- Bauer, F. L. (1971). Software Engineering. IFIP-Congress, Ljubljana, Yugoslavia, Springer-Verlag, Berlin.
- Baxter, I. D. (1992). "Design Maintenance Systems." Communications of the ACM **35**(4): 73-89.
- Bersoff, E. H. and A. M. Davis (1991). "Impacts of Lifecycle Models on Software Configuration Management." Communications of the ACM **34**(3): 105-118.
- Blackburn, J. D., G. Hoedemaker, et al. (1996). "Concurrent Software Engineering: Prospects and Pitfalls." IEEE Transactions on Engineering Management **43**(2): 179-188.
- Boehm, B. W. (1981). Software Engineering Economics. Englewood Cliffs, NJ, Prentice-Hall.
- Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement." IEEE Computer **21**(5): 61-72.
- Boehm, B. W. (1991). "Software Risk Management: Principles and Practices." IEEE Software January 1991: 32-41.
- Boehm, B. W., P. Bose, et al. (1995). Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach. Seventeenth International Conference on Software Engineering, IEEE.
- Boulding, K. E. (1956). "General systems theory - the skeleton of science." Management Science **2**(3).
- Briand, L. C., V. R. Basili, et al. (1992). "A Pattern-Recognition Approach for Software Engineering Data-Analysis." IEEE Transactions on Software Engineering **18**(11): 931-942.

- Briand, L. C., S. Morasca, et al. (1996). "Property - Based Software Engineering Measurement." IEEE Transactions on Software Engineering **22**(1): 68-85.
- Brocklehurst, S. and B. Littlewood (1992). "New Ways to get Accurate Software Reliability Modelling." IEEE Software(July).
- Brooks, F. P. (1975). The Mythical Man-Month. Reading, MA, Addison-Wesley.
- Brown, D. S. (1999). PEL Evaluation. Derby, Rolls-Royce BR700 Control Systems.
- Burke, M. (1999). PEL Evaluation. York, BAE SYSTEMS Dependable Computing Systems Centre.
- Cann, R. G. (1999). Booking into SAP and the Use of PEL. Derby, Rolls-Royce plc.
- Checkland, P. B. (1981). Systems Thinking, Systems Practice, Wiley.
- Chillarege, R., I. S. Bhandari, et al. (1992). "Orthogonal Defect Classification - A Concept for In-Process Measurements." IEEE Transactions on Software Engineering **18**(11): 943-956.
- Clark, G. D. (1997). Process Engineering Language - Principles and Application.
- Clark, G. D. and A. L. Powell (1999). Experiences with Sharing a Common Measurement Philosophy. International Conference on Systems Engineering (INCOSE'99), Brighton, UK.
- Collier, K. W. and J. Collofello (1995). Issues in Software Cycle Time Reduction. Proceedings of the Phoenix Computers and Communication Conference.
- Conklin, P. F. (1992). "Enrollment Management: Management the Alpha AXP Program." Digital Technical Journal **4**(4): 193-205.
- Costello, R. J. and D. B. Liu (1995). "Metrics for Requirements Engineering." Journal of Systems and Software **29**(1): 39-63.
- Curtis, B., M. I. Kellner, et al. (1992). "Process Modeling." Communications of the ACM **35**(9): 75-90.
- Curtis, B., H. Krasner, et al. (1987). On Building Software Models under the Lamppost. 9th International Conference on Software Engineering, Washington DC, IEEE Computer Society.

- Dalal, S. R. and A. A. McIntosh (1994). "When to Stop Testing for Large Software Systems with Changing Code." IEEE Transactions on Software Engineering **20**(4): 318-323.
- Daskalantonakis, M. K. (1992). "A Practical View of Software Measurement and Implementation Experiences within Motorola." IEEE Transactions on Software Engineering **18**(11): 998-1009.
- DeMarco, T. (1982). Controlling Software Projects - Management, Measurement and Estimation. New York, Yourdon Press.
- Deming, W. E. (1986). Out of the Crisis. Cambridge, MA, MIT Center for Advanced Engineering Study.
- Dion, R. (1993). "Process Improvement and the Corporate Balance-Sheet." IEEE Software **10**(4): 28-35.
- Duncan, A. S. (1988). Software Development Productivity - Tools and Metrics. Tenth International Conference on Software Engineering, Singapore.
- Evanco, W. M. (1995). "Modelling the Effort to Correct Faults." Journal of Systems and Software **29**: 75-84.
- Evanco, W. M. and R. Lacovara (1994). "A Model-Based Framework for the Integration of Software Metrics." Journal of Systems and Software **26**(1): 77-86.
- Fenton, N. E. (1991). Software Metrics - A Rigorous Approach. London, Chapman & Hall.
- Fenton, N. E. (1994). "Software Measurement - A Necessary Scientific Basis." IEEE Transactions on Software Engineering **20**(3): 199-206.
- Fernström, C., K. H. Narfelt, et al. (1992). "Software Factory Principles, Architecture, and Experiments." IEEE Software(March): 36-44.
- Forrester, J. W. (1961). Industrial Dynamics. Cambridge, MA, MIT Press.
- Genuchten, M. and H. Koolen (1991). "On the use of Software Cost Models." Information & Management **21**: 37-44.
- Gilb, T. (1985). "Evolutionary Delivery versus the Waterfall Model." ACM Sigsoft(July): 49-62.

- Gilb, T. (1996). "Level 6: Why We Can't Get There from Here." IEEE Software January: 97-103.
- Gilb, T. (1998). "Inspection and 'Spiral' (Evolutionary Delivery)." Software Engineering Notes **23**(1).
- Gleick, J. (1988). Chaos: Making a New Science. London, Heinemann.
- Grady, R. B. (1990). "Work-Product Analysis: The Philosopher's Stone of Software?" IEEE Software(March): 27-34.
- Grady, R. B. and D. Caswell (1987). Software Metrics: Establishing a Company-Wide Program. Englewood Cliffs, NJ, Prentice-Hall.
- Griss, M. L. (1993). "Software Reuse: From Library to Factory." IBM Systems Journal **32**(4): 548-565.
- Gulla, B., E. A. Karlsson, et al. (1991). "Change-Orientated Version Descriptions in EPOS." Software Engineering Journal(November): 378-397.
- Halstead, M. H. (1975). Elements of Software Science. N-Holland, Elsevier.
- Hamer, P. G. and G. D. Frewin (1982). Halstead's Software Science - A Critical Examination. 6th International Conference on Software Engineering.
- Haney, F. M. (1972). Module Connection Analysis - A Tool for Scheduling Software Debugging Activities. AFIPS Fall Joint Computer Conference.
- Hansen, G. A. (1996). "Simulating Software Development Processes." IEEE Computer January: 73-77.
- Harper, K. G. (1995). BR700 Series EEC Systems and Software Development Plan. Derby, Rolls Smiths Engine Controls (RoSEC).
- Harrison, W. (1997). Proposal: Creating the Center for Software Process Improvement and Modeling, [http://www.cs.pdx.edu/~warren/SIM\\_GROUP/proposal.html](http://www.cs.pdx.edu/~warren/SIM_GROUP/proposal.html).
- Hayes, N. (1998). Cost Booking System User Guide. Derby, UK, Rolls-Royce plc.
- Heemstra, F. J. (1992). "Software Cost Estimation." Information and Software Technology **34**(10): 627-639.
- Henry, J., R. Blasewitz, et al. (1996). "Defining and Implementing a Measurement-Based Software Maintenance Process." Journal of Software Maintenance: Research and Practice **8**: 79-100.

- Henry, S. and D. Kafura (1981). "Software Structure Metrics based on Information-Flow." IEEE Transactions on Software Engineering **7**(5): 510-518.
- Hitchins, D. K. (1992). Putting Systems to Work, Wiley.
- Humphrey, W. S. (1991). "Software and the Factory Paradigm." Software Engineering Journal September: 370-376.
- Imai, M. (1986). Kaizen - The Key to Japan's Competitive Success, McGraw-Hill.
- Ince, D. C. and M. J. Sheppard (1988). System Design Metrics: A Review and Perspective. Proceedings of the IEE/BCS Conference on Software Engineering (SE'88).
- ISO (1995). Information Technology - Software Lifecycle Processes, International Standards Organisation and International Electro-technical Committee.
- ISO/IEC (1996). ISO/IEC 15504: Software Process Improvement and Capability dEtermination (SPICE), International Committee on Software Engineering Standards, International Standards Organisation.
- Jaffe, M. S., N. G. Leveson, et al. (1991). "Software Requirements Analysis for Real-Time Process Control Systems." IEEE Transactions on Software Engineering(March).
- Jørgensen, M. (1995). "An Empirical Study of Software Maintenance Tasks." Journal of Software Maintenance-Research and Practice **7**(1): 27-48.
- Kafura, D. and J. T. Canning (1985). A Validation of Software Metrics using Many Metrics and Two Resources. 8th International Conference on Software Engineering, London.
- Kanoun, K. and J. C. Laprie (1994). "Software Reliability Trend Analyses from Theoretical to Practice Considerations." IEEE Transactions on Software Engineering **20**(9 (September)): 740-747.
- Kemerer, C. F. (1987). "An Empirical Validation of Software Cost Estimation Models." Communications of the ACM **30**(5): 416-429.
- Khoshgoftaar, T. M. and D. L. Lanning (1995). "A Neural-Network Approach for Early Detection of Program Modules having High-Risk in the Maintenance Phase." Journal of Systems and Software **29**(1): 85-91.

- Khoshgoftaar, T. M., R. M. Szabo, et al. (1995). "Exploring the Behaviour of Neural Network Software Quality Models." Software Engineering Journal **10**(3): 89-96.
- Khoshgoftaar, T. M., E. B. Allen, et al. (1996). "Early Quality Prediction: A Case Study in Telecommunications." IEEE Software(January): 65-71.
- Kitchenham, B. A. (1998). The Certainty of Uncertainty. European Software Measurement Conference FEMSA 98, Antwerp, Technologisch Institute.
- Kitchenham, B. A. and S. Linkman (1997). "Estimates, Uncertainty, and Risk." IEEE Software(May/June): 69-74.
- Kitchenham, B. A., S. G. Linkman, et al. (1994). "Critical Review of Quantitative Assessment." Software Engineering Journal March: 43-53.
- Kitchenham, B. A. and S. J. Linkman (1990). "Design Metrics in Practice." Information and Software Technology **32**(4): 304-310.
- Kitchenham, B. A., L. M. Pickard, et al. (1990). "An Evaluation of Some Design Metrics." Software Engineering Journal **5**(1): 50-8.
- Krasner, H., J. Terrel, et al. (1992). "Lessons Learned from a Software Process Modeling System." Communications of the ACM **35**(9): 91-100.
- Lanubile, F. and G. Visaggio (1995). "Decision-Driven Maintenance." Journal Of Software Maintenance-Research and Practice **7**(2): 91-115.
- Lederer, A. L. and J. Prasad (1995). "Causes of Inaccurate Software-Development Cost Estimates." Journal of Systems and Software **31**(2): 125-134.
- Lehman, M. M. (1976). "A Model of Large Program Development." IBM Systems Journal **15**(3).
- Lehman, M. M. (1978). Why Software Projects Fail. Infotech State of the Art Conference, Pergamon Press.
- Lehman, M. M. (1995). Process Improvement - The Way Forward. 7th International Conference on Advanced Information Systems Engineering (CAiSE'95), Jyaskyla, Finland.
- Lehman, M. M. (1996). "Feedback in the Software Evolution Process." Information and Software Technology - Special Issue on Software Maintenance: 681-686.

- Lehman, M. M. (1998). "Software's Future: Managing Evolution." IEEE Software(January/February): 40-44.
- Lehman, M. M., D. E. Perry, et al. (1996). Why is it so Hard to find Feedback Control in Software Processes? Nineteenth Australasian Computer Science Conference, Melbourne, Australia.
- Lind, R. K. and K. Vairavan (1989). "An Experimental Investigation of Software Metrics and their Relationship to Software Development Effort." IEEE Transactions on Software Engineering **15**(5): 649-53.
- Linkman, S. G. (1990). "Quantitative Monitoring of Software Development by Time-Based and Intercheckpoint Monitoring." Software Engineering Journal **5**(1): 43-9.
- Logothetis, N. (1992). Managing for Total Quality: From Deming to Taguchi and SPC. New York, Prentice Hall.
- Madhavji, N. H. (1991). "The Process Cycle." Software Engineering Journal(September): 234-242.
- Mathews, A. I. (1997). RoSEC Change Control System (RoCC) User Guide. Derby, Rolls Smiths Engine Controls (RoSEC).
- Mays, R. G., C. L. Jones, et al. (1990). "Experiences with Defect Prevention." IBM Systems Journal **29**(1): 4-32.
- McCabe, T. J. (1976). "A Complexity Measure." IEEE Transactions on Software Engineering **5**(December): 308-320.
- McChesney, I. R. (1995). "Toward a Classification Scheme for Software Process Modeling Approaches." Information and Software Technology **37**(7): 363-374.
- McDermid, J. A. (1991). Software Engineer's Reference Book. Oxford, Butterworth-Heinemann.
- McDermid, J. A. (1997). Measurement and Assurance. Lifecycle Management for Dependability. F. Redmill and C. Dale, Springer-Verlag.
- McDermid, J. A. and P. Rook (1991). Software Development Process Models. Software Engineer's Reference Book. J. A. McDermid, Butterworth-Heinemann: 16/1-16/21.

- Mili, H., F. Mili, et al. (1995). "Reusing Software: Issues and Research Directions." IEEE Transactions on Software Engineering **21**(6 (June)): 528-561.
- Mukhopadhyay, T. and S. Kekre (1992). "Software Effort Models for Early Estimation of Process-Control Applications." IEEE Transactions on Software Engineering **18**(10): 915-924.
- Munson, J. C. and T. M. Khoshgoftaar (1990). "Applications of a Relative Complexity Metric for Software Project Management." Journal of Systems and Software **12**(3): 283-291.
- Munson, J. C. and T. M. Khoshgoftaar (1992). "The Detection of Fault-Prone Programs." IEEE Transactions on Software Engineering **18**(5): 423-33.
- Musa, J. (1975). "A Theory of Software Reliability and its Application." IEEE Transactions on Software Engineering **SE-1**: 312-327.
- Musa, J. D., A. Lannino, et al. (1987). Software Reliability, Measurement, Prediction and Application, McGraw-Hill.
- Myers, G. J. (1979). The Art of Software Testing. New York, Wiley.
- Naur, P. and B. Randell (1969). Software Engineering. Report on a Conference Sponsored by the NATO Science Committee, Garmish, Germany.
- Nikora, A. P. and M. R. Lyu (1995). An Experiment in Determining Software Reliability Model Applicability. Sixth International Symposium on Software Reliability Engineering, Toulouse, France.
- Nolan, A. J. (1999). Assessment of BR700 FADEC Controls' Verification and Validation Process, Rolls-Royce Control Systems.
- Nolan, A. J. (1999). Metrics Collection and Assessment for the BR700 FADEC Control Department. Derby, Rolls-Royce BR700 FADEC Controls.
- Oivo, M. and V. R. Basili (1992). "Representing Software Engineering Models: The TAME Goal Oriented Approach." IEEE Transactions on Software Engineering **18**(10): 886-98.
- Olsen, N. C. (1993). "The Software Rush Hour." IEEE Software(September): 29-37.
- Palmer, J. D. and R. P. Evans (1994). Software Risk Management: Requirements-Based Risk Metrics. IEEE International Conference on Systems, Man and Cybernetics, Information & Software Technology.



- Papapanagiotakis, G. and P. Breuer (1994). "A Software Maintenance Management Model-Based on Queuing-Networks." Journal of Software Maintenance-Research and Practice **6**(2): 73-97.
- Parets, J. and J. C. Torres (1996). Software Maintenance Versus Software Evolution: An Approach to Software Systems Evolution. IEEE Symposium and Workshop on the Engineering of Computer-Based Systems (ECBS), Friedrichshafen, Germany.
- Parkinson, J. (1996). 60 Minute Software: Strategies for Accelerating the Information Systems Delivery Process. New York, John Wiley & Sons.
- Parnas, D. L. and P. C. Clements (1986). "A Rational Design Process: How and Why to Fake It." IEEE Transactions on Software Engineering **SE12**(2): 251-257.
- Pattee, H. H. (1973). Hierarchy Theory, George Braziller.
- Pengelly, A. (1995). "Performance of Effort Estimating Techniques in Current Development Environments." Software Engineering Journal(September): 162-170.
- Pfleeger, S. L. and C. McGowan (1990). "Software Metrics in the Process Maturity Framework." Journal of Systems and Software **12**(3): 255-261.
- Porter, A. A. and R. W. Selby (1990). "Empirically Guided Software Development using Metric-Based Classification Trees." IEEE Software **7**(2): 46-54.
- Powell, A. L. (1995). Analysis of PCMS Error Trend Metrics. Derby, Rolls-Royce Industrial Power Group.
- Powell, A. L. (1995). Implementation of PCMS Error Trend Metrics. Derby, Rolls-Royce Industrial Power Group.
- Powell, A. L. and G. D. Clark (1999). Learning from Simulation? Mind your Language. Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany.
- Powell, A. L., K. C. Mander, et al. (1999). "Strategies for Lifecycle Concurrency and Iteration - A System Dynamics Approach." The Journal of Systems & Software **46**(2-3): 151-161.

- Putnam, L. H. (1978). "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." IEEE Transactions on Software Engineering **4**(4): 345-361.
- Reibman, A. L. (1991). "Reliability Modeling: An Overview for System Designers." IEEE Computer(April): 49-57.
- Rombach, H. D., B. T. Ulery, et al. (1992). "Toward Full Lifecycle Control - Adding Maintenance Measurement to the SEL." Journal of Systems and Software **18**(2): 125-138.
- Rombach, H. D. and M. Verlage (1995). Directions in Software Process Research, Academic Press.
- Royce, W. W. (1970). Managing the Development of Large Systems: Concepts and Techniques. IEEE WESCON, IEEE Press.
- Schneidewind, N. F. (1992). "Methodology for Validating Software Metrics." IEEE Transactions on Software Engineering **18**(5): 410-422.
- Shen, V. Y., T. J. Yu, et al. (1985). "Identifying Error-Prone Software - An Empirical Study." IEEE Transactions on Software Engineering **SE-11**(4): 317-25.
- Shepperd, M. (1992). "Products, Processes and Metrics." Information and Software Technology **34**(10): 674-680.
- Shepperd, M. and D. C. Ince (1994). "A Critique of Three Metrics." Journal of Systems and Software **26**(3): 197-210.
- Sims, D. (1997). "Vendors Struggle with Costs, Benefits of Shrinking Cycle Times." Computer(September): 12-14.
- Strens, M. R. and R. C. Sugden (1996). Change Analysis: A Step Towards Meeting the Challenge of Changing Requirements. IEEE Symposium and Workshop on the Engineering of Computer-Based Systems (ECBS), Friedrichshafen, Germany.
- Subramanian, G. H. and S. Breslawski (1995). "An Empirical Analysis of Software Effort Estimate Alterations." Journal of Systems and Software **31**(2): 135-141.
- Sugden, R. C. and M. R. Strens (1996). Strategies, Tactics and Methods for Handling Change. IEEE Symposium and Workshop on the Engineering of Computer-Based Systems (ECBS), Friedrichshafen, Germany.
- Vensim (1997). Vensim 3.0. Harvard, MA, Ventana Systems Inc.

- Verner, J. and G. Tate (1992). "A Software Size Model." IEEE Transactions on Software Engineering **18**(4): 265-78.
- Weinberg, G. M. (1992). Systems Thinking. New York, Dorset House.
- Weller, E. F. (1994). "Using Metrics to Manage Software Projects." Computer **27**(9): 27-33.
- Wild, R. (1989). Production and Operations Management. London, Cassell.
- Williams, J. D. (1996). "Managing Iteration in OO Projects." IEEE Computer(September): 39-43.
- Withrow, C. (1990). "Error Density and Size in Ada Software." IEEE Software **7**(1): 26-30.
- Wohlin, C. and U. Körner (1990). "Software Faults: Spreading Detection and Costs." Software Engineering Journal(January).
- Wright, D. (1996). Problem Reporting and Change Control Procedures for BR700 Projects. Derby, Rolls Smiths Engine Controls (RoSEC).
- Yau, S. S. (1985). "Design Stability Measures for Software Maintenance." IEEE Transactions on Software Engineering **11**(9): 849-856.
- Yau, S. S. and J. S. Collofello (1980). "Some Stability Measures for Software Maintenance." IEEE Transactions on Software Engineering **6**(6): 545-552.
- Yu, T. J., V. Y. Shen, et al. (1988). "An Analysis of Several Software Defect Models." IEEE Transactions on Software Engineering **14**: 1261-1270.