

Modelling Agile Software Processes Using Bayesian Networks

by

Peter Stewart Hearty

Submitted for the degree of Doctor of Philosophy
Queen Mary, University of London
2008

I certify that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

I acknowledge the helpful guidance and support of my supervisor, Professor Norman Fenton, and the additional guidance of Professor Martin Neil. This work was made possible through funding from the EPSRC ExDecide project. I also wish to gratefully acknowledge the generous student grant provided by Agena Ltd.

24 Sep 2008

Peter Hearty

Date

Abstract

In a fast moving world, business requirements often change throughout the lifetime of a project. Many organisations now recognise that software specifications can no longer be fixed before development begins: software teams must “embrace change” [18].

Competitive pressure demands rapid software development that can track these changing requirements. “Agile” development methods attempt to address this by adopting an iterative approach to software development. Each iteration delivers more functionality but also responds to constant customer feedback. Functionality is modified where necessary to ensure that the product continues to meet customers’ changing needs.

Agile environments eschew the formalisms of traditional specification and design. They favour rapid application development techniques which lead to quicker working prototypes of the final product. The downside of this is a lack of specification metrics with which to plan the project. Yet managers of Agile projects have just as great a need to cost and plan their projects as any traditional project manager.

This thesis shows how the iterative nature of Agile methodologies can be leveraged to create a learning model of the software development process. This is illustrated with a Dynamic Bayesian Net model of Extreme Programming’s key Project Velocity metric. The model is validated against a real Extreme Programming project where it demonstrates good predicative power.

To Terry, who's put up with me for the last 25 years.

Glossary

API	Application Programming Interface
BN	Bayesian Network
CASE	Computer Assisted Software Engineering
CI	Conditionally Independent
CPD	Conditional Probability Distribution
CPT	Conditional Probability Table
DAG	Directed Acyclic Graph
DBN	Dynamic Bayesian Net
DD	Dynamic Discretisation
DEMC	Data Extraction, Mapping And Cleansing
FPS	Function Points
HMM	Hidden Markov Model
IDE	Integrated Development Environment
IED	Ideal Engineering Day
JPD	Joint Probability Distribution
JTA	Junction Tree Algorithm
KFM	Kalman Filter Model
KDSI	Thousand (Kilo) Delivered Source Instructions
KLOC	Thousand (Kilo) Lines Of Code
LOC	Line Of Code
NPT	Node Probability Table
OO	Object Oriented
PP	Pair Programming
TDD	Test Driven Development
UML	Unified Modelling Language
XP	eXtreme Programming

Table of Contents

1	INTRODUCTION	14
	1.1 Hypothesis	15
	1.2 Structure of this Thesis	15
	1.3 Published Papers	16
2	SOFTWARE PROCESS MODELS	18
	2.1 Aims of Software Process Models	18
	2.2 Regression models	18
	2.3 Function Points	20
	2.4 Including Complexity	23
	2.5 Multivariate Models	25
	2.6 General Problems with Parametric Models	26
	2.6.1 Data Collection and Tuning	26
	2.6.2 Accuracy	27
	2.6.3 Uncertainty in Parametric Models	28
	2.6.4 Missing Data in Parametric Models	29
	2.7 Other Types of Model	30
3	BAYESIAN NETS	32
	3.1 Bayesian Net Modelling Techniques	32
	3.1.1 Indicator Nodes	33
	3.1.2 Ranked Nodes, TNormal Distributions, and Weighting Functions	35
	3.1.3 Deterministic Functions	38
	3.2 State-Space Models	40
	3.2.1 The Frontier Algorithm and the Interface algorithm	43
	3.2.2 The Boyen-Koller Algorithm	46
	3.2.3 Parameter Learning in DBNs	47
	3.2.4 “Forward only” Learning in DBNs	54
	3.3 Summary	55
4	EXISTING MODELS	56
	4.1 Introduction to Bayesian Net Software Models	56
	4.2 MODIST Project Model	57
	4.2.1 Management and Communication Subnet	58
	4.2.2 Resources Subnet	59
	4.2.3 Functionality Delivered Subnet	60
	4.3 Fenton, Krause & Neil’s “Phase” Model	61
	4.3.1 Design and Development Subnet	63
	4.3.2 Defect Insertion and Discovery Subnet	64
	4.4 The Philips Model	65
	4.4.1 Existing Code Base Subnet	66
	4.4.2 Indicator Nodes Become Causal Factors	68
	4.4.3 KLOC Instead of Function Points	69
	4.4.4 Philips Model Results	71
	4.5 Wooff, Goldstein and Coolen Software Test Model	72
	4.6 Siemens Model	75

4.7	Combination with COCOMO	76
4.8	Modelling Anti-Patterns in XP	77
4.9	Rational Unified Process	78
4.10	Conclusions	78
5	EXTREME PROGRAMMING	79
5.1	XP and other Development Processes	79
5.2	Values, Principles, Practices	80
5.3	XP Practices	81
5.4	The XP Lifecycle.	83
5.4.1	User Stories	84
5.4.2	System Metaphor	84
5.4.3	Architecture	85
5.4.4	The Planning Game	86
5.4.5	Iteration Planning	87
5.4.6	Ideal Engineering Days and Load Factors	88
5.4.7	Test Driven Development	89
5.4.8	Pair Programming	91
5.4.9	Development Practises	95
5.4.10	Workspace	96
5.4.11	Acceptance Testing and Onsite Customer	96
5.5	Project Velocity	98
5.5.1	Story Points	102
5.5.2	Project Velocity and Project Planning	103
5.6	XP Models	105
5.7	Summary	106
6	ADAPTING CAUSAL MODELS TO ITERATIVE DEVELOPMENT	108
6.1	Model Size	108
6.2	Entering Data into BN Models	111
6.2.1	Quantitative Data Collection	111
6.2.2	Problems with Data Collection	113
6.2.3	Software Reliability Theory	115
6.3	Separating Model and Object Structure	121
6.4	Summary	122
7	A LEARNING PROJECT VELOCITY MODEL	124
7.1	Causal vs. Learning Models	124
7.2	Why model Project Velocity?	125
7.3	Process factors	126
7.4	Bayesian Net Model of Project Velocity	127
7.5	Iteration Model	129
7.6	Setting the initial conditions	130
7.7	Model Behaviour	132
7.7.1	Parameter Learning in Different Scenarios	133
7.7.2	Indicator Nodes	135
7.8	Model Validation	137
7.8.1	The Motorola Project	138
7.8.2	Parameter Learning	139
7.8.3	“Onsite customer” as an Indicator Node	141

7.8.4	Calibrating the Onsite Customer Node	144
7.8.5	Timescale Prediction	146
7.8.6	Accuracy of the FOFFBK Algorithm	148
7.9	Conclusions and Discussion	150
8	EXTENDING THE MODEL	153
8.1	Adding Quality to the Model	153
8.1.1	Model Structure	154
8.1.2	Validating Model Consistency	156
8.1.3	Model Learning and Prediction	158
8.1.4	Future Model Work	161
8.2	Creating an Iterative Model for Scrum	161
8.3	Extending BN Models to Other Agile Methodologies	164
8.3.1	FDD	165
8.3.2	AUP	165
8.3.3	Agile MSF	166
9	OVERALL SUMMARY AND CONCLUSIONS	167
9.1.1	Summary	167
9.1.2	Conclusions	168
	APPENDIX A - AN INTRODUCTION TO BAYESIAN NETS	170
A.1	Conditional Independence and Bayesian Nets	173
A.2	The Chain Rule for Bayesian Networks	174
A.3	D-Separation	175
A.4	Algorithms Used	179
A.5	Advantages of Bayesian Nets	180
	APPENDIX B – THE JUNCTION TREE ALGORITHM	192
B.1	Moralising	194
B.2	Triangulation	195
B.3	Join Trees and Junction Trees	196
B.4	Message Passing	198
B.5	Evidence Propagation	204
	APPENDIX C – AGENARISK SCRIPTING LANGUAGE	205
C.1	Risk Object Commands	205
C.2	Node Commands	207
C.3	Scenario Commands	211
C.4	Output Commands	211
C.5	Miscellaneous Commands	213
C.6	- Example Scripts	215
	APPENDIX D - NOISY OR	218
	APPENDIX E – FORMAL DESCRIPTION OF DATA IMPORT AND MAPPING	221
	APPENDIX F – MODEL SCRIPTS	225
	APPENDIX G - SOFTWARE COMPLEXITY REVISITED: AN APPROACH FOR USE IN CAUSAL MODELS	227
G.1	The Problem	227
G.2	Factors excluded from Technical Complexity	227

G.3 Technical Complexity Factors	229
APPENDIX H - AUTOMATED DATA IMPORT, EXAMPLE XML CONFIGURATION	234

Figures

Figure 2-1 Frequency distribution of FPs/hour	22
Figure 2-2 CC vs. LOC for methods in a large commercial program.....	24
Figure 3-1 Indicator nodes	33
Figure 3-2 Combining ranked nodes.....	36
Figure 3-3 A typical HMM	40
Figure 3-4 Types of inference in DBNs.....	42
Figure 3-5 Example DBN to illustrate the Interface Algorithm.....	44
Figure 3-7 The triangulated 1.5TBN.....	45
Figure 3-8 Learning the parameters of a Normal distribution.	48
Figure 3-9 Two similar BNs.....	48
Figure 3-10 Two alternative learning models	50
Figure 3-11 Example of evidence subnets	50
Figure 3-12 Example of regular evidence subnets.....	51
Figure 4-1 The Project Level model – high level overview.....	57
Figure 4-2 Management and communication subnet	58
Figure 4-3 Effort subnet.....	59
Figure 4-4 Functionality delivered.....	60
Figure 4-5 Phase model overview	62
Figure 4-6 Design and development subnet.....	63
Figure 4-7 Defect insertion and discovery subnet.....	65
Figure 4-8 The Philips Model	66
Figure 4-9 Existing code base subnet.....	67
Figure 4-10 Philips model results, before and after inclusion of initial phase.....	71
Figure 4-11 The COCOMO 81 BN.....	76
Figure 4-12 Anti-pattern Bayesian Net	77
Figure 5-1 Percentage of defect fixing tasks in different projects	97
Figure 5-2 A typical Burn Down chart.....	104
Figure 6-1 DEMC - the Data extraction, mapping and cleansing tool.....	113
Figure 6-2 Musa fault data - Project 1.....	117
Figure 6-3 Musa data - Project SS1C.....	118
Figure 6-4 phpMyAdmin bug reports	120

Figure 7-1 Project velocity model.....	127
Figure 7-2 PV Model as a DBN.....	129
Figure 7-3 Fragment 1 - Process effectiveness nodes.....	129
Figure 7-4 Fragment 2 - Effort nodes.....	130
Figure 7-5 Fragment 3 - Project Velocity	130
Figure 7-6 Initial Velocity model.....	131
Figure 7-7 Project velocity values V_i – median, mean, mean \pm 1 SD	132
Figure 7-8 Bias distribution iteration 10, b_{10}	133
Figure 7-9 Effectiveness Limit l_i , median, 5 iterations	134
Figure 7-10 Process Improvement r_i , median, 5 iterations	134
Figure 7-11 Bias b_i , Average scenario, median, mean \pm 1 SD.....	135
Figure 7-12 The "Collective Ownership" indicator node.....	136
Figure 7-13 Effectiveness Limit l_i with and without indicator node evidence	137
Figure 7-14 Distributions for V_i , one per timeslice.....	138
Figure 7-15 Predicted vs. actual Motorola V (medians). Actual values are bold, predicted values are dashed. The shaded area shows predicted medians +/- 2 standard deviations.....	139
Figure 7-16 Predicted and actual V , 2 observations. Actual values are bold, predicted values are dashed. The shaded area shows predicted medians +/- 2 standard deviations.	140
Figure 7-17 Relative error of model before and after learning	141
Figure 7-18 The "Onsite Customer" indicator node.....	141
Figure 7-19 Effectiveness Limit with and without indicator node evidence. Actual values are bold, predicated values (after “Onsite Customer” evidence) are dashed. The shaded area shows predicted medians +/- 2 standard deviations. The dotted line shows the learned values without “Onsite Customer” evidence.....	142
Figure 7-20 V with and without Onsite Customer evidence. Actual values are bold, predicted values are dashed. The shaded area shows predicted medians +/- 2 standard deviations. The solid grey curve shows the predicted values before Onsite customer evidence.....	143
Figure 7-21 Change in relative error with onsite customer.....	143
Figure 7-22 BN used to calibrate the Onsite Customer node.....	144
Figure 7-23 Defect effort % for each Onsite Customer setting.....	145
Figure 7-24 Project Velocity summed to date.....	146

Figure 7-25 Sum V_i to date.....	146
Figure 7-26 Sum V_i to date, Relative Error	147
Figure 7-27 Iteration 8 cumulative distributions.....	148
Figure 7-28 Magnitude of relative errors between the exact and approximate inference algorithms' PV predictions (both with and without noise).	149
Figure 8-1 - PV model updated to include code quality	153
Figure 8-2 Defects inserted d , defects found f and residual defects rd , as q is varied from Very Low (0.1) to Very High (0.9), with $E = 50$ and $V = 25$	157
Figure 8-3 Defect insertion rate dr (α) and product quality q as number of defects found varies from 1 to 10. $C = 20$, $T = 20$	157
Figure 8-4 Defect insertion rate dr and product quality q with $C = 10$, $f = 5$ and rising value of T	158
Figure 8-5 Learned values for q under different scenarios.	159
Figure 8-6 Learned values of alpha in varying scenarios.....	160
Figure 8-7 Residual defects in each iteration for different scenarios.....	160
Figure 8-8 Modelling burn down	163
Figure 8-9 Burn down graphs for the Motorola project.	164

Tables

Table 2-1 COCOMO model.....	19
Table 3-1 Evaluating the expression $A_n = A_{n-1}$	53
Table 4-1 FP to KLOC distribution parameters.....	61
Table 4-2 KLOC per FP based on ISBSG data.....	61
Table 4-3 Indicator nodes and causal factors.....	68
Table 4-4 Scale of new functionality implemented in the two models.....	70
Table 5-1 Principles in 1st and 2nd editions of Beck's book.....	81
Table 5-2 Practices in 1st and 2nd editions of Beck's book.....	82
Table 5-3 Pair Programming studies.....	94
Table 5-4 Estimating user stories.....	100
Table 6-1 Largest clique sizes in Philips junction tree.....	110
Table 6-2 Model data sources and their uses.....	111
Table 7-1 Symbol definitions.....	128
Table 7-2 PV values for three scenarios.....	133
Table 7-3 Motorola project data.....	138
Table 7-4 The true functionality delivered after iteration i is s_i . The initial prediction for s_i is s_i^0 . The MRE for s_i^0 is MRE_i^0 . The prediction for s_i after two iterations is s_i^2 . The MRE for s_i^2 is MRE_i^2 . The Mean MREs are shown at the bottom of the relevant columns.	147
Table 8-1 Quality model symbol definitions.....	154
Table 8-2 Test results when $E = 50$, $V = 25$ and q is varied from Very Low through to Very High (0.1 to 0.9 in steps of 0.2).....	156
Table 8-3 Dynamic quality model test scenarios.....	158

1 Introduction

This thesis investigates the use of Bayesian Networks (BNs) to construct agile software development process models. The main novel contribution is to introduce and validate learning Dynamic Bayesian Nets (DBNs) as a means of constructing models of agile environments.

Predicting the cost and quality of software is critical to many major business and government projects. Yet despite decades of research, even the best industry practise has a chequered history of success. Traditional approaches have proved to be either unreliable, or require sophisticated metrics collection programmes to render them reliable. Recent advances in software process modelling however, have shown that many of the problems experienced by traditional models can be overcome by using causal models based on BNs. The challenge in this research, is to adapt these advances so that they can be applied in an agile environment.

Agile methodologies vary in their exact characteristics. However they all share some common features. These include a lack of formal documentation and specification, an iterative approach to software development and a willingness to adapt to changing requirements. As a result, many of the metrics used to condition existing BN models simply do not exist in an agile environment. This gives the agile models an additional problem to contend with: if the basic input metrics are unavailable, how can the models make accurate predictions?

The solution proposed here is to take advantage of the iterative nature of the environment. Agile development teams always need *some* measure of productivity in order to plan the next iteration of their project. By combining this data with empirically derived prior distributions, it is possible to learn some basic abstract parameters which govern the productivity of the team. This allows models to be constructed which provide accurate predictions about future productivity and so deliver estimates of timescale and cost. Unlike existing approaches, there is no need to construct a full causal model of the environment - only *changes* in the environment need to be included. This dramatically reduces the amount of data that the model needs.

1.1 Hypothesis

The main hypothesis of this thesis may be stated as follows.

H1. It is possible to construct learning models of iterative agile development environments.

H2. These models require minimal “training” and minimal data collection programmes.

Proof is by demonstration. I construct a learning model of the Project Velocity (PV) metric in Extreme Programming (XP). I then show that, with surprisingly little data, it is possible to produce highly accurate estimates of future values of PV which can in turn be used to create time, effort and cost estimates.

1.2 Structure of this Thesis

The thesis is laid out as follows.

Chapter 2 discusses software process models, covering the need for both cost and quality models. Parametric models are introduced together with a discussion of some of the problems that simple regression based parametric models face. This is followed by an overview of some of the previous attempts to overcome the shortcomings of traditional modelling techniques. The chapter finishes with a brief mention of BNs and how they can address many of the problems identified with other types of model.

Chapter 3 provides the necessary background on BNs and Dynamic BNs (DBNs). An associated appendix (Appendix A) includes an overview of the conditional independence relationships inherent in all BNs and how this leads to the chain rule for BNs. This provides sufficient information to discuss the advantages of BNs when applied to software process modelling.

Chapter 4 contains a detailed examination of existing software process BNs.

Chapter 5 takes an in-depth look at XP, one of the most popular agile development methodologies. The lifecycle of a typical XP project is covered in detail together with a review of the extensive literature associated with XP and its various recommended practices. A rigorous definition is given of Project Velocity (PV), the principle productivity metric in XP. PV is common to many agile development methods, thus allowing any model of PV to be generalised beyond XP alone.

Chapter 6 argues that existing BN models are inappropriate when applied to XP. The main problems involve scalability, data collection, data entry, and data consistency across iterations. I provide a general method for importing data from multiple data sources into causal models and show how a simple scripting language can be used to separate timeslice and model structure in a DBN.

Chapter 7 describes a DBN model of PV, validated using project data from a real XP project. The model demonstrates how, with remarkably little data, and close to zero project overhead, a DBN model can learn from its environment. The model quickly adapts to a particular project and generates highly accurate predictions.

Chapter 8 discusses future directions that this research can take. In particular it points out how future defect models can be created and combined with dynamic effort models to provide classic tradeoffs between cost, functionality and quality. A simple model is described and tested for internal consistency. I also show how the XP model can be used in various other agile development environments, using SCRUM as an example of the ease with which the model can switch from PV to alternative metrics.

Chapter 9 provides a summary of the main points of the thesis, expressed in list format together with the thesis conclusion.

1.3 Published Papers

1. *Hearty, P., Fenton, N., Marquez, D., & Neil, M. (in press). Predicting Project Velocity in XP using a Learning Dynamic Bayesian Network Model. In IEEE Transactions on Software Engineering.*

This paper is largely based on chapters 5, 6, and 7 of this thesis. It includes a brief overview of the problem being addressed and discusses why BNs, and in particular DBNs, are an appropriate solution. The main XP model of project velocity is discussed with the result of the real world validation that is presented in chapter 7. The new research content of the paper is almost entirely my own work, with contributions from the co-authors on presentation, style, accuracy, terminology and background information.

2. *Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P., and Mishra, R. 2007. Predicting software defects in varying development lifecycles using Bayesian nets. Inf. Softw. Technol. 49, 1 (Jan. 2007), 32-43.*

This paper describes how the “Philips Model” (see section 4.4) can be applied to projects with different combinations of lifecycle stages. I created the component

based version of the Philips model described here and also compared the model's predictions with the actual defect values at Philips .

3. Neil M, Tailor M, Marquez D, Fenton N, Hearty P. *Modelling Dependable Systems using Hybrid Bayesian Networks. Reliability Engineering and System Safety, 2008. 93(7): p. 933-939.*

I corrected some mathematical errors and ambiguities in this paper.

4. Martin Neil, Manesh Tailor, Norman E. Fenton, David Marquez, Peter Hearty: *Modeling Dependable Systems using Hybrid Bayesian Networks. ARES 2006: 817-823.*

I corrected some mathematical errors in this paper.

5. Peter Hearty, Norman E. Fenton, Martin Neil, Patrick Cates: *Automated population of causal models for improved software risk assessment. ASE 2005: 433-434.*

This paper describes a demonstration of the heterogeneous data import capability described in Chapter 6 of this thesis.

6. Fenton NE, Neil M, Marsh W, Hearty P, Krause P, Radliński Ł. , *"Project Data Incorporating Qualitative Factors for Improved Software Defect Prediction, ICSE PROMISE 2007 .*

This paper presented the raw data and results from the Philips trials (see section 4.4). I created the regression based model against which the trial results were compared and found a stronger correlation between actual and predicted defects in the Philips model than for the regression model.

7. Neil M, Tailor M, Marquez D, Fenton N, Hearty P, *Modelling Dependable Systems using Hybrid Bayesian Networks, Reliability Engineering and System Safety, Vol 99 No. 7, July 2008, pp. 933-939.*

This paper relies on building a hierarchical model for predicting Mean Time Between Failure (MTBF) of two untested subsystems based on reliability data for three similar subsystems. The model is repetitive and requires multiple data entries. To reduce the data entry workload and the probability of data entry errors, the models were initially constructed using the scripting language described in Appendix C of this thesis, although a scripted version of the model was later found to be unnecessary.

2 Software Process Models

In this chapter we examine traditional software process models. These vary in complexity from simple parametric models through to complex simulations. Models are covered in what is essentially historical order, beginning with regression models and proceeding to multivariate models, followed by more dynamic models. We shall see that, as models become progressively more sophisticated, they incorporate a wider range of causal factors, and take an increasingly realistic view of project risk. This allows us to build a strong case for constructing causal models of software processes using BNs.

Novel contributions include a frequency analysis of developer productivity in function points per hour from the ISBSG database [90], and a comparison of cyclomatic complexity against lines of code for the classes in a large commercial computer program. I also give a detailed hierarchical description of technical complexity (via Appendix G).

2.1 Aims of Software Process Models

Most software process models fall into one of two broad categories [61]: cost models and quality models. Cost models aim to provide estimates of cost, effort and timescale. They generally take into account such factors as: the size of the requirements, the type of application, the novelty of the project, the expertise needed, interaction with other systems, level of testing required, the experience of the manufacturer.

Quality models make predictions regarding defect counts and mean time to failure. They are used to determine when a software product has reached the necessary level of quality and to allocate appropriate support after release. In addition to the factors listed above, they often include: code size, complexity metrics, defect reports. There is clearly a large overlap between the constituent data used in both types of model. Data collected in order to support a cost model can often be reused in a complementary quality model.

2.2 Regression models

The most common types of software process models are regression models based on empirically derived relationships. Data is gathered on two or more measurements

which are believed to be related. A numerical relationship is then derived between these measurements. Some attempts may be made to justify this relationship, usually after the fact, but there is normally little or no theoretical foundation from which to derive it.

As early as 1974, Wolverton [217] proposed a series of simple correlations between various software and management measurements. In 1976, Walston [205] derived a simple empirical relationship for use in projects at IBM.

$$E = 5.2L^{0.91} \quad \text{Equation 2-1}$$

Where effort E is in man months, and code size L is in thousands of lines of code (KLOC).

The most famous of this class of models is the COConstructive COst MOdel, COCOMO, introduced by Barry Boehm in 1981 [24]. This considers three types of software projects: organic, semidetached and embedded, supposedly in order of increasingly strict compliance with specifications. “Organic” systems are essentially data processing systems, while “embedded” correspond to real time systems. “Semidetached” systems combine elements of both – a real time data feed for example. The equations and coefficients are shown in Table 2-1. (This model uses Thousands of Delivered Source Instructions - KDSI).

COCOMO equations				
$E = aL^b$	$E = \text{effort in man-months}$ $L = \text{code size in KDSI}$			
$D = cE^d$	$D = \text{calendar months}$			
COCOMO coefficients				
	a	b	C	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 2-1 COCOMO model

It is interesting to note, that even at this early stage, researchers had identified the need for such models to be tailored to their environment by selecting one of three

project types. The model in Table 2-1 is known as the *Basic* COCOMO model. An additional multiplier can be applied when more information about a project is known.

In general, if we assume a relationship of the form $E=aL^b$, between E and L , then the values of a and b can be calculated from a set of sample projects with known E and L by simply plotting $\log(E)$ against $\log(L)$ and performing a regression analysis (usually least squares). Note that, by doing this, we are losing information. Some knowledge about the variation in the data points is lost. In the case of “outliers”, this procedure may even explicitly discard some data, although it should be noted that later versions of COCOMO incorporated more causal factors in an attempt to explain outliers.

Simple defect prediction models arose at the same time as early effort models. For example, Akiyama [10] of Fujitsu, in Japan, constructed the following formula:

$$D = 4.86 + 0.018L \qquad \text{Equation 2-2}$$

where D is the number of defects (in testing and two months after release) and L is the number of lines of code. As discussed in section 2.6.1, care must be taken in the definition of a “defect”.

All of the early authors were well aware of the crudity of these models. Walston, for example, tried to augment Equation 2-1 by introducing measures of programmer productivity. These measures relate to many of other factors, such as interface complexity, experience of the programmers, quality of requirements and so forth. Akiyama proposed alternatives that included the number of decisions in a program, and the number of subroutine calls. Boehm extended the basic COCOMO model to include a set of hardware, product, personnel and project attributes which could be used to adjust the basic model.

2.3 Function Points

A major problem with the original COCOMO model is that it is driven by a variant of LOC. Since LOC are only available at the end of a project, we must make do with estimating its value in order to make predictions at the start of a project. But estimating LOC can be just as hard as estimating the effort that we are trying to predict. COCOMO II overcomes this weakness by allowing project size to be estimated from the specifications. COCOMO II was further updated in 1998 by

Chulani, Boehm and Steece [43] using a Bayesian analysis technique (see Chapter 3 for an overview of Bayes techniques). The authors claimed that this resulted in greater accuracy for the resulting model.

An industry standard has emerged to measure project size without the need for LOC. Function Points (FPs) claim to provide a technology independent means of quantifying the size of a software specification [11]. FPs take into account the number of interfaces, files and queries in a specification. These are then weighted according to their complexity: simple, average or complex. The sum of these complexity weighted attributes is the Unadjusted FP Count (UFC). This is then multiplied by a Technical Complexity Factor (TCF) which is composed of a variety of technical and project factors which make the development more complex. The result is an Adjusted FP Count (AFC).

The International Function Point User Group (IFPUG) maintain a manual of FP counting practises [89] which is updated regularly to reflect modern programming techniques. This enables FP counting to be applied in GUI, object oriented and web based applications, even though it was originally designed for batch based database systems. According to Hotle [86], most trained IFPUG counters produce FP counts which differ from each other by no more than $\pm 11\%$ for the same specification.

FPs can be used as an input to effort prediction models. This can be a simple combination of the number of FPs multiplied by team productivity. Alternatively, they can be used as inputs to regression based models, such as COCOMO. The big advantage of FPs over LOC is that they are available before development commences. By concentrating on functional requirements, FPs also avoid the many ambiguities associated with LOC.

Fenton and Pleege [61] have summarised some of the criticisms of FPs. These include:

1. The subjective nature of the TCF.
2. Possible double counting of internal complexity.
3. Difficulty in comparing FP counts at project start and project end. These can vary by several hundred percent [110]. Because of this difference, any metrics collection programme must take care which values are stored and used for future predictions.

4. FPs are not completely technology independent. See for example Verner and Tate [204], and Ratcliff and Rollo [168].

Updated versions of FPs have been proposed, such as Mark 2 Function points by Symons [200], Object Oriented Function Points by Antoniol et al. [14] and Class Points by Costagliola et al. [51]. Despite these criticisms and alternative proposals, the original Function Point counting method (as updated by IFPUG) remains the most popular. For example, in the ISBSG database [90], which contains metrics for over 3,000 software projects, IFPUG counting accounts for nearly 90% of the projects listed.

The IFPUG database allows us to examine the productivity of developers and create a frequency chart for productivity (Figure 2-1 – a log-normal distribution is shown as a continuous line for comparison). The median productivity for this database is approximately 0.1 FPs per person-hour. However there is a 26% chance that productivity will be less than half this value and a 23% chance that it will be more than double. Clearly, if these variations are not taken into account, any effort predictions based on FPs have a considerable margin for error. This type of variation in productivity is well known and is not restricted to FPs [175] [165].

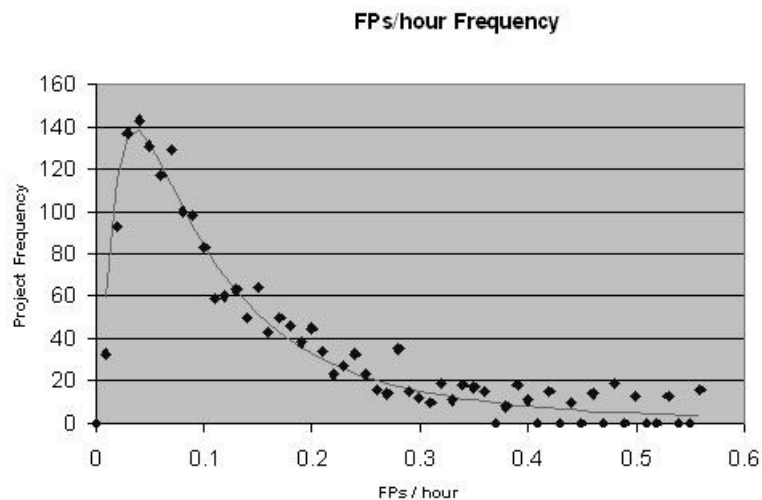


Figure 2-1 Frequency distribution of FPs/hour

The problem with FPs in agile models is that they rely on a complete, detailed specification being prepared in advance of the development stage of a process, something which we certainly cannot rely on in agile environments. However, they remain important because they demonstrate the uncertainty involved in judging developer productivity. FPs are a relatively well defined measure of problem

description. If FPs result in such uncertainty, then we can expect agile perceptions of project size, which are often based on much less well defined criteria, to be even more uncertain.

2.4 Including Complexity

We have already seen how FPs attempted to include software complexity into regression based models. In both effort and quality prediction models it is assumed that complexity will have a major effect. Complex software is assumed to be harder to write and therefore more likely to contain program errors. Similarly, more complex software is assumed to require greater attention and more debugging time, therefore effort should also increase.

The word “complexity” is overloaded in its use in the English language. This overloading extends to its use in technical and engineering environments. For example, when we say that a software project is complex, we could be referring to any of the following complications.

- (1) The project is large and requires multiple relationships between many managers and many development teams.
- (2) The project is scattered across several locations.
- (3) The project must handle radically different types of data and interfaces.
- (4) The project involves new technology where there is little previous experience of its use.
- (5) The functionality requires a compromise between competing requirements.
- (6) There are strict requirements in terms of quality, performance, cost or delivery times.
- (7) Specialised algorithms must be employed.
- (8) The structure of the software is complex as measured by some well defined metric.

In Appendix G, I give a much more detailed discussion of complexity in software projects. Following Baccarini [17] and Xia and Lee [221], I draw a distinction between *management complexity* and *technical complexity* and show how a hierarchy of technical complexity factors can be constructed.

Computer Science has traditionally concentrated on point (8) above. For example, McCabe’s cyclomatic complexity [130] measures the number of independent paths through a piece of software. Watson and McCabe [208] point to several studies which

seem to validate the claim that cyclomatic complexity is a good predictor of defect counts. However, the cyclomatic number is also a strong indicator of program size [187]. All but the most trivial of programs will include some conditional branching, and the larger the program, the more branches there are likely to be. This can be seen in Figure 2-2 which shows LOC against cyclomatic complexity for the classes in a large, professionally developed program. Kitchenham et al. [111] give a more systematic approach to this question.

This represents a general problem with static code metrics. Number of attributes, number of methods, number of statements etc. are often used as complexity metrics. All are strongly proportional to LOC.

Apart from this relationship between program size and cyclomatic complexity, this metric has also been criticised for giving equal weight to all types of code branches, even though it is widely believed that nested flow control changes are much more likely to lead to programming defects [164][191].

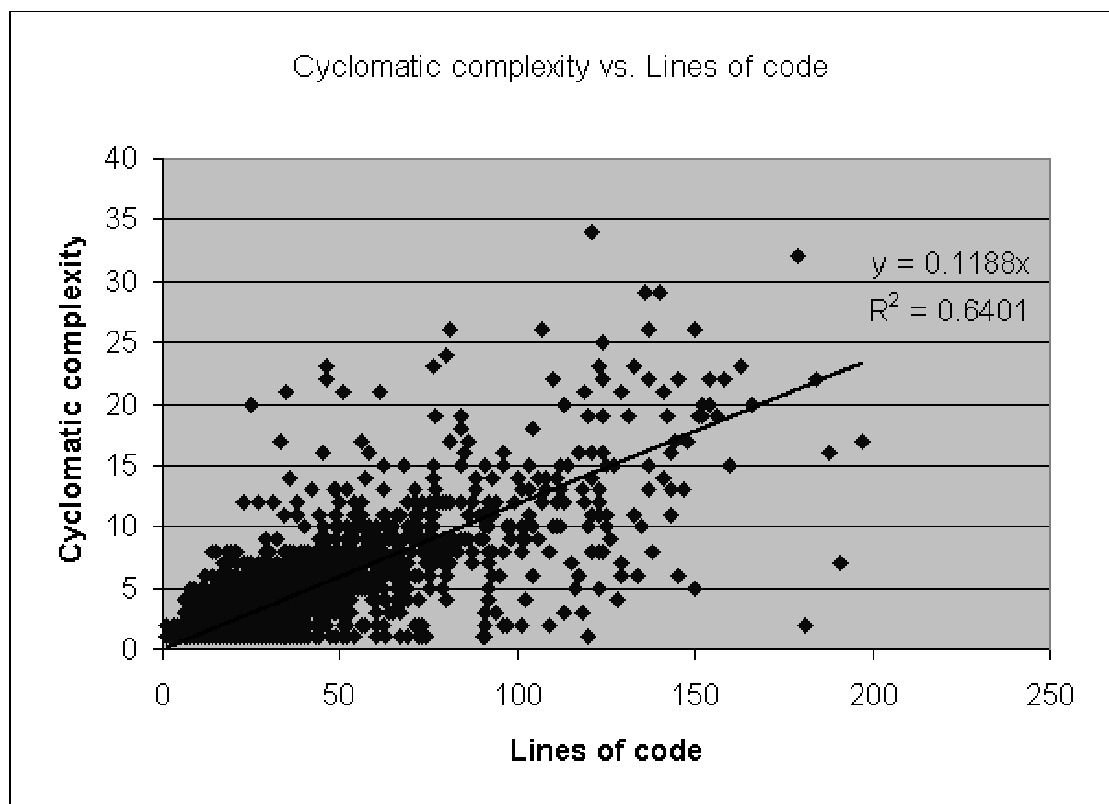


Figure 2-2 CC vs. LOC for methods in a large commercial program

The McCabe complexity metric is one of the best known, however it is not the only way to measure software complexity. For example, the coupling and cohesion [61] of a module can also be regarded as a measure of complexity. High coupling between modules reduces the maintainability of code, leading to higher defects and higher

costs. Conversely, high cohesion within a module makes its behaviour clear and predictable, leading to lower defects and costs. Equivalent definitions exist for OO software (see for example [40] and [124]).

In addition to colinearity with LOC, all of these software complexity metrics have other problems. While they may account for some of the variability seen in software process models, they cannot account for all of it. They take no account of data structure complexity (although other complexity metrics do [143][220]), programmer experience, test quality, process maturity, problem domain or any of the other host of factors that affect final defect count and production costs. Given these additional factors, it makes sense to see if multivariate models, which attempt to combine a large number of factors into the model, can be more successful.

2.5 Multivariate Models

Multivariate models attempt to extend size and complexity models by including other aspects of the software development process. These can include different size and complexity metrics as well as metrics covering requirements, testing and process quality.

One major challenge with all multivariate models is to minimise colinearity. As mentioned above, it is easy to choose size and complexity metrics which are significantly correlated. When dealing with all of the factors that contribute to software cost and quality there are many more such opportunities. As Munson and Khoshgoftaar have pointed out [142], models based around up to 6 orthogonal metrics can be almost as accurate as those using up to 30, demonstrating that multiple metrics are in fact measuring the same underlying attribute.

Colinearity is minimised using techniques such as Principal Component Analysis (PCA – see for example Lay [118]). This applies a linear transformation to the data such that the variance of the data is maximised along the new basis vectors. (The components with the largest variance vary the most with respect to the predicted variable so a large change in the component causes a large change in the prediction). The components with the smallest variance are closer to being constant and if below some threshold value can be discarded. A similar technique, Factor Analysis [114], can also be used to create multivariate models. A detailed discussion of the application of multivariate techniques to software metrics can be found in Prof. Martin Neil's Ph.D. thesis [150].

While this successfully reduces the data to a set of nearly orthogonal metrics, the results are difficult to interpret in terms that would be understood by developers. Each basis metric is now a linear combination of software engineering metrics. If we wish to reduce the cost or defect count for a given class or module, it is not clear how to proceed since the same result can be obtained by varying the underlying software metrics in multiple ways.

2.6 General Problems with Parametric Models

All of the models discussed so far result in a simple set of algebraic formulae. The number of metrics used in the formula and the methodologies used to establish coefficients or exponents may vary, but their usage is always the same: a manager is required to collect some metrics, put the values into the formula as parameters and a prediction is made about effort, defects, or some other related value. This section discusses some of the problems with this approach.

2.6.1 Data Collection and Tuning

These parametric models often require tuning to the local environment. There are a variety of reasons for this.

1. Definitions may vary. A “defect” in one environment may be a simple failure of a system test, whereas in another it may only be defects in released software which count. The severity of a fault may also be significant. A customer request to change the background colour of a form may be classified as a defect by one organisation but as a change request by another.
2. There is a similar uncertainty surrounding what actually constitutes a line of code. We must decide whether to include comments, blank lines, declarative lines, required syntactic elements and so forth. Then there is the question of how LOC should be compared across different programming languages and different coding standards. This problem is discussed at length in [61]. This is not to suggest that LOC is not a useful measure of program size. However, we must always ensure that its exact meaning in any given circumstance is carefully specified and that we are comparing like with like.
3. Standards for project size estimation vary. For example, function point counting can vary between different organisations.

4. The development process, environment and team skills can all vary, leading to the type of uncertainty in productivity that we saw in Figure 2-1. Models such as COCOMO try to take account of these variations by introducing a set of *cost drivers* that modify the basic COCOMO equations. However even with these modifications, the accuracy of the model is poor (see below).

Tuning a model to a particular organisation requires data collection and analysis, often as part of projects which do not directly benefit from the process. In 2004, the dominant method of software cost estimation remained expert judgement, with little evidence that it was any less effective than model based assessments [101]. Given this fact, many project managers regard data collection programmes as little more than an unwelcome overhead.

Empirical formulae also provide no mechanism to explore “what-if” style analysis. For example, suppose a company decides to invest in greater staff training. How will this affect Equation 2-1? We cannot tell which of the two constants to vary in order to reflect our (hopefully) improved performance. We can always conduct another regression analysis, but how do we know how much of the change is the result of better trained staff, and how much is due to different requirements or other small variations in conditions? To do this, a company would have to perform a separate regression analysis on trained and untrained staff working on identical projects with otherwise identical resources. In practise this is impossible in an industrial setting.

2.6.2 Accuracy

The accuracy of parametric models also continues to be in doubt. Kemerer [109] found variations of up to 772% between actual and predicted effort, although it is possible to tune COCOMO-like models to work in different environments [77][25].

Molokken and Jorgensen [137] performed a review of surveys of software effort estimation and found that the average cost overrun was of the order 30-40%. One of the most famous such surveys, the Standish Report [194] puts the mean cost overrun even higher, at 89%, (although this report is not without its critics [73]).

Briand et al. [28] used data collected by the same tool on 206 projects from 26 companies. They used various techniques to construct cost models, including ordinary least squares, analysis of variance (ANOVA), classification and regression trees (CART – discussed in section 2.7) and analogy based methods (the last two being non-parametric models).

If we define the actual effort in project i as A_i , and the predicted effort as E_i , then the Magnitude of Relative Error for project i is given by: $MRE = |A_i - E_i| / A_i$. Briand et. al. found that all of the models they constructed had a mean MRE of 0.5 or greater. The median values, which are less sensitive to outliers, fared slightly better, managing a median MRE of 0.41.

An alternative measure of prediction accuracy is the percentage of predictions which are within 25% of the actual value. This is often referred to as PRED(25). The models had PRED(25) level of between 14% and 34%. In other words, even the best model's predictions were out by more than 25% more than two thirds of the time.

2.6.3 Uncertainty in Parametric Models

Parametric models tell us nothing about the uncertainty of their estimates. We know that this uncertainty is significant for several reasons.

1. The inaccuracy of many of the results, as detailed in [28] and [109], for example, tells us that the predictions can often be significantly different from the actual value.
2. Some of the inputs to the models consist of subjective expert judgements. For example, suppose an expert is asked to judge the complexity of a data interface on a five point scale when the true complexity resolution should be much finer. There is immediate uncertainty from the fact that the expert's choice spans 20% of the whole range of options. If the expert's judgement is borderline, where the true value could be contained in one of two values on the five point scale, then the "true" value could lie anywhere in a 40% range.
3. If the model has not been tuned to the local environment, then it is probably based on coefficients and exponents chosen to form a best fit with a wide range of projects. It is unlikely that these parameters will be a good fit for any particular project. For example, suppose we are building a cost model and have already estimated the size of the software in FPs. We now choose a value for productivity based on the median value from Figure 2-1 and divide it into the number of FPs in order to estimate the number of programmer-hours needed. We know from the shape of Figure 2-1 that we have nearly a 50% chance of being out by a factor of two.

Many of these sources of uncertainty are present no matter what type of process model we build. However, in parametric models there is no explicit mechanism for

expressing or quantifying this uncertainty. We can try varying the inputs to a parametric model, using a simulation to explore its sensitivity to measurement errors, but this is a cumbersome, ad-hoc, approach that has to be added onto the model. It is not part of the model itself. We discuss this type of simulation further in the next section on missing data.

2.6.4 Missing Data in Parametric Models

What should we do when required data is missing in parametric models? Suppose we are constructing a defect model and our software includes significant amounts of third party software. We would like to include the number of defects from the third party software in our model, yet we may know little or nothing about the conditions under which it was created.

As with uncertain data, we can use estimates of the missing data. One possible approach is to estimate the minimum, maximum and mean values of the missing data and run the model with each value. However, we also need to estimate the probability of each estimate occurring. We can improve on this by choosing more estimates from what we believe to be a likely range of possibilities, in each case attaching a probability to our estimates. In effect, we are building a probability distribution for the missing data and determining its effect on the model.

Now suppose we have, not one piece of missing data, but two. Again, we might start by choosing a small number of discrete values for each missing data value. Now however we must try out each combination of values in our model in order to determine the full range of possible results. If we have N required data items missing, and we choose M discrete states for each, then we must run our model M^N times to get the full range of results. In addition, if we wish to determine the probability of each model outcome, we must also determine the Joint Probability Distribution (JPD) for these M^N states. If the missing variables are independent then the JPD is simply the product of the probability distributions of each missing variable. However, we are unlikely to be this lucky due to the co-linearity between software metrics.

As we shall see in the next chapter, the kind of calculation that we have just attempted to do by hand in a parametric model is precisely the calculation that occurs naturally in a BN. Furthermore BN models are not restricted to dealing with independent variables. Causal relationships between variables are explicitly encoded in BN models and are taken into account when calculating the JPD.

2.7 Other Types of Model

Several alternatives have been suggested to parametric models[29]. Classification and regression trees (CART) ask a series of “IF...THEN...ELSE” questions about a project. Each set of answers leads to a specific value for some continuous variable. Techniques are available for deciding what questions to ask and how deeply to nest the decision tree [27]. Srinivasan and Fisher [192] found that CART models using the COCOMO data had an MRE > 3.6 . This was considered an improvement on COCOMO itself, which had an MRE > 6.1 .

Shepperd and Schofield [188] recommended a Case Based Reasoning (CBR) method of cost estimation. A database of software projects is built. Each new project is then compared against the existing projects, with each project attribute defining a dimension in n-dimensional space. The project with the shortest Euclidean distance to the new project is used as the basis for the effort estimation. In general, analogy outperformed regression based models using both the MRE and PRED(25) measures. However, Briand et al. [28] found them to be worse.

Finnie et al. [68] used Artificial Neural Networks (ANNs) trained using a random sample of 50 projects chosen from a dataset of 299. The MRE for these ANNs was approximately 0.3, considerably better than the 0.6 for regression models based on the same data set. The same study also created CBR models which also had MREs of approximately 0.3. The PRED(25) measures of the ANN and CBR models also improved over the regression based models.

Although better at predicting costs, the CART, CBR and ANN models all require a data collection and training phase, just as regression models do. They also share the property with regression models, that they provide only point values, with no automatic assessment of risk or uncertainty. In addition, ANN project models, are open to the criticisms levelled at all ANNs: namely that ANNs are “black boxes” where it is often difficult to map the model to rule based reasoning. i.e. ANNs give us no insight into the reasoning behind the results.

Software process simulation models, such as Abdel-Hamid’s system dynamics model [1][2], view the software process as a series of sources, sinks, queues and feedback loops. So, for example, a requirement acts as a source of code requirements, whereas coding acts as a code requirement sink which in turn acts a source of bugs.

Testing is a bug sink, but is itself a code requirement source which feeds back into the coding sink.

System dynamic models produce timeseries predictions of effort, cost and defects. By varying the inputs, simulation models can also provide limited decision support, tradeoff analysis and risk analysis. Ruiz et. al. [174] considered the full Abdel-Hamid model to be too large and too complex, requiring too many metrics, and so reducing its domain of applicability. They therefore proposed a simplified version of the model, but based on the same essential principles. A good summary of software process simulation in general can be found in [108].

However, as Christie has pointed out [42]:

“The predictive power of simulation is strongly dependent on how well the models are validated. Although many scientific and engineering fields can base their models on established physical law, organizational models have to deal with human and other less quantifiable issues. Not only is gathering data difficult when that data must come from human actors, the reproducibility of scenarios used to validate models cannot as easily be standardized as in experiments based on physical law.”

In other words, simulation of human processes is subject to the same limitations of lack of data and measurement uncertainty as we found in simpler regression based models.

3 Bayesian Nets

We have seen how previous efforts to model the software development process have met with mixed levels of success. There are wide variations in the requirements, resources, skills and techniques in each project. This means that parametric models, AI models and simulation models, all need an extensive data collection and/or training phase. The presence of a mixture of expert judgement and often patchy data only serve to complicate this further.

Appendix A provides a short introduction to the basic theory behind Bayesian nets (BNs) and how they can overcome the problems described in the previous chapter. Any reader who is unfamiliar with BNs is encouraged to read Appendix A before proceeding. Readers unfamiliar with the Junction Tree Algorithm (JTA) are also advised to read Appendix B.

In this chapter I concentrate on some of the techniques used by BN software process models. I then define state-space models and in particular define Dynamic Bayesian Nets (DBNs) which will be needed to build agile models in later chapters.

The novel contributions introduced in this chapter are:

1. A more compact notation for probability distributions for use when describing BNs. (This is only used extensively in Appendix B so as not to place a learning burden on anyone familiar with the more conventional notation.)
2. Suggestions for incorporating aspects of the Shenoy-Shafer algorithm into the junction Tree Algorithm (Appendix B).
3. A proof of the equivalence of parameter learning in BNs and DBNs and a demonstration of how such parameter learning can be achieved using the implementation of deterministic functions in AgenaRisk.
4. A description of the advantages of *Forward Only Learning* in DBNs and the circumstances where it is appropriate.

3.1 Bayesian Net Modelling Techniques

A Bayesian Network (BN) is a Directed Acyclic Graph (DAG), where the nodes represent random variables and the arrows represent causal influences. Nodes without parents are defined by a random variable's probability distribution. Nodes with parents are defined by Conditional Probability Distributions (CPDs). Probability distributions and CPDs are collectively known as Node Probability Tables (NPTs).

It can be a difficult task to elicit CPDs from experts. If there are a number of factors involved then the size of the combined state space grows exponentially with the number of factors. Deriving a consistent, comprehensible and comprehensive CPD in such circumstances is not a trivial task. In this section we cover some of the techniques which have been used to build the models discussed in this thesis.

3.1.1 Indicator Nodes

A technique used extensively by many of the models introduced in chapter 4 is *indicator nodes* [65]. These are nodes with no children and a single parent. They are often used to enter measures of an underlying node (the parent) which cannot be directly measured. An example in software engineering might be a node like “Process quality”. Software development process quality is an abstract quantity that cannot be directly measured. However there are strong indicators of process quality, such as Capability Maturity Model Integration (CMMI) level [44], requirements management, risk management etc.

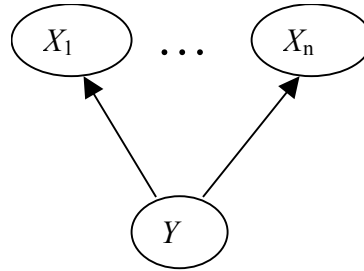


Figure 3-1 Indicator nodes

An example of a BN with a single parent Y and a set of indicator nodes $\{X_i : i = 1, \dots, n\}$, is shown in Figure 3-1. The JPD is given by:

$$P(X_1 \dots X_n, Y) = P(Y) \prod_{i=1}^n P(X_i | Y) \quad \text{Equation 3-1}$$

Marginalising gives the posterior distribution for Y :

$$M(Y) = \sum_{X_i} (P(Y) \prod_{i=1}^n P(X_i | Y)) = P(Y) \prod_{i=1}^n (\sum_{X_i} (X_i | Y)) = P(Y) \quad \text{Equation 3-2}$$

So with no evidence present, indicators have no effect on their parent. Suppose we now enter evidence in indicator node X_1 . Represent this evidence by the distribution Λ_1 . Equation 3-2 then becomes:

$$\begin{aligned} M(Y) &= \kappa \left(\sum_{X_i} P(Y) \prod_{i=1}^n [\Lambda_i P(X_i | Y)] \right) = \kappa P(Y) \prod_{i=1}^n \left[\sum_{X_i} \Lambda_i P(X_i | Y) \right] && \text{Equation 3-3} \\ &= \kappa P(Y) \left(\sum_{X_1} \Lambda_1 P(X_1 | Y) \right) \prod_{i=2}^n \left[\sum_{X_i} P(X_i | Y) \right] = \kappa P(Y) \left(\sum_{X_1} \Lambda_1 P(X_1 | Y) \right) \end{aligned}$$

Where κ is a suitable normalising constant. i.e. Evidence in an indicator node back propagates to update the posterior distribution of Y .

Deciding when to use indicator nodes is not always straightforward. Returning to our software engineering example, the question of whether good requirements management is an *indicator* of good process quality, or whether good requirements management *causes* good process quality is an open one. The answer does more than just dictate the direction of the arrow on a graph.

Let us simplify the example shown in Figure 3-1 by having only a single indicator node, X . As before, the marginal distribution of Y is just its prior. However, consider the case where X is now interpreted as a cause of Y . i.e. The arrow goes from X to Y and not the other way around. The marginal probability distribution of Y is now given by:

$$M(Y) = \sum_X P(Y | X) P(X) \quad \text{Equation 3-4}$$

Y 's dependency on X means that X 's prior affects the marginal distribution of Y . This was not the case when X was an indicator node of Y .

If we turn the multiple indicator nodes of Figure 3-1 into multiple *causes* of Y then we must find a way to combine those causes into a single conditional probability distribution for Y , a problem that we do not face when the X 's are indicator nodes. This CPD is potentially very large. For every possible combination of states of $\{X_i\}$, we must define a probability distribution for Y . If Y has m states and each X_i has n_i states, then the CPD will have $m \prod n_i$ values in total. There are two problems with such large CPDs.

- 1 They are computationally complex, affecting the size of every clique and sepset in which they appear in the JTA.
- 2 It is difficult to elicit large consistent CPDs from experts.

One technique for dealing with the computational complexity of large numbers of causal factors (typically more than 4) is to introduce intermediate nodes which group the factors into a tree. Techniques for addressing the elicitation problem are discussed in the section below on “Ranked Nodes, TNormal Distributions, and Weighting Functions”.

The D-separation properties of indicators and causes are also quite different. In the case of multiple causes, each X_i is D-separated from the others until evidence is entered in Y (or one of its descendents if it has any). The opposite is the case when each X_i is represented as an indicator: indicators are D-connected until Y is blocked by some evidence. These independence considerations should be one of the primary determinants when deciding whether nodes should be causes or indicators.

3.1.2 Ranked Nodes, TNormal Distributions, and Weighting Functions

Many of the models introduced in section 4 make extensive use of ranked nodes, particularly those defined using the Truncated Normal (*TNormal*) distribution [52]. This is a Gaussian distribution that has been truncated at both ends.

Just as with a Gaussian distribution, the TNormal must specify its mean and standard deviation. Unlike the Gaussian distribution however, the domain of the distribution must also be specified. In the case of ranked nodes, this domain is always the interval $[0, 1]$. The TNormal is an extremely versatile distribution. With appropriate choices of parameters it can approximate a Gaussian, a uniform, a monotonically increasing or a monotonically decreasing distribution. There is also no need for the distribution to be fixed at zero at its extremes. This is an extremely useful property which often appears in distributions elicited from experts.

Let $N(\mu, \sigma^2)$ be a normal distribution with mean μ and standard deviation σ . Let $\phi(x)$ be the probability density function for N . We write the truncated normal for N over the interval $[a, b]$ as follows.

TNormal (μ, σ, a, b)

Where TNormal (μ, σ, a, b) has a probability density function, $\phi(x)$, given by:

$$\phi(x) = \frac{\phi(x)}{\eta}, \eta = \int_a^b \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt.$$

When the node is a ranked node, which is always defined over [0, 1] in AgenaRisk, we will omit the upper and lower bounds.

Another property that crops up regularly in expert opinions is the behaviour of ranked nodes with multiple parents. An example of this, taken from [65], is shown in Figure 3-2.

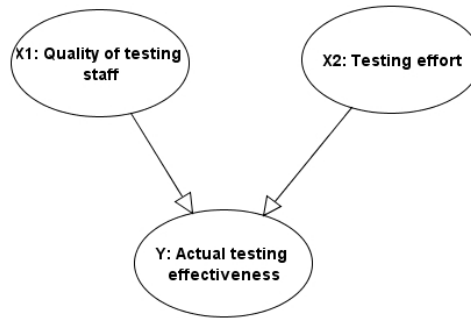


Figure 3-2 Combining ranked nodes

As Fenton and Neil point out in [65]:

In this case we elicit the following information:

- 1) *When X1 and X2 are both 'very high' the distribution of Y is heavily skewed toward 'very high'.*
- 2) *When X1 and X2 are both 'very low' the distribution of Y is heavily skewed toward 'very low'.*
- 3) *When X1 is 'very low' and X2 is 'very high' the distribution of Y is centred toward 'very low'.*
- 4) *When X1 is 'very high' and X2 is 'very low' the distribution of Y is centred toward 'low'.*

Intuitively, the expert is saying here that for testing to be effective you need not just to have good people, but also to put in the effort. If either the people or the effort are insufficient then result will be poor. However, really good people can compensate, to a small extent, for lack of effort.

We could try to capture this using a simple weighted average over the parent nodes.

$$\text{wmean}(X_1 \dots X_n) = \frac{\sum_{i=1}^n w_i X_i}{\sum_{i=1}^n w_i} \quad \text{Equation 3-5}$$

In Fenton and Neil's example above, point (3) suggests that the largest weighting should go to X_1 . However, point (4) would then force the weighted mean to give a value of 'very high'. This suggests that we need a new function to produce the CPD elicited from the experts.

The function used to combine X_1 and X_2 is a bit like a minimum function, except that X_1 can be weighted. To this end, we define a weighted minimum function, wmin , as follows.

$$\text{wmin}(X_1 \dots X_n) = \min_{i=1..n} \left[\frac{w_i X_i + \sum_{j=1, j \neq i}^n X_j}{w_i + (n-1)} \right] \quad \text{Equation 3-6}$$

Where the w_i are weightings which can be applied by the model developer. When all the weightings are set to one, wmin returns the mean value. When all the weightings are equal and sufficiently large, wmin returns a value close to the minimum value. For all other settings, wmin returns a value between these two.

A similar function, wmax , returns a value between the average and the maximum of its inputs.

$$\text{wmax}(X_1 \dots X_n) = \max_{i=1..n} \left[\frac{w_i X_i + \sum_{j=1, j \neq i}^n X_j}{w_i + (n-1)} \right] \quad \text{Equation 3-7}$$

Even for relatively simple BNs, such as the one shown in Figure 3-2, ranked nodes with just five states each leads to an NPT for the child node of 125 entries – something which would be difficult to elicit explicitly. The combination of ranked nodes, TNormal distributions and weighting functions allows a wide category of NPTs such as these to be construct relatively easily. Ranked nodes provide a simple, ordered set of values whose effects can be easily described be experts. The TNormal distribution provides a flexible distribution which is not fixed at zero at its extremes – something which is needed in order to construct distributions in ranked nodes which are heavily skewed to one of the ranked node’s extreme values. Finally, the weighting functions allow subtle interactions between multiple causes to be efficiently captured.

3.1.3 Deterministic Functions

Appendix A shows how we can define CPDs for discrete approximations to continuous nodes and how standard statistical distributions can be used to initialise these CPDs. However, there is another way to define the CPD of a node, namely via an algebraic expression based on its parents. In other words, given a node A , discretised using a set of states, $\{a_i\}$, with CPD defined as some function of its parents, $f(\pi(A))$, we wish to find the probability that A will be in any given state given the states of its parents.

One possible approach is to sample f for each combination of parents, count the results for each child state a_i , and normalise when finished. However, as explained in [153], this approach runs the risk that some of these counts will be zero simply because of the sampling method. This error will then be propagated throughout the model. Further, if the erroneous zero value is observed and entered as hard evidence then the model will be inconsistent. AgenaRisk, the tool used to build my own models, takes the alternative proposed in [153] and described below.

for each combination of parents states q

for each combination of upper and lower bounds in q, b_i

$$x_i = f(b_i)$$

$$l = \min(\{x_i\})$$

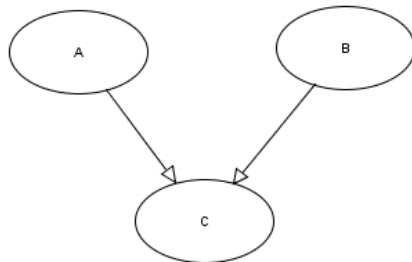
$$u = \max(\{x_i\})$$

$$P(A \mid \pi(A)=q) = \text{Uniform}(l, u)$$

As the discretisation of A and $\pi(A)$ is made finer, the uniform distribution assumption will become a better approximation to f . The following example illustrates the method.

Example 3-1

Consider the case of three continuous nodes A, B and C with the states and probabilities shown below.



A, B states	$P(A)$	$P(B)$
0.0 - 1.5	0.3	0.5
1.5 - 2.0	0.7	0.5

C states	$P(C A,B)$
0.0 - 1.0	$f(A) = A + B$
1.0 - 2.0	
2.0 - 3.0	
3.0 - 4.0	

For each combination of parent states we evaluate f , using the possible upper and lower bound combinations, and record the maximum and minimum values. These are shown in the table below.

A	0.0 - 1.5	0.0 - 1.5	1.5 - 2.0	1.5 - 2.0
B	0.0 - 1.5	1.5 - 2.0	0.0 - 1.5	1.5 - 2.0
min (A+B)	0	1.5	1.5	3
max (A+B)	3	3.5	3.5	4

The CPD for C is then set to uniform distributions across the above limits.

A	0.0 - 1.5	0.0 - 1.5	1.5 - 2.0	1.5 - 2.0
B	0.0 - 1.5	1.5 - 2.0	0.0 - 1.5	1.5 - 2.0
0.0 – 1.0	1/3	0	0	0
1.0 – 2.0	1/3	1/4	1/4	0
2.0 – 3.0	1/3	1/2	1/2	0
3.0 – 4.0	0	1/4	1/4	1

Where the uniform distribution does not cover an entire state from node C, it is adjusted in proportion to the amount that it does cover.

Note that the CPD depends only on f and on the states of the parents. As evidence in the BN changes, the JPDs and marginals will vary, but the CPD will remain fixed.

3.2 State-Space Models

State space models assume that a system can be modelled by an underlying state which evolves over time but which is normally unobservable. We can, however, observe indicators of the system state that cause us to update our belief about the unobserved variables.

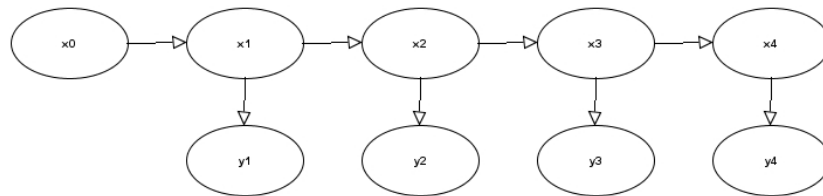


Figure 3-3 A typical HMM

Hidden Markov Models (HMMs) [167] are the simplest possible state-space models. In HMMs, such as the one shown in Figure 3-3, there is an initialisation vector, x_0 , a series of unobserved system states, x_i ($i > 0$), and a series of observations, y_i . All variables are discrete. In each case the *transition matrix* $P(x_{i+1} | x_i)$ and the *observation matrix* $P(y_i | x_i)$ are fixed.

If T is the number of observations then we can calculate the JPD as follows.

$$P(x_{0..T}y_{0..T}) = P(x_0) \prod_{i=1}^T P(x_i | x_{i-1})P(y_i | x_i). \tag{Equation 3-8}$$

We can then marginalise to find the probability of any given node. This procedure is exponential in T however and so is not a practical approach. The *Forward-Backward* algorithm, described in [167] takes advantage of the fixed nature of the transition and observation matrices to create an algorithm which grows as $O(N^2T)$, where N is the number of states in $x(i)$.

Dynamic Bayesian Nets (DBN) generalise HMMs by allowing both the hidden and observed variables to consist of multiple nodes with complex dependencies. They extend BNs by adding a temporal dimension to the model. Formally, a DBN is a temporal model representing a dynamic system, i.e. it is the system being modelled which is changing over time, not the structure of the network [144]. A DBN consists of a sequence of identical Bayesian Nets, \mathbf{Z}_t , $t = 1, 2, \dots$, where each \mathbf{Z}_t represents a snapshot of the process being modelled at time t . We refer to each \mathbf{Z}_t as a *timeslice*.

The models presented here are all first order Markov. This means that:

$$P(\mathbf{Z}_t | \mathbf{Z}_{1:t-1}) = P(\mathbf{Z}_t | \mathbf{Z}_{t-1}). \quad \text{Equation 3-9}$$

(Informally, the future is independent of the past given the present). The first order Markov property reduces the number of dependencies, making it computationally feasible to construct models with larger numbers of timeslices.

As Murphy [144] points out, any DBN that is not 1st-order Markov can be changed to make it so by changing the state space appropriately. For example, suppose \mathbf{X}_t is a 2nd order Markov DBN with states x , y and z at times t , $t-1$ and $t-2$ respectively. The probability distribution at time t is: $P(x|y,z)$. We can define a new DBN, \mathbf{Y}_t , over the state space (x, y) , with probability distribution:

$$P((x, y) | (w, z)) = \delta_{wy} P(x | y, z),$$

where w is a dummy holder for any possible state at time $t-1$, and δ is the Kronecker delta.

Nodes that contain links between two timeslices are referred to as *link nodes*.

Figure 3-4 shows the different types of inference in a DBN (this diagram is based on similar diagram provided by Murphy [144]). In *Filtering* we have evidence for all observed nodes up to time t and wish to estimate the hidden state at time t : $P(X_t | Y_{1..t})$.

The process is referred to as “filtering” because of its use in signal processing where we are attempting to determine a true, hidden, signal from a noisy one. The alternative name for this type of inference is *Monitoring* and is perhaps more appropriate for our models.

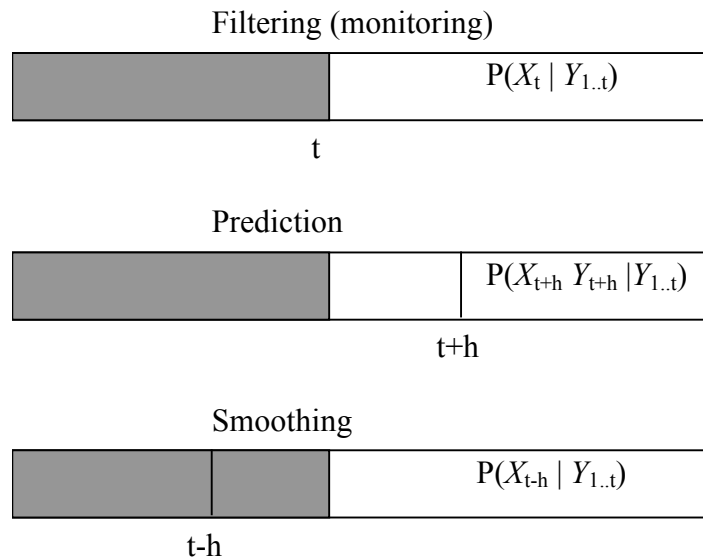


Figure 3-4 Types of inference in DBNs

We perform *Prediction* when we have evidence up to time t but wish to infer either the hidden or observed states at time $t + h$. Smoothing takes place when evidence up to time t is used to infer the most likely state at some point in the past.

The simplest way to perform inference in a DBN is to “unroll” the timeslices and use an exact inference algorithm such as the JTA. However this tends to result in large clique sizes which are repeated for each timeslice.

Several alternatives to the JTA have been developed for exact inference in DBNs. These include the Forwards-backwards algorithm, the frontier algorithm and the interface algorithm. These are discussed in detail by Murphy [144]. All of the algorithms take advantage of the Markov condition which allow information from previous timeslices to be marginalised once the present has been added to the JPD.

Exact inference in DBNs tends to be slow and memory hungry. Consequently, various approximate inference algorithms have also been developed. These can be split into stochastic (i.e. sampling) algorithms and deterministic algorithms. The latter include the Boyen-Koller (BK) algorithm, the factored-frontier algorithm and loopy belief propagation. Again, Murphy provides a detailed description [144]. With each

deterministic algorithm, a similar procedure to the exact inference algorithms mentioned above is used, but with different simplifying assumptions.

In the sections that follow we briefly describe some of these algorithms.

3.2.1 The Frontier Algorithm and the Interface algorithm

The Frontier Algorithm [224] (FA) begins by computing the JPD for the first timeslice \mathbf{Z}_0 . Child nodes in \mathbf{Z}_1 are then added to the JPD one by one (where “adding” means multiplying a node’s CPD into the frontier). When all children of a given node have been added, the parent can be marginalised. The set of nodes in the JPD at any given time is called the *frontier*. This process continues until all of \mathbf{Z}_0 has been removed and all of \mathbf{Z}_1 has been added. The same process is then repeated from \mathbf{Z}_i to \mathbf{Z}_{i+1} until all timeslices have been computed.

The preceding paragraph describes the *forward* pass of the algorithm. If we are only interested in filtering or prediction then the forward pass is sufficient. For smoothing, a similar procedure is repeated in reverse, the *backward* pass. There we begin with the JPD of \mathbf{Z}_t and calculate \mathbf{Z}_{t-1} by adding and removing nodes in precisely the reverse order. In this case we “add” a node by expanding the domain of the frontier to contain all the states of the node being added. A node is “removed” by multiplying its CPD into the frontier and then marginalising. (The multiplication in the backwards pass occurs when a node N is “removed” since it can only take place once the domain of the frontier has been expanded to contain $\text{pa}(N)$.)

The “frontier” in the FA is always at least the size of a timeslice (in terms of number of nodes) and is often larger. For the types of BN involved in software process models this results in JPDs which are too large to be currently practical.

The FA works because at every stage the frontier d-separates the past from the future. Murphy [144] pointed out that it is not necessary to have the whole frontier to perform this d-separation. The outgoing link nodes from one timeslice to another is sufficient to d-separate the past from the future. This observation forms the basis of the *Interface Algorithm* (IA). The IA uses a modified version of the JTA to ensure that the outgoing link nodes from one timeslice are all part of the same clique. Similarly, the JTA is modified to ensure that the incoming link nodes are also included in the same clique. We illustrate this with the help of the example DBN shown in Figure 3-5. This shows \mathbf{Z}_0 and \mathbf{Z}_1 for a small, four node DBN. The links between the two timeslices are shown as dashed lines.

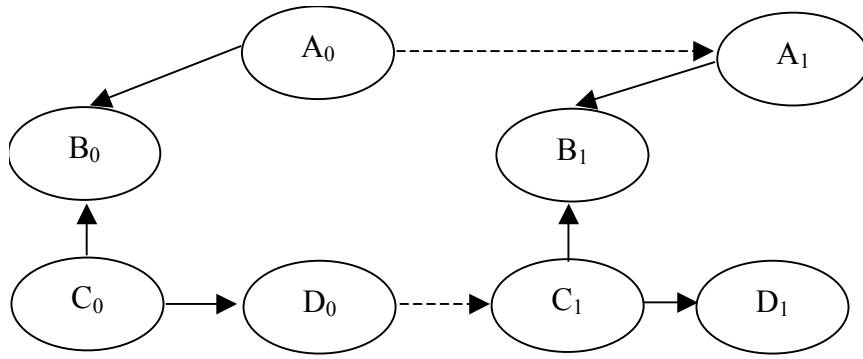


Figure 3-5 Example DBN to illustrate the Interface Algorithm

The diagram (Figure 3-5) shows a two timeslice BN (2TBN). If we restrict ourselves to the initial timeslice Z_0 and run the JTA on it, the algorithm will create a junction tree with two cliques $\underline{A_0B_0C_0}$ and $C_0\underline{D_0}$, where an underscore signifies the base clique for a node. The IA algorithm modifies this as follows.

1. During triangulation, add undirected links between all outgoing nodes. In the example, we force a link between A_0 and D_0 . The resulting junction tree has cliques $\underline{A_0B_0C_0}$ and $A_0C_0\underline{D_0}$ (see Figure 3-6).

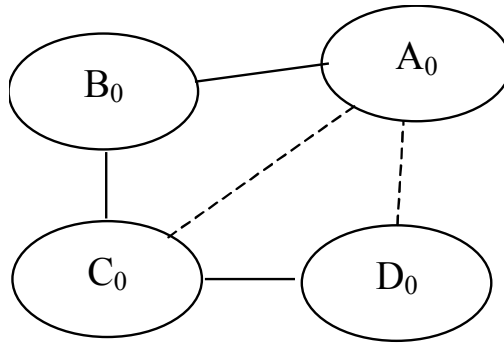


Figure 3-6 Modified triangulated graph for Z_0

2. Collect to $A_0C_0\underline{D_0}$. Denote the resulting potential by $P_0(A_0, C_0, D_0)$. Note that at this stage, no distribution from $A_0C_0\underline{D_0}$ is performed.
3. Marginalise C_0 from $P_0(A_0, C_0, D_0)$ to create $P_0(A_0, D_0)$.

Inference is carried forward to Z_1 as follows.

1. A new BN is created consisting of the output nodes of \mathbf{Z}_0 , together with all of the nodes in \mathbf{Z}_1 . This contains more than a single timeslice, but less than a 2TBN, so it is often called a 1.5TBN.
2. Link the incoming nodes (A_0 and D_0) in the 1.5TBN so that they appear in the same clique, and link the outgoing links (A_1 and D_1) so that they also share a clique. The result, with dashed lines showing undirected links and including the normal moralisation that results in the parents of B_1 being linked, is shown in Figure 3-7. The cliques are $\underline{A_0D_0A_1C_1}$, $A_1\underline{B_1C_1}$ and $A_1C_1\underline{D_1}$.
3. Multiply the appropriate CPDs into the cliques in the 1.5TBN, except for the following. Do *not* multiply the potentials for A_0 and D_0 into $\underline{A_0D_0A_1C_1}$. Instead, multiply $P(A_0, D_0)$ into $\underline{A_0D_0A_1C_1}$. The JPD $P_0(A_0, D_0)$ contains all the information that we need from \mathbf{Z}_0 in order to do consistent forward propagation in the 1.5TBN.
4. Collect to $A_1C_1\underline{D_1}$. Again, no distribution takes place at this stage.

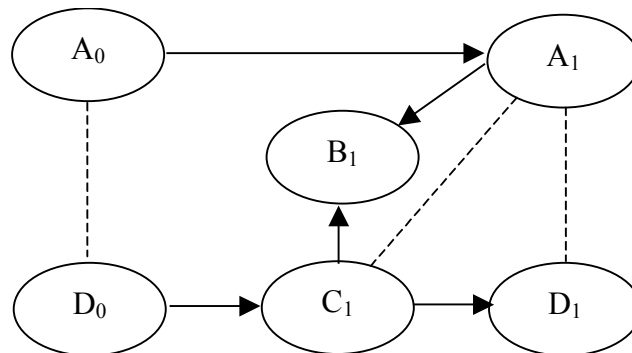


Figure 3-7 The triangulated 1.5TBN

If we had more than two timeslices then the same 1.5 DBN could be used to propagate from each timeslice \mathbf{Z}_t to its successor \mathbf{Z}_{t+1} , but since no distribution is performed on each timeslice, only the clique $A_{t+1}C_{t+1}\underline{D}_{t+1}$ contains a fully informed distribution. When the forward pass on all timeslices is complete, the IA performs a backwards pass. In the example in Figure 3-5 this consists of the following.

1. Distribute from $A_1C_1\underline{D_1}$ to the remainder of the \mathbf{Z}_1 1.5TBN.
2. Marginalise A_1C_1 from $\underline{A_0D_0A_1C_1}$ in the \mathbf{Z}_1 1.5TBN to create $P_1(A_0, D_0)$.
3. Modify the potential of $A_0C_0\underline{D_0}$ in \mathbf{Z}_0 as follows:

$$P_1(A_0, C_0, D_0) = P_0(A_0, C_0, D_0) \times \frac{P_1(A_0, D_0)}{P_0(A_0, D_0)}$$

4. Distribute from $A_0C_0\underline{D}_0$ to the remainder of \mathbf{Z}_0 .

If we are only interested in filtering or prediction then we can modify the IA to include only a forward pass. In our example this would involve the following steps.

1. Create \mathbf{Z}_0 with the additional link between A_0 and D_0 as before.
2. Collect to $A_0C_0\underline{D}_0$ and then distribute from $A_0C_0\underline{D}_0$ to the rest of \mathbf{Z}_0 . All cliques in are now consistent with all information in \mathbf{Z}_0 .
3. Marginalise $A_0C_0\underline{D}_0$ to create $P(A_0, D_0)$. There is no need to distinguish between the forward and backward versions of this potential now since there will be no backwards pass.
4. Create the 1.5TBN and add $P(A_0, D_0)$ as before.
5. Collect to $A_1C_1\underline{D}_1$ and then distribute to \mathbf{Z}_1 . All cliques in the 1.5TBN are now consistent with all information in \mathbf{Z}_0 and \mathbf{Z}_1 .

I will refer to this modified version of the IA as the *Forward Only Interface Algorithm* (FOIA).

3.2.2 The Boyen-Koller Algorithm

The IA links together all the interface nodes in a timeslice into a single clique. This creates larger cliques than are normally created by the JTA. If there are a large number of nodes in the interface, or the nodes represent continuous distributions, then the cliques can become intractable. The Boyen-Koller algorithm (BKA) [26] addresses this by representing the interface clique as the product of a set of smaller cliques.

Clearly any correlation between interface nodes will be lost if the nodes are in distinct cliques. Where this correlation is significant, the BKA only provides approximate inference. What is remarkable about the BKA is that, for a wide range of models, the error in the approximation does not grow across timeslices. This is due to the fact that stochastic processes “mix” states when going from one timeslice to another. As Boyen and Koller point out, two distributions ϕ_t and ψ_t over timeslice \mathbf{Z}_t might have no overlap. However, after the transition to \mathbf{Z}_{t+1} , and providing the

transition process involves at least some minimal mixing of states, ϕ_{t+1} and ψ_{t+1} are likely to share some overlap in \mathbf{Z}_{t+1} .

In the case where ϕ_t is an estimate for ψ_t , the two distributions are likely to share a considerable overlap to begin with. If the transition to \mathbf{Z}_{t+1} is approximate, as is the case with BKA, then the two distributions tend to grow apart. However this is balanced by the stochastic nature of the process, which tends to bring the two distributions back together again. Conditioning, due to evidence in \mathbf{Z}_{t+1} , further tends to compensate for the errors due to approximate inference.

The IA only has a single interface clique. As such it is sufficient to collect to this clique during forward propagation and to distribute from it during backwards propagation. BKA has multiple interface cliques which must be updated. It must therefore run a full collect and distribute cycle on each timeslice during both the forwards and the backwards passes.

An extreme implementation of BKA assigns a separate clique to each interface node. Copying cliques between timeslices is then the equivalent of copying the marginals of the interface nodes across timeslices. Murphy [144] calls this the *fully factored* BKA (FFBKA). A forward only FFBKA is the default algorithm employed by AgenaRisk [8]. As we shall see shortly, not only is this adequate for the DBN models developed in this thesis, it actually has several advantages over a full forwards-backwards pass algorithm.

3.2.3 Parameter Learning in DBNs

Suppose we have a random variable Y , which we have reason to believe is normally distributed. We also happen to know that the parameters governing Y , the mean m and variance s , vary depending on the environment. We want to learn the values of m and s using observations of the variable Y . If we were to use a BN to learn these values then we could use a model similar to the one in Figure 3-8.

Each of the y_i nodes has the same prior: Normal (m, s^2). Evidence is entered as one value per y_i node. The prior of m is set to be uniform across its expected range. The prior of s is set to an inverse gamma distribution (see for example [30] or [69] for an explanation of this choice). The posteriors of m and s give the most likely values of the mean and variance given the data.

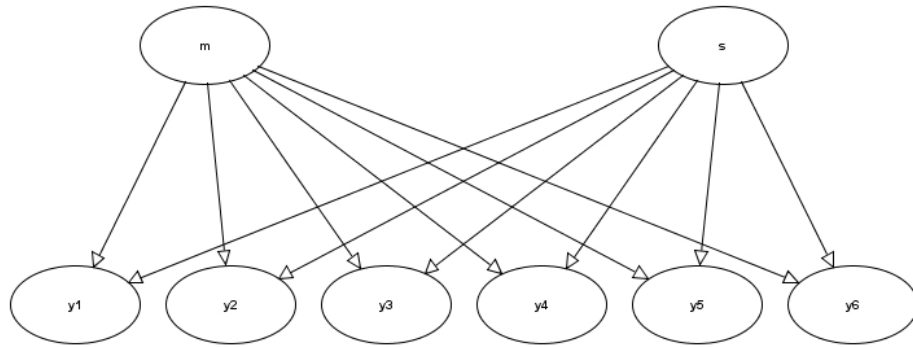


Figure 3-8 Learning the parameters of a Normal distribution.

The problem with this type of model is its potential size in memory. The size of the junction tree grows linearly with the number of observations. Each new evidence node, y_i , results in a new clique $\{m, s, y_i\}$. If the nodes are continuous and discretised to say, 100 states, then there are 10^6 states in each clique. This is a relatively simple model. When each observation node is replaced by a more complex subnet, the cliques can be much larger. If we had a dynamic model then we could add observations one at a time, enabling us to incorporate an indefinite number of observations. The theorem which follows allows us to do this. Before embarking on the set of formal definitions necessary to prove the theorem, it is worth looking at a very simple example which shows the essence of the proof.

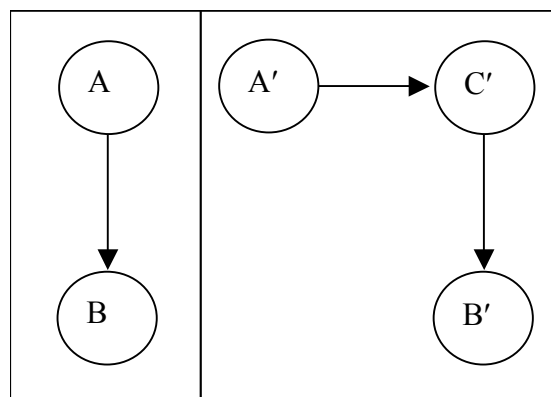


Figure 3-9 Two similar BNs

Figure 3-9 shows two similar BNs. Let's assume that each node is a boolean node and the NPTs are set up as follows.

1. $P(A') = P(A)$ A' and A have the same priors
2. $P(B'|C') = P(B|A)$ B' has the same dependency on C' as B has on A

3. $P(C'|A') = \delta_c^a$ The CPD of C' is just the identity matrix.

Where c and a are the states of C and A respectively. The JPD of the right hand BN is:

$$P(A', B', C') = P(B'|C')P(C'|A')P(A').$$

If we marginalise C' and look at $P(A'=true, B'=true)$ we get:

$$\begin{aligned} P(A'=true, B'=true) &= P(B'=true | C'=true)P(C'=true | A'=true)P(A'=true) \\ &+ P(B'=true | C'=false)P(C'=false|A'=true)P(A'=true) \end{aligned}$$

But we know that $P(C'=true | A'=true) = 1$ and $P(C'=false|A'=true) = 0$. We also know that $P(B'=true | C'=true) = P(B=true | A=true)$ and that A' and A have the same priors, so we get:

$$P(A'=true, B'=true) = P(B=true | A=true)P(A=true)$$

This is identical to the corresponding JPD entry of the left hand BN. since the JPDs are the same, the marginals for the learned parameter A will also be identical in both BNs. If we have evidence in B then it simply means that some of the JPD entries get zeroed and a renormalizing constant has to be introduced. So evidence entered into B on either BN has the same effect on A .

Definition 3-1

Let A be a node with no parents in a BN. The node A is called a *root node*. If A is never instantiated with evidence then A is called a *root learning node*.

Definition 3-2

Let $\{Z_i, i = 1..n\}$ be a DBN. Let $A_1 \in Z_1$ be a root learning node. Let $A_i \in Z_i, i = 2..n, \pi(A_i) = \{A_{i-1}\}, P(A_i | A_{i-1}) = \delta_c^d$, where δ is the Kronecker delta, c is a state in A_i and d is a state in A_{i-1} . A_i is called a *proxy learning node* of A_1 .

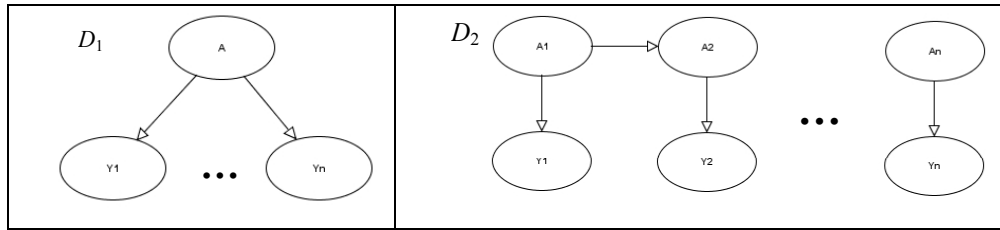


Figure 3-10 Two alternative learning models

In Figure 3-10 (D_1), A is a root learning node. Figure 3-10 (D_2), A_1 is a root learning node and A_2 through to A_n are proxy learning nodes (provided the transition matrix is the identity matrix).

Lemma 3-1

All proxy learning nodes of a root learning node have the same marginal probability distribution as the root learning node.

Proof

This follows trivially from the definition of conditional probability and the identity transition matrix.

$$\begin{aligned}
 P(A_{i+1} = \alpha) &= \sum_{A_i} P(A_{i+1} = \alpha \mid A_i = \beta) P(A_i = \beta) \\
 &= \sum_{A_i} \delta_{\alpha}^{\beta} P(A_i = \beta) = P(A_i = \alpha)
 \end{aligned}$$

Induction from A_1 proves the lemma. QED

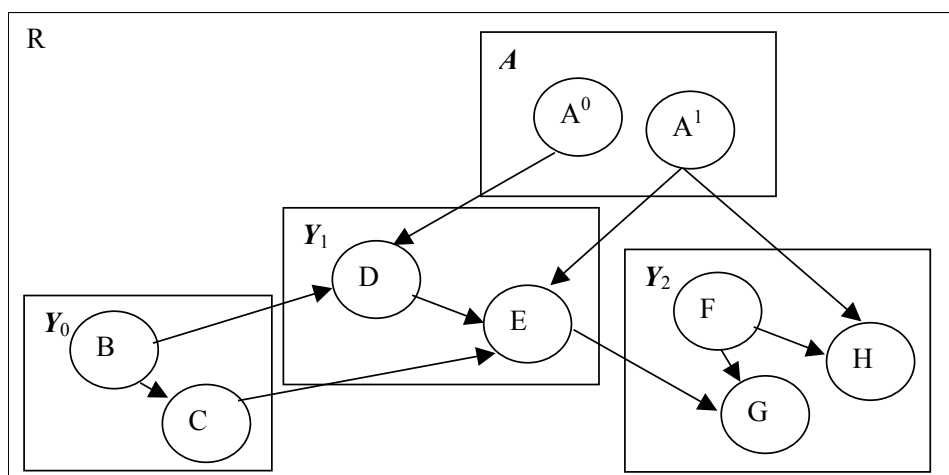


Figure 3-11 Example of evidence subnets

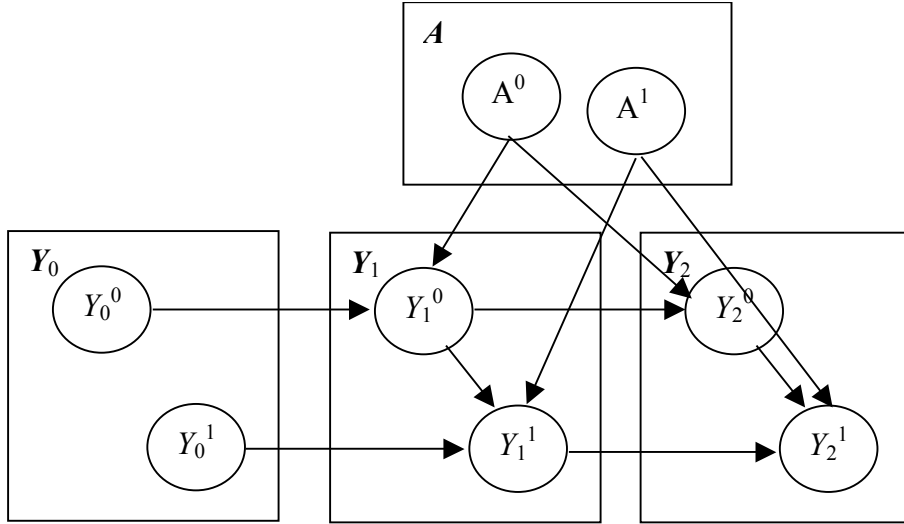


Figure 3-12 Example of regular evidence subnets

Definition 3-3 (See Figure 3-11)

Let $D = (\mathbf{R}, \mathbf{E})$ be a DAG with nodes \mathbf{R} and edges \mathbf{E} .

Let $\mathbf{A} = \{A^i, i = 0 \dots n\} \subset \mathbf{R}$, be the set of root learning nodes of \mathbf{R} .

Let $\{Y_k \subset \mathbf{R}, k = 0 \dots m\}$ be a set of subnets that span \mathbf{R}/\mathbf{A} , i.e. $\mathbf{R}/\mathbf{A} = \cup \{Y_k, i = 0 \dots m\}$, such that $\pi(Y) \subset Y_0, \forall Y \in Y_0$, and $\pi(Y) \in \mathbf{A} \cup Y_k \cup Y_{k-1}, \forall Y \in Y_k$ when $k = 1 \dots m$. We call Y_0 the *prior subnet* and $\{Y_k\}$ the *evidence subnets* of D .

Definition 3-4 (See Figure 3-12)

Let $D_1 = (\mathbf{R}, \mathbf{E})$ be a DAG with nodes \mathbf{R} and edges \mathbf{E} , having root learning nodes \mathbf{A} and evidence subnets $\{Y_k\}$. If each Y_k consists of a set of nodes $\{Y_k^i \subset \mathbf{R}, i = 1 \dots p\}$ and $P(Y_k^i | \pi(Y_k^i)) = P(Y_j^i | \pi(Y_j^i)), k \neq j, k \neq 0, j \neq 0$, then $\{Y_k\}$ are called *regular evidence subnets*.

Definition 3-5

Let $D_1 = (\mathbf{R}, \mathbf{E})$ be a DAG with nodes \mathbf{R} and edges \mathbf{E} , having root learning nodes \mathbf{A} and regular evidence subnets $\{Y_k, i = 0 \dots m\}$. Let D_2 be a DBN with timeslices $\{Z_t, t = 0 \dots m\}$ constructed as follows:

1. $Z_0 = Y_0 \cup \mathbf{A}$,
2. $Z_i = Y_i \cup \mathbf{A}_i, i = 1 \dots m$, where \mathbf{A}_i are proxy learning nodes of \mathbf{A} ,
3. $\pi(Z_k^i) = \{\pi(Y_k^i) / \mathbf{A}\} \cup \mathbf{A}_i, \forall Z_k^i \in Z_k$,
4. $P(Z_k^i | \mathbf{A}_i) = P(Y_k^i | \mathbf{A})$.

We call D_2 the *derived DBN* of D_1 .

In Figure 3-10, D_1 is a simple example of a BN consisting of root learning nodes and regular evidence subnets. In this case $Y_0 = \phi$ and there are no links between evidence subnets. D_2 is the derived DBN of D_1 .

Theorem 3-1

Let D_1 be a BN consisting of root learning nodes $A = \{A^j, j = 0 \dots n\}$ and regular evidence subnets $\{Y_k, k = 0 \dots m\}$. Let D_2 be the derived DBN of D_1 . with proxy learning nodes A_k^j and timeslices $\{Z_k\}$. Then $P(A_k^j) = P(A^j)$.

Proof

We have already shown via Lemma 1, that the proxy learning nodes in D_2 have the same marginal distribution as their corresponding root learning nodes in D_2 . What remains to be shown is that the marginal distributions of the root learning nodes in D_2 have the same marginal distributions as the root learning nodes in D_1 .

Let $\pi_k(Y_k^j) = \pi(Y_k^j) / A_k$. The JPD J_1 of the DAG D_1 is given by:

$$J_1 = P(Y_0^j | \pi(Y_0^j)) P(A^0) \dots P(A^n) \prod_{k=1}^m P(Y_k^j | \pi_k(Y_k^j), A^0 \dots A^n), \forall Y_k^j \in Y_k$$

The JPD J_2 of the DAG D_2 is given by:

$$J_2 = P(Y_0^j | \pi(Y_0^j)) P(A^0) \dots P(A^n) \prod_{k=1}^m P(Y_k^j | \pi_k(Y_k^j), A_k^0 \dots A_k^n) P(A_k^0 | A_{k-1}^0) \dots P(A_k^n | A_{k-1}^n)$$

We can examine the contribution to this expression from Z_m . Call this W_m^m

$$W_m^m = \prod_j P(Y_m^j | \pi_m(Y_m^j), A_m^0 \dots A_m^n) P(A_m^0 | A_{m-1}^0) \dots P(A_m^n | A_{m-1}^n)$$

Marginalising the proxy learning nodes A_m^0 through A_m^n gives:

$$\sum_{A_m} W_m^m = \sum_{A_m} \prod_j P(Y_m^j | \pi_m(Y_m^j), A_m^0 \dots A_m^n) P(A_m^0 | A_{m-1}^0) \dots P(A_m^n | A_{m-1}^n).$$

But each of the transition matrices $P(A_m^0 | A_{m-1}^0)$ through $P(A_m^n | A_{m-1}^n)$ is just the identity matrix. They only have non-zero values when $A_m^i = A_{m-1}^i$, so the above expression reduces to:

$$W_m^{m-1} \equiv \sum_{A_m} W_m^m = \prod_j P(Y_m^j | \pi_m(Y_m^j), A_{m-1}^0 \dots A_{m-1}^n).$$

Placing this back into our expression for J_2 and adopting similar definitions for W_k^k gives:

$$\sum_{A_m} J_2 = P(Y_0^j | \pi(Y_0^j))P(A^0) \dots P(A^n) W_m^{m-1} \prod_{k=1}^{m-1} W_k^k$$

We can now marginalise the proxy learning nodes A_{m-1}^0 through A_{m-1}^n . This gives:

$$\sum_{A_m A_{m-1}} J_2 = P(Y_0^j | \pi(Y_0^j))P(A^0) \dots P(A^n) W_m^{m-2} W_{m-1}^{m-2} \prod_{k=1}^{m-2} W_k^k$$

Proceeding inductively, we eventually arrive at:

$$\sum_{A_1 \dots A_m} J_2 = P(Y_0^j | \pi(Y_0^j))P(A^0) \dots P(A^n) \prod_{k=1}^m W_k^0$$

which is identical to the expression for J_1 . Further marginalising down to any required root learning node shows that their marginal distributions are identical in both nets. QED

The proof of Theorem 3-1 is almost identical when evidence is present except that the evidence selection vectors and suitable normalisation constants must be included.

In AgenaRisk, we can create link nodes with a transition matrix equal to the identity matrix simply by setting the CPD for A_i equal to the arithmetic expression: A_{i-1} . To see this, recall our discussion of deterministic functions in section 3.1.3. Let us take a simple example where the A_i each have three interval states: 5-10, 10-15 and 15-20. The rule when evaluating expressions is to find the minimum and maximum values for the expression for each combination of states of parents, and then create a uniform distribution between those limits as shown in Table 3-1.

A_{i-1}	5-10	10-15	15-20
min (A_i)	5	10	15
max (A_i)	10	15	20

Table 3-1 Evaluating the expression $A_n = A_{n-1}$

In this example, a uniform distribution gets created which fills exactly the same state as the selected parent state. We therefore have a probability mass of one for the state corresponding to the input parent state, and zero for all other states. This therefore gives us the desired transition matrix.

The combination of theorem 3-1 and the AgenaRisk expression evaluation mechanism, therefore allows us to build dynamic learning models of any desired size in AgenaRisk.

3.2.4 “Forward only” Learning in DBNs

We have seen how it is possible to perform parameter learning in DBNs, where the learned parameter can be informed by an arbitrarily large number of observations. Implicit in this is the assumption that, whatever algorithm is used, all observations inform all timeslices. This is not always required, and indeed can complicate the model unnecessarily.

Consider the DBN version of the model shown in Figure 3-10. If we are only interested in filtering then it makes no difference whether we perform the backwards pass or not. In both cases, at time t the posterior for A_t will be:

$$P(A_t | A_{1..t-1}, Y_{1..t}) = P(A_t | A_{t-1}, Y_t)$$

(by the Markov property). Similarly, if we are interested in prediction at time $t + n$, the forward only pass is sufficient to enable us to compute $P(A_{t+n} | A_t, Y_t)$.

There are advantages to performing forward only learning, the most obvious being that the algorithm can complete in approximately half the time since the backward pass is omitted. Another advantage is that forward only learning provides a “history” of the learned parameters. We can show this by returning to the DBN model in Figure 3-10 and running a full forward and backwards algorithm. If we run the forwards-backwards algorithm with $n = t$, then each of the A_i nodes will be conditioned using all of the Y_i nodes:

$$P(A_i) = P(A_i | Y_{1..t}).$$

In the case where the transition is the identity matrix, they all end up equal. We then advance time and add a new timeslice corresponding to $n = t + 1$. Again, the forwards-backwards algorithm will condition each of the A_i 's using all of the Y_i 's and all of the A_i 's will end up equal. However, the value of the A_i 's at time $t + 1$ will be different from the value at time t due to the extra conditioning provided by evidence on node Y_{t+1} .

Now consider the same example, but this time using forward only learning. The addition of evidence from Y_{t+1} affects only A_{t+1} . The previous values of the learned parameter remain unaffected. Each A_i continues to be conditioned only on the evidence up to its own point in time: $P(A_i) = P(A_i | Y_{1..i})$. This enables us, for example, to extract the mean and variance of each A_i and show how it changes as new evidence is provided. Being able to track the changing value of a learned parameter over time can be useful for descriptive and diagnostic purposes (as will become apparent when we construct a DBN model of an agile software development environment).

3.3 Summary

This chapter has covered the various desirable properties of BNs. It has provided an introduction to some of the theory needed to understand propagation and learning in BNs and DBNs. In the next chapter we will see how this can be applied to building software process models.

4 Existing Models

This chapter examines existing BN software process models. It begins with a quick reminder of the principle advantages of BNs with respect to software process modelling. We then cover some of the BN models developed by Fenton, Neil and Krause and finish with an overview of some of the other BN software process models that have been developed.

Novel contributions include:

- an analysis of the ISBSG data set [90] with regard to KLOC per FP for Java and C++,
- a description of some attributes of the “Phase” model that may make it suitable as a starting point for an XP model,
- a proposed hierarchy of technical complexity (via Appendix G),
- an analysis of the result of the “Philips” model,
- a review of existing BN models of the software development process.

4.1 *Introduction to Bayesian Net Software Models*

In [64], Fenton and Neil explain the rationale behind creating causal models of the software development process using BNs. Much of this argument has already been outlined in the previous chapters and in Appendix A. It can be summarised as follows.

1. In BNs, cause and effect relationships between the elements are stated explicitly in a way that is naturally understood by domain experts.
2. They can incorporate many of the empirical findings of previous regression based models.
3. Both inductive reasoning (likely effect resulting from one or more causes) and abductive reasoning (determining the likely cause of a perceived effect) are possible.
4. Where evidence is available, either in the form of an observation or a probability distribution, this can be propagated in a systematic fashion throughout the model.
5. Where no evidence is available, prior conditional probabilities can be used to make predictions.
6. Uncertainty is quantifiably built into all aspects of the model.

7. Setting desired outcomes and observing the distributions of causal factors allows us to perform trade off analysis.

Fenton, Krause, Neil and others have gone on to develop a series of BN models, culminating in the AID tool [152], the MODIST models [63], and the extensive trials of revised models at Philips [151]. Those models were used to provide improved methods of risk assessment for project managers, with special emphasis on defect predictions and effort prediction. A similar model has been developed by Siemens [206].

Several other groups have also researched the use of BN based software process models. Researchers at Motorola have followed Fenton, Krause and Neil’s lead in creating defect prediction models: [135] and [76]. Wooff, Goldstein, and Coolen [219] have developed BNs modeling the software test process while Bibi and Stamelos [22] have shown how BNs can be constructed to model IBM’s Rational Unified Process. These models are discussed in more detail in the sections that follow.

4.2 MODIST Project Model

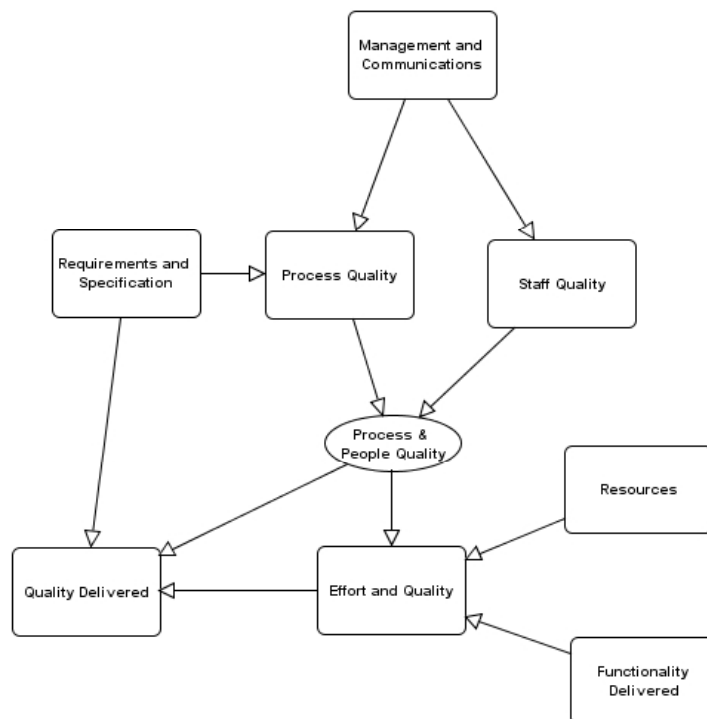


Figure 4-1 The Project Level model – high level overview

A high level overview of the MODIST Project Level model [7] is shown in Figure 4-1. It includes the main competing elements in all software projects: scope, cost, timescale and quality. The aim of the model is to allow project managers to perform various forms of trade off analysis and risk assessment. For example, the amount of functionality to be delivered can be fixed and the model allowed to predict the effort and overall timescale. Alternatively, the timescale can be fixed and the impact on functionality and quality can be predicted.

Each of the boxes shown in Figure 4-1 is actually a subnet consisting of between 2 and 12 nodes, with the whole model containing 49 nodes. However, many of these nodes are functions of other nodes. Only about half of them are generally input by the user and as always with BN models, where information is not available, default distributions are used.

We will not examine the full model in detail. Instead we shall examine a selected set of subnets, in varying levels of detail, in order to examine some of the techniques used.

4.2.1 Management and Communication Subnet

This subnet models the size of a project team, its geographical dispersal, the amount of work that has been subcontracted and the quality of the subcontract work.

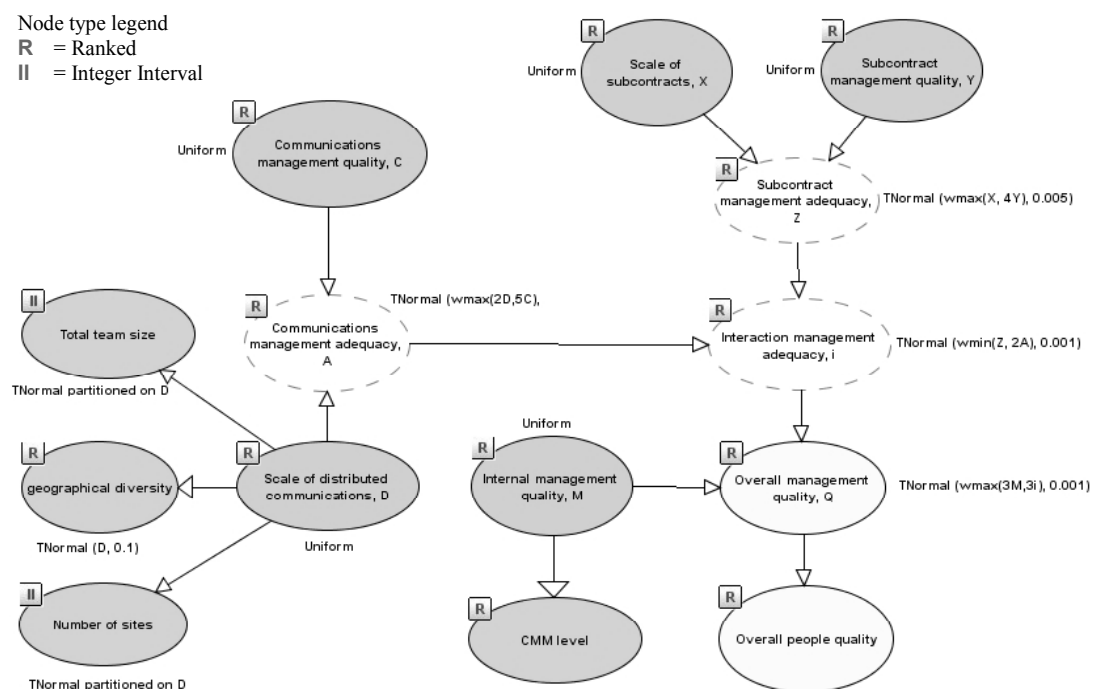


Figure 4-2 Management and communication subnet

In Figure 4-2 we see examples of the TNormal, wmin and wmax expressions discussed in section 3.1.2. We also see the use of *partitioned expressions*, discussed in the Heterogeneous Modelling section of Appendix A. Appendix A only covered continuous nodes that are children of ordinal or nominal parents. The examples here show integer interval nodes as the children. The algorithm is analogous to that for continuous nodes except that a rounding stage is also required.

Ranked nodes are used extensively by this subnet, and indeed throughout the entire model. The prevalence of ranked nodes is indicative of the scale of expert judgement required in this model. This is because part of the aim of this type of model is to quantify aspects of software development which are often omitted due to lack of metrics.

The *Scale of distributed communications* node captures the extent to which a project is developed across multiple sites. Managers would not be expected to enter a value directly into this node. Instead, they would use some of the indicator nodes provided.

4.2.2 Resources Subnet

This small but important subnet is shown in Figure 4-3.

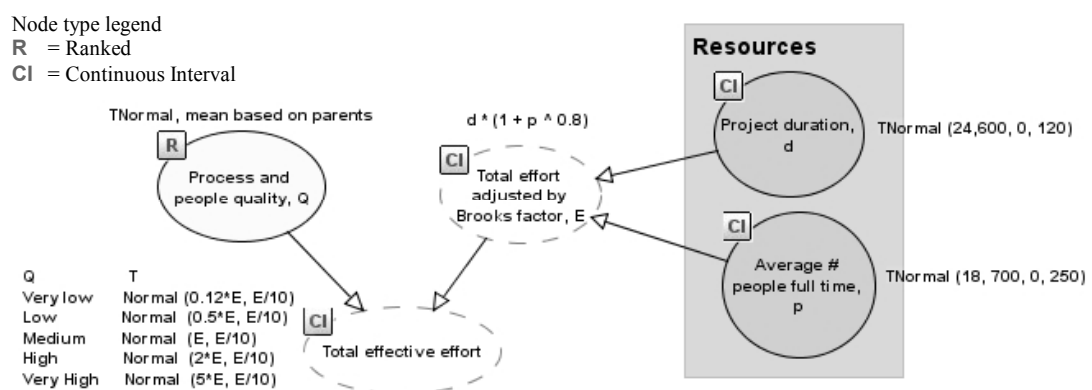


Figure 4-3 Effort subnet

The exponent in *Total effort adjusted by the Brooks factor*, 0.8, is the Brooks factor [31] that reduces the marginal contribution of each additional team member due to the communications and training overhead.

The *Total effective effort* node is an example of a partitioned expression, where each ranked value of *Process and people quality* gives rise to a different NPT distribution across the *Total effort adjusted by the Brooks factor*.

An interesting question is whether the Brooks factor is changed by XP. i.e. Does the introduction of pair programming, regular rotation of responsibilities and collective ownership, increase the effectiveness of additional personnel? According to Williams et al [215] the answer is “yes”. Using 30 responses to a survey that they sent out, they constructed a system dynamics model which isolated the mentoring overhead of assimilating new staff. Their model indicated that pair rotation and pairing of experienced with novice staff, could substantially reduce the time taken for new team members to become productive.

4.2.3 Functionality Delivered Subnet

This subnet models the project size. It is shown in Figure 4-4. As can be seen, the subnet relates functionality to effort.

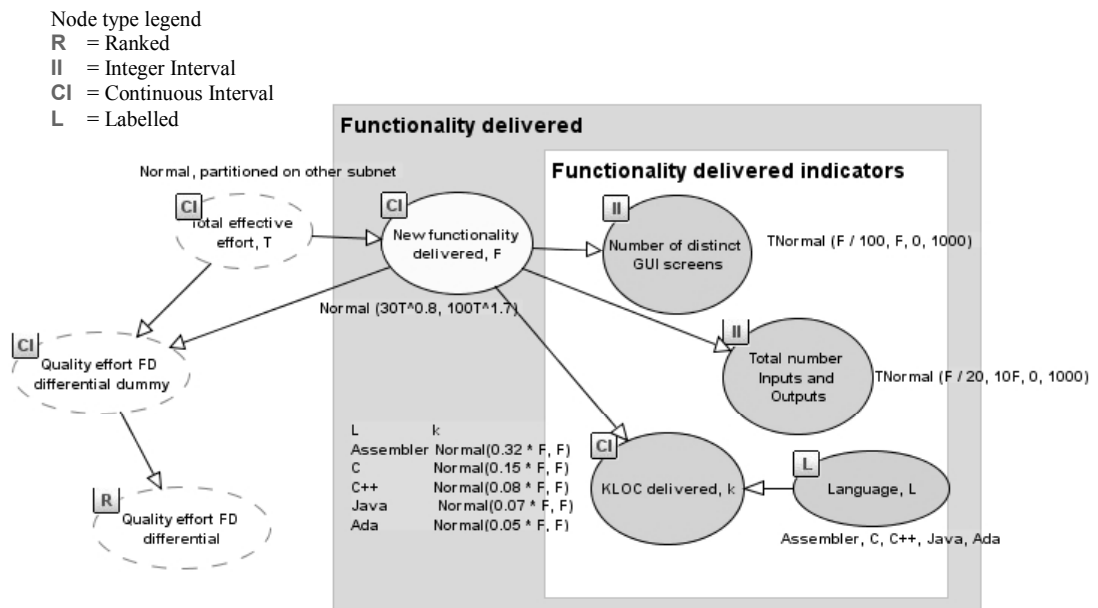


Figure 4-4 Functionality delivered

F is measured in Function Points [11]. The exponent relating effort T to mean functionality F is taken from the COCOMO model [24], where the inverse relationship is defined:

$$E = aK^b$$

Equation 4-1

E is effort in person months, K is source code size in KLOC, and a and b are constants. The value of b is always greater than one. i.e. All projects exhibit diseconomies of scale. The default value is 1.2.

Function Points is not the only measure of program size that can be used. Programming language and KLOC can also be entered. Data provided by Jones [96][97], is used to convert from KLOC to FPs. *Language* is used to create a partition function. Each partition is a normal distribution based around F using the parameters shown in Table 4-1. Thus, a phase consisting of 100 FP would yield a distribution with a mean of $100 * 0.07 = 7$ KLOC of Java code.

Language	μ	σ^2
Assembler	0.32F	F
C	0.15F	F
C++	0.08F	F
Java	0.07F	F
Ada	0.05F	F

Table 4-1 FP to KLOC distribution parameters

Language	μ	σ^2
C	0.17	0.09
C++	0.087	0.005

Table 4-2 KLOC per FP based on ISBSG data

My own analysis of the ISBSG data [90], shown in Table 4-2, is consistent with the Capers Jones data. The similarity in the mean values is striking given that the datasets involved span two decades of software development. It should be noted that the meaning of σ^2 in the two tables is different. In Table 4-1 it is the variance for a project of a given size in function points, whereas in Table 4-2 it is the variance of the overall KLOC per FP distribution for the given programming language.

Other indicators of Function Points can be entered via the *Number of distinct GUI screens* and *Total number Inputs and Outputs* nodes.

4.3 Fenton, Krause & Neil's "Phase" Model

A high level overview of Fenton, Krause and Neil's Phase Model is shown in Figure 4-5. Its aim is to model the residual defects remaining in a delivered piece of software.

As with the Project Model, it is too large to be considered all at once. Both models lend themselves to a natural breakdown into smaller subnets. However, there is an

important distinction between the two. The Project Model only makes sense when all subnets are included. The Phase Model on the other hand was designed to reflect the principal activities in any software production process: specification, design, code and test.

The basic structure of the Phase Model is straightforward. The “Scale of new functionality” subnet determines the size of the software being delivered. This gives rise to a certain number of inherent potential defects in the “Defect insertion and recovery” subnet. The remaining subnets then determine how many of these potential defects are discovered through the specification, development and test activities.

It is not necessary to include all activities in each phase. For example, there could be development only phases, development and test phases, or requirements gathering phases. Individual phases can then be chained together, with the defects output from one phase acting as an input to a subsequent phase.

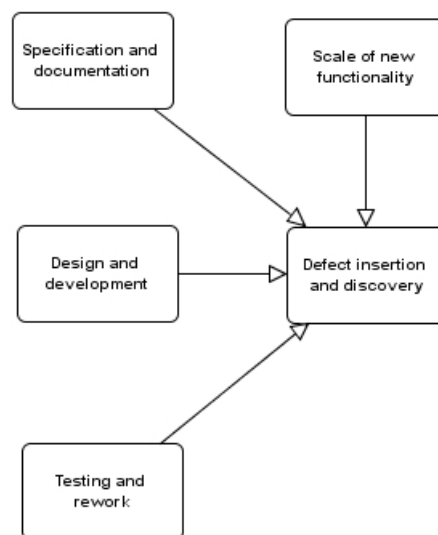


Figure 4-5 Phase model overview

This gives this model a number of attractive features when used as a basis for any XP model.

1. It readily lends itself to “objectification”. More details on an object based version of this model, as created by me, can be found in Fenton et. al. [66].
2. The iterative nature of the Phase Model fits well with the iterative nature of agile development methods in general, and XP in particular.
3. The clean separation of subnets allows the *Specification and documentation* subnet to be simplified to reflect its lack of emphasis in XP.

- The greater detail afforded to development and testing compliments XP's similar concentration on these aspects of the lifecycle.

The phase model has several points where nodes must be “linked” to connect one phase to another. It is important to set the inputs of the first phase to an appropriate value. In some cases, such as the initial defect count, this must be explicitly set to zero.

The model shown in Figure 4-5 is an example of the All Activities Phase Model: a model that includes one copy of each activity. Other development phases with a variety of activities have also been tested and validated. A description of these can also be found in [66].

Some of the subnets of the model are now discussed in detail.

4.3.1 Design and Development Subnet

This subnet covers all effort expended on both design and code. It is shown in Figure 4-6. Unless otherwise stated, all nodes are Ranked nodes.

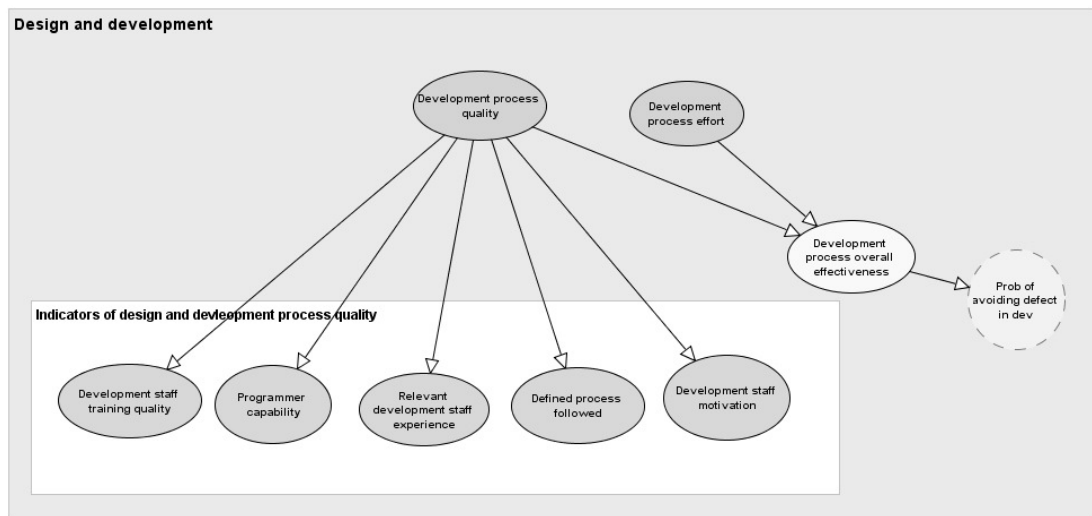


Figure 4-6 Design and development subnet

Development process quality has a manually coded, triangular, prior probability distribution. Evidence can be entered directly or updated using a set of indicator nodes. Each of these is a TNormal centred on their parent.

Development process effort has the same prior probability distribution as *Development process quality*. This is one of several cases where a Ranked node, representing an ordinal scale of measurement is used, where a ratio scale, such as person-days, might be expected. When we say it is High, we mean that it is high given the needs of this phase. This is an extremely important point as it allows a subjective

judgement to be used where a software metric might otherwise be needed. This is one of the features that allows the models to be used without an extensive data gathering phase.

It could be argued that extracting data from the project plan, or some other database of effort, would make the model more objective, but this is not the case. Effort estimates in the plan are either based on expert judgement themselves, or they are based on some model such as COCOMO, which we have already seen (Jørgensen [101]) is often no better than expert judgement. Even if the figures in the plan are based on rigorous Function Point counting then they can only be turned into reliable effort estimates if there has been an extensive metrics collection programme in similar, previous projects – precisely the thing that we wish to avoid. Even then, the results are only as good as the completeness of the specification and the model which is used to turn the FP count into an effort estimate.

4.3.2 Defect Insertion and Discovery Subnet

The Defect Insertion and Discovery subnet is the core of the model and is shown in **Figure 4-7**.

Any given amount of new functionality is associated with two different sets of defects: specification defects, which are essentially inaccurately gathered requirements, and software defects. Other subnets then predict the probability of these defects being avoided, detected and fixed. Each of these possibilities is treated as a set of Bernoulli trials on the remaining defects and therefore appears as a binomial distribution in the model. We will now discuss some of the nodes in this subnet in detail.

Defects from previous phases of the software can be included via the *Residual defects pre* node. This node should always have some evidence entered. It should be zero when this is the first phase of the software. Otherwise it should take its value from previous phases.

The *Residual defects post* node is one of the main links between phases in the Phase Model. By associating its value with *Residual defects PRE* in the next phase, we can chain development phases together to determine the cumulative number of defects remaining.

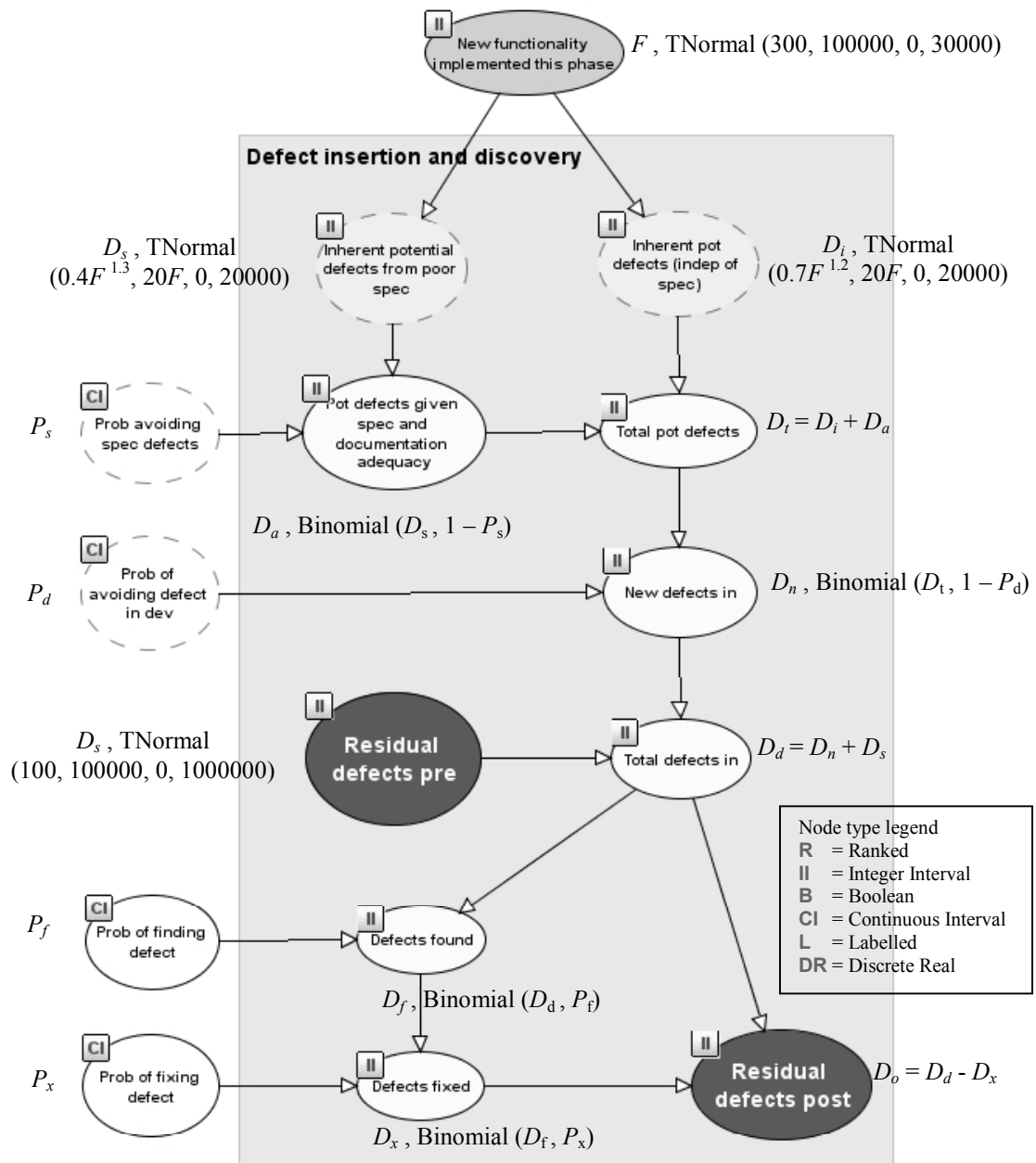


Figure 4-7 Defect insertion and discovery subnet

4.4 The Philips Model

This model was developed in collaboration with Royal Philips Electronics N.V. Netherlands, and Philips PSC Bangalore.

The high level structure of this model is very similar to the All Activities Phase Model described in the previous section. As with the Phase Model, this is a defect prediction model. The subnet, **Scale of new functionality implemented**, gives rise to a distribution of inherent potential defects. Depending on the process, development

and test quality, these defects are gradually removed, leaving *Residual defects post* as the final defect distribution.

However, there are also several significant differences between the Philips Model and the Phase Model.

1. The introduction of an **Existing code base** subnet.
2. The introduction of a **Common influences** subnet.
3. Many nodes that were indicator nodes in the previous model have become causal factors in the Philips Model.
4. A concentration on KLOC and complexity as the preferred size measure, rather than Function Points.

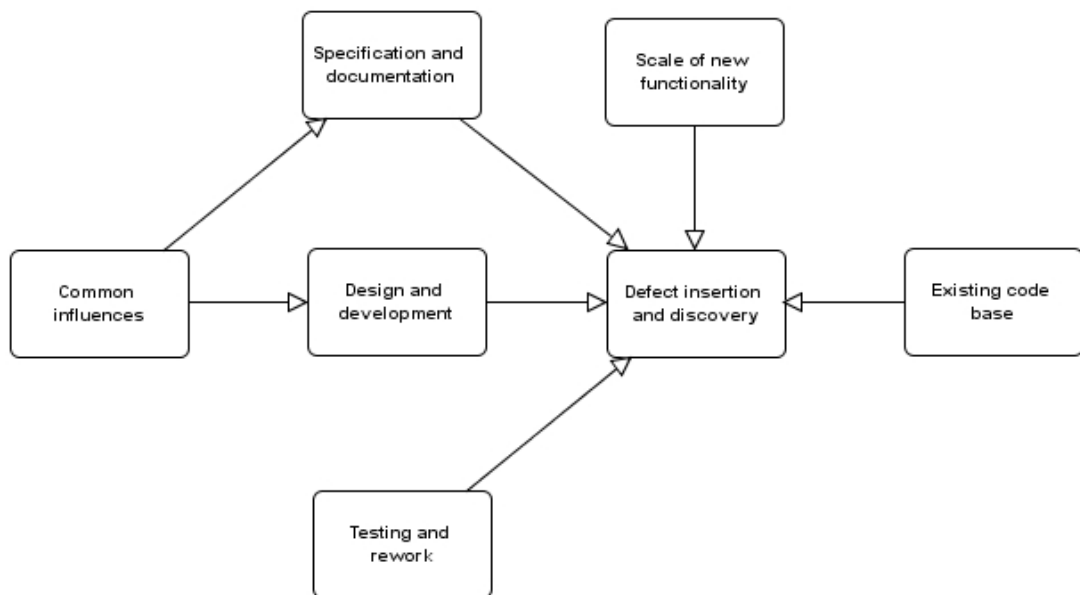


Figure 4-8 The Philips Model

4.4.1 Existing Code Base Subnet

This subnet allows the defects present in existing code to be taken into account. It consists of a highly simplified and very compact defect prediction model. It combines code size and complexity with an overall estimation of the process and test quality.

As the code already exists there is no difficulty in describing its size in terms of lines of code rather than Function Points.

The subnet contains no mention of the primary programming language used. This is because all programming at the Philips sites was in C. A more general version of this subnet should include a dependence on programming language.

KLOC existing code base has a prior distribution set to a beta distribution with k (in KLOC) normalised from the range 0 – 1000, to the range 0 – 1. The exact shape of the KLOC distribution curve will normally be of very little consequence to the model. If there is no pre-existing code base then zero will be entered in *Residual defects pre*, blocking any propagation from the **Existing code base** subnet. If there is a pre existing code base, then evidence will be entered in *KLOC existing code* making the shape of the distribution irrelevant.

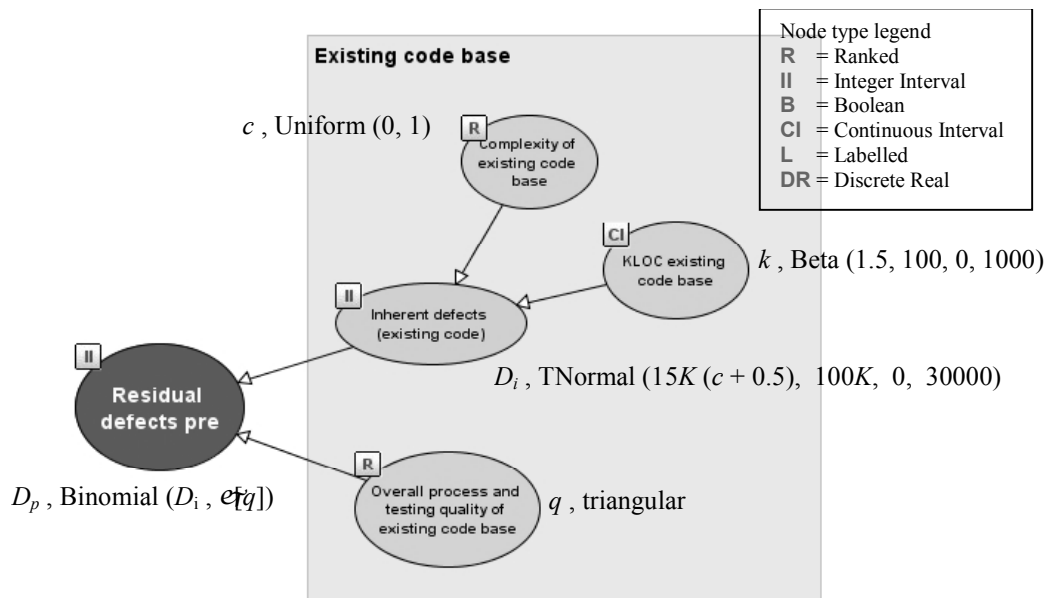


Figure 4-9 Existing code base subnet

Complexity of existing code base is a standard “Very low” to “Very high” ranked node, with a uniform distribution.

Inherent defects combines the KLOC and complexity figures into a single NPT.

The node, *Overall process and testing quality of existing code base*, is a ranked node with a manually created prior distribution that favours medium values. It determines the probability of letting *Inherent defects* through to *Residual defects pre*, by acting as the basis of a partition on the latter. Each partition is a binomial where *Inherent defects* represents the number of trials. A low process and testing quality will allow a lot of potential defects through, whereas a high process and test quality will let fewer potential defects through.

4.4.2 Indicator Nodes Become Causal Factors

We have already described some of the differences between indicator nodes and causal factors in section 3.1.1. Many of the indicator nodes in the Phase model become causal factors in the Philips model.

The causal relationship between many of the nodes in the Phase and Philips models is ambiguous. For example, consider two of the nodes shown in the BN fragments in Table 4-3: *Programmer capability* and *Development process quality*. Is *Programmer capability* an indicator of *Development process quality*, or is it a cause? In the Phase model it is an indicator, while in the Philips model it is a cause. In fact many of the indicator nodes in the Phase Model have become causal factors in the Philips Model.

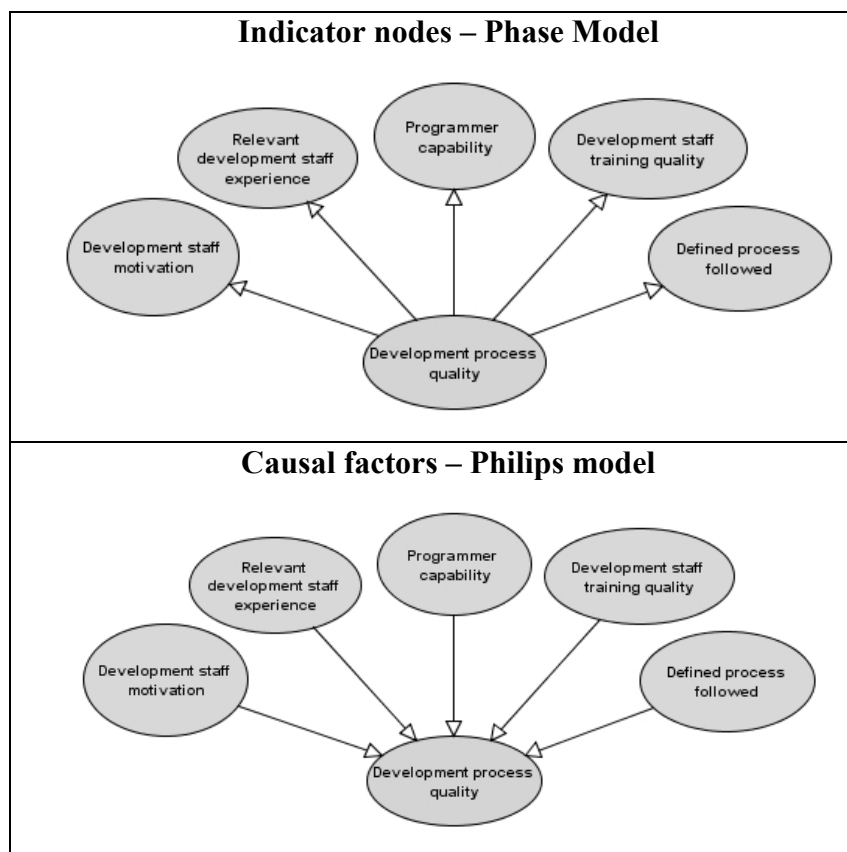


Table 4-3 Indicator nodes and causal factors

Our knowledge of software production suggest that the various factors shown in Table 4-3 are not independent. If *Development staff training quality* is high then we would expect *Programmer capability* to be high also. This suggests that Fenton and Neil’s original approach, to model the factors as indicators, was correct. The question therefore arises as to why they decided to change these to causal factors, which

remain independent provided no evidence is entered in the *Development process quality* node.

The main reason for this was a tendency for the users in Philips to enter evidence in both the indicator node and its underlying cause. Users were unaware that evidence entered in an indicator node would be blocked by this behaviour. Although the same is true when the direction of causality is reversed, users were able to more easily grasp the nature of d-connectedness in the case of converging, rather than diverging connections.

If this is true in general then it represents a serious disadvantage to the use of indicator nodes. It would suggest that indicator nodes are convenient for model builders, but misunderstood by users. This is clearly a topic of research that is beyond the scope of this study.

4.4.3 KLOC Instead of Function Points

The Phase Model uses Function Points (FPs) as its primary sizing mechanism. While FPs represent a widely accepted, technology independent size measure, there were a number of problems when Philips used the Phase Model.

The main problem was that Philips did not normally use FPs. Introducing FPs into Philips purely for these projects was not a practical option. Consequently Philips used the indicator node, *KLOC*, to enter their size information (see top half of Table 4-4). This was then back propagated to *New functionality implemented this phase* where it gives rise to a distribution of function point values, as opposed to a single point value. This immediately introduces a level of uncertainty into the model that was not previously present. This uncertainty is undesirable for two reasons. First, the uncertainty is propagated throughout the model, and second, the model tries to explain away model outcomes using the FP uncertainty.

A second problem with FPs is that they conflate simple size measures with project complexity. Some measure of software complexity is undoubtedly required: a complex program can be expected to give rise to more defects than a simple one. However, function points confuse two independent aspects of project complexity, namely *management complexity* and *technical complexity*. These two distinct types of complexity are explained in some detail in Appendix G. Briefly, management complexity includes aspects of a project which complicate management tasks. This might include factors such as the size of the project, its geographic dispersal or the

amount of sub-contracted functionality. Technical complexity refers to complexity which is inherent in the problem, such as the number of database interactions, timing and throughput constraints or specialised mathematical algorithms. Management complexity is already included in other parts of the Phase Model, thus leading to the possibility of “double counting”.

The Philips model therefore removes FPs as the primary sizing mechanism and replaces them with KLOC, as shown in the bottom half of Table 4-4. Since, FPs already include a measure of technical complexity, the transfer to KLOC involves adding some extra nodes to take account of this missing complexity element.

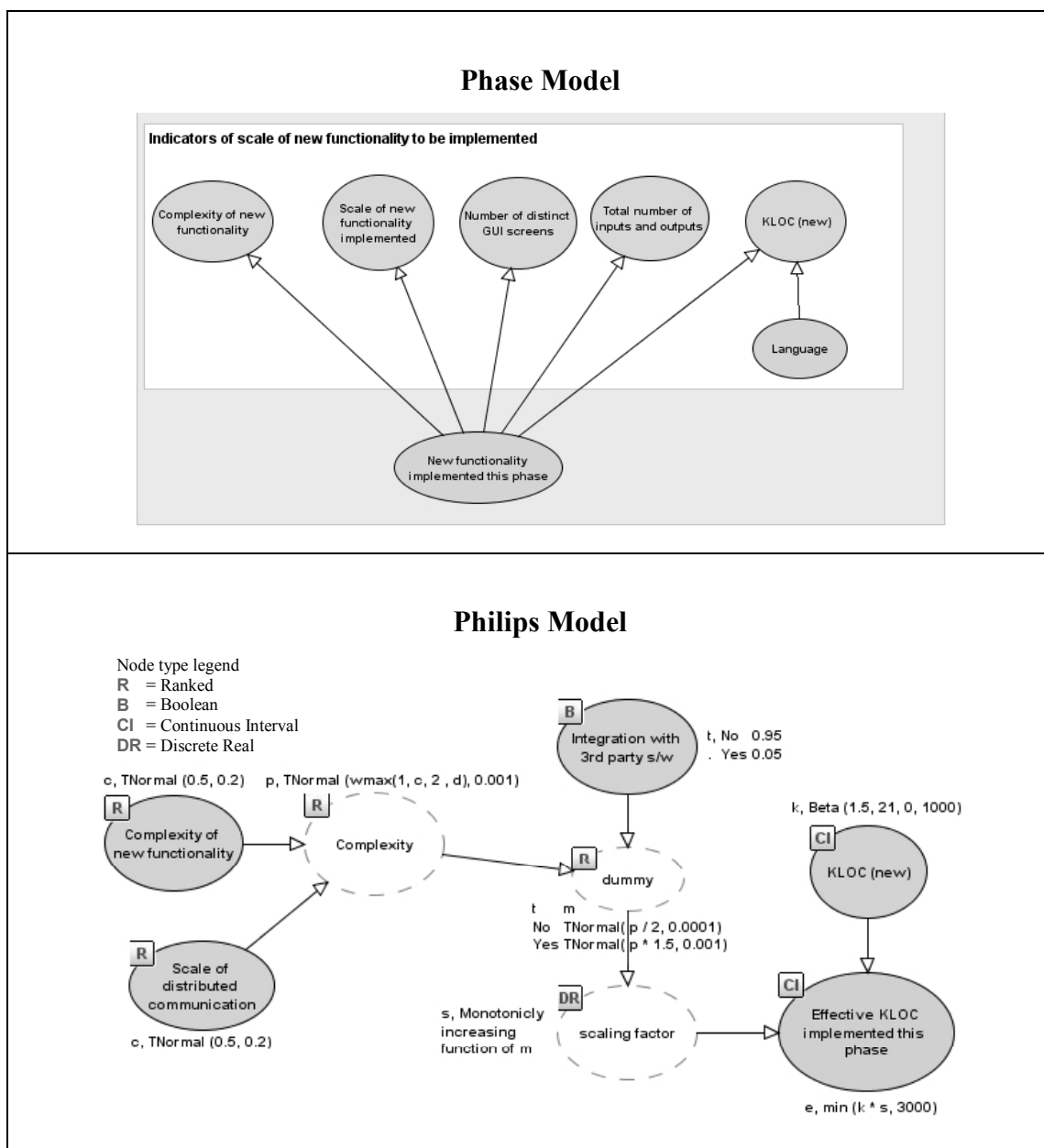


Table 4-4 Scale of new functionality implemented in the two models

A *dummy* Ranked node (scale 1 to 10) is used to combine a complexity rating with the Boolean node *Integration with 3rd party software*. Depending on its value the complexity can either be halved (False) or increased by 50% (True). A manually defined translation table in the *scaling factor* node then transforms the *dummy* node into a factor in the range 0.5 to 5 based on an empirically observed range provided by Philips.

In Appendix G I outline how a hierarchy of technical complexity factors can be constructed and added to models such as the Philips model. However, for reasons that will be explained in chapter 6, this approach to managing complexity was not pursued.

4.4.4 Philips Model Results

The Philips model was validated against 31 projects within Philips. The initial results are shown in the left hand graph in Figure 4-10. This shows the median predicted number of defects against the actual number of defects for each project. The solid line shows a least squares best fit linear trend line. For a perfect model, the predicted values would always exactly match the actual values. The trend line for a perfect model would therefore have a slope of exactly one and a y-axis intercept of exactly zero. Despite the reasonably good correlation of the predicted versus actual values, the slope and intercept of the trend suggest some systematic problem with the model.

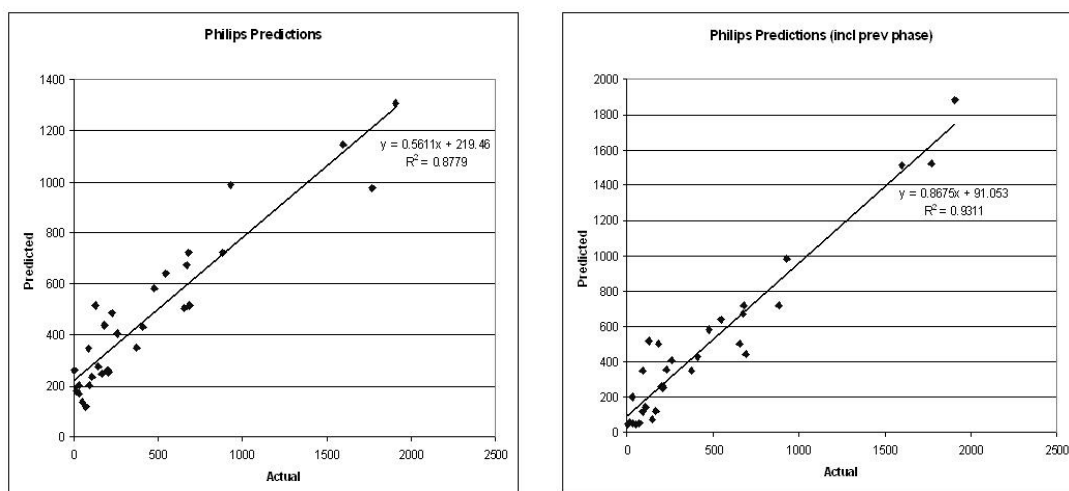


Figure 4-10 Philips model results, before and after inclusion of initial phase

After further investigation, Fenton, Neil and Krause determined that many projects included some existing code taken from previous projects. These projects were effectively built in two phases. Even where no code was included from other projects, the *Residual Defects PRE* node (Figure 4-7) was often left to use its prior distribution instead of being explicitly set to zero. Once these facts were taken into account, the results improved considerably. The improved results are shown in the right hand graph in Figure 4-10.

Of course, it is often possible to get equally good results using an appropriate regression analysis. The difference is that a regression analysis is only possible after an extensive metrics gathering programme. The regression analysis also suffers from all the other problems described in chapter 2. The Philips model results were obtained without a metrics gathering programme.

A more detailed analysis of the Philips' results can be found in [66], where I contributed a component based version of the model together with the graphs shown above (Figure 4-10).

4.5 Wooff, Goldstein and Coolen Software Test Model

Wooff, Goldstein and Coolen [219][49][50][169] have shown how Bayesian models can be used to improve software testing. Their work does not present a single model of the software test process. Instead, they have developed a method of creating a set of test models for any particular software project.

The resultant models can be used as management decision support tools to determine when a piece of software has reached the necessary level of reliability. The models can also be used to optimise test ordering by, for example, placing tests with the greatest chance of failure at the start of a test cycle, or by prioritising code where a failure would have the greatest impact.

Wooff, Goldstein and Coolen develop their models in a set of discrete steps.

1. List the *transactions* that the software performs. These are the major pieces of useful functionality. In object oriented terms these might roughly correspond to use cases, or in the case of XP, of user stories (discussed in chapter 5).
2. Each transaction consists of a (possibly large) number of *software actions* (SAs). Each SA is an individually testable processing function. Each SA becomes a node in a test model.

3. The relationship of the SAs must be determined. Some SAs depend on other SAs. For example, an SA that checks if a debit card number is valid, might depend on an SA which checks if a string is a number. The latter SA might be common to several other SAs. Similarly, related SAs may share some code or data, even if that shared code or data is not individually testable. The failure of common or related SAs clearly affects the chances of dependent SAs and so defines graphical dependency relationships.
4. The chronological ordering of SAs must be determined. If “number checking” must complete before “debit card number checking” can commence then it is the latter which depends on the former, and not the other way around.
5. For each SA the testable characteristics are identified. For example, the length of debit card numbers might vary between card suppliers, in which case this would be a testable characteristic. Each testable characteristic has a number associated with it, p , which is the proportion of tests of that characteristic that are expected to pass. This is an unknown and is modelled as a point mass (the probability that, given a single test failure, all tests of that characteristic will fail), plus a binomial. Each testable characteristic can have multiple SAs as parents.
6. Each characteristic must be partitioned into exchangeable sets. These are sets of inputs that, if the SA successfully processes one set of inputs from an exchangeable set, it will probably pass all of them. Likewise, if it fails on one set of inputs, it will probably fail all other inputs from the same exchangeable set. Each exchangeable set corresponds to a single test case and becomes an indicator node of the characteristic probability distribution, p , with probability equal to p .
7. Multiple inputs and multiple dependent SAs, are combined using the Noisy OR function (Appendix D).

The above procedure results in a set of disconnected BNs. Root node priors are determined by taking into account historic test results, the complexity of the code, the amount of new code, the track record of the programmers, the application domain and comparison with code of known validity. The following procedure is used to rank the reliabilities of each root node.

1. Rank each BN in order of expected reliability.
2. Within each BN, rank according to the reliability of each testable characteristic.

3. Within each characteristic, rank exchangeable sets. In doing so, the tester must be able to identify specific problems which are likely to affect the exchangeable sets that partition a characteristic.
4. Make initial estimate of unreliability in most unreliable BN and most reliable BN.
5. Use the initial estimates in 4 and the rankings to assign equidistant reliability priors to the roots.
6. If the initial priors do not produce results in agreement with the testers expectations (e.g. predicts too many or too few faulty modules) then adjust until they do.

Wooff et. al.'s paper discuss two case studies. In the first, which involved the management of several related credit card databases, they were able to produce the following conclusions.

- Of the 233 tests of the software, when run in their original order, the last 57 tests produced no gain in information.
- Of the 168 observables, 55 were not tested.
- The best 11 tests reduced the probability of at least one fault remaining by 95%. The next 156 tests reduced this by a further 1.5%. The remaining 66 tests had no effect.
- It was possible to demonstrate that a single additional test could be created that reduced the residual probability of software failure by a further 57%.

The BN approach was therefore able to reduce the number of tests that provided the same level of coverage, assisted in designing tests to exercise the remaining functionality and helped to schedule tests in a more efficient order.

In the second case study, a column of numbers in a database had to be changed using a fixed set of rules. Wooff et. al.'s approach produced the following.

- Of the 20 tests to be run, the last 8 produced no gain.
- 9 of the 20 tests reduced the probability of a remaining fault by 74%.
- It was demonstrated that a suite of only 6 test cases produced 100% coverage of the software characteristics.

In addition to quantifying the risk of remaining defects, the Wooff et. al. method captures the knowledge of the software test team, and so represents a valuable asset for a software project in its own right. However, in order to achieve this, a new model

must be built for each system being tested. This is a highly skilled and specialised task. In addition, the final software must be analysed and broken down into SAs, testable characteristics and exchangeable sets. Apart from the overhead that this involves, it assumes a relatively static code base - an assumption that is quite inappropriate in XP projects.

4.6 Siemens Model

Wang et. al. [206] from the Siemens company, have developed a model which shares characteristics with Fenton, Krause and Neil's project and phase models. The Siemens model is modular, like the Phase Model. Development of distinct models and phases of development can be combined by connecting separately constructed BNs. In common with the Project model, Wang et. al. claim to be able to perform trade off analysis between effort, schedule and quality.

The model consists of four separate sub-nets.

- The Component Estimation Model. This is a size model and is used to predict both the number of defects inserted and the duration of the code development. COQUALMO defect estimation parameters and COCOMO II effort estimation parameters are used.
- Test Effectiveness Estimation Model. Contains nodes relating to the experience of the test team and the effort put into designing the tests. A single output node contains the effectiveness of the testing process.
- Residual Defect Estimation model. The Component Estimation Model and Test Effectiveness Estimation Model are combined in this model to determine the number of residual defects after testing.
- Test Estimation Model. Uses information about the number of tests and the number of people to make predictions about test effort and duration.

The four sub-nets can be combined to form a model of a single software component. In their paper, Wang et. al. illustrate this further by combining models of three unit tested software components into a single integration tested software component with a further system test phase.

This model builds on the work of Fenton, Krause and Neil and is important in that it demonstrates that it is possible to build a single comprehensive model which predicts:

quality, effort and schedule, and that it is possible to perform trade-offs between the three.

4.7 Combination with COCOMO

Stamelos et. al. [193] have implemented aspects of the COCOMO 81 [24] model using BNs. This model is shown in Figure 4-11. Each of the four leaf nodes: Product, computer, Personnel and Project, are combinations of the corresponding intermediate COCOMO cost factors. For example, the Product node is an aggregation of the three product based cost factors:

- RELY - Required Software Reliability
- DATA - Data Base Size
- CPLX - Software Product Complexity

The aim of the model is to give better predictions of productivity $P = S/E$ where S is the size of the software and E is the effort involved. Each node, except the Productivity node is a ranked node with five values. The NPTs are defined manually using a combination of the COCOMO 81 project data and the authors' expert judgement.

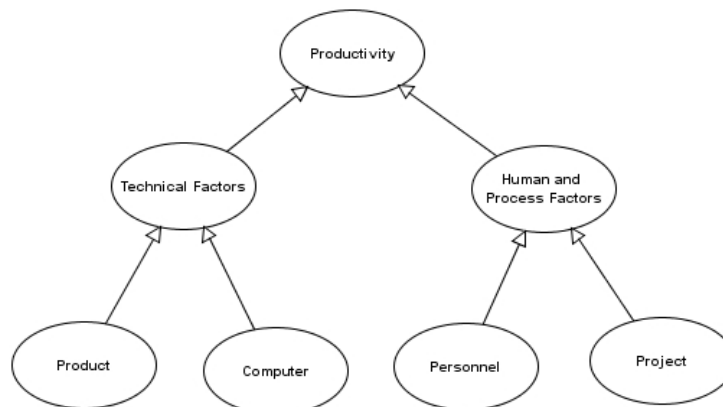


Figure 4-11 The COCOMO 81 BN

The model was tested using the COCOMO 81 project data [24]. The predicted productivity range included the actual productivity in 52-67% of cases depending on whether one or two neighbouring intervals were chosen.

Yang et. al. [222] took an almost identical approach, this time predicting effort rather than productivity.

This approach takes a traditional parametric formula and constructs a hierarchical BN model from it. This allows it to make some predictions regarding risk. However the approach makes no attempt to fully model causal interactions and therefore has little value when trying to perform trade off analysis.

4.8 Modelling Anti-Patterns in XP

Settas et. al. [181] have shown how to model anti-patterns [32][33] as a BN. They have given the name “Shaken but not stirred” to the particular anti-pattern chosen. It is an example of a “*commonly occurring solution to a problem that generates decidedly negative consequences*” [181]. In this case it is the problems created by rushing to use pair programming on an XP project without taking into account the personality profiles of the programmers involved. Psychology suggests that teams of mixed personality types should outperform teams with two of the same personality type [147], something that has been confirmed by Sfetsos et. al. [184].

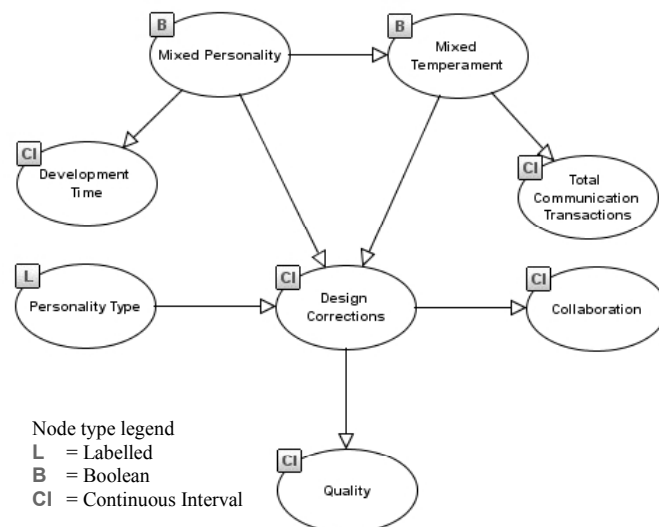


Figure 4-12 Anti-pattern Bayesian Net

The priors for this model were established empirically [182]. A set of 84 undergraduate programmers were paired according to their personality profiles as established by standard psychological tests [147]. Eight variables, corresponding to the eight nodes of Figure 4-12, were included in the data set. The PowerConstructor tool [39], which analyses conditional independence relationships in datasets, was used to examine the dataset and determine the topology of the BN.

Setting *Mixed Personality* to *True* and *Mixed Temperament* to *True* resulted in reduced *Development Time*, increased *Design Correctness* and increased *Quality*.

This is really a small causal model of a particular social factor affecting an XP environment. A full causal model would include this and many other factors.

4.9 Rational Unified Process

Bibi and Stamelos [22] suggested using “*iterative Bayesian Belief Networks*” as a means of modelling the software development process. They used IBM’s Rational Unified Process (RUP). Although iterative in nature and designed to track volatile requirements, RUP is far from being an agile methodology. It is a highly structured approach closely related to a specific toolset.

RUP breaks software development down into four major phases: inception, elaboration, construction and transition (essentially: analysis, design, development and maintenance). Each phase involves nine identified disciplines such as: business modelling, implementation and test. Each discipline can appear in any of the four phases, although the effort in each discipline will vary from one phase to another. So, for example, business modelling will be more prominent in the inceptions phase, whereas the test discipline will dominate in the transition phase.

Bibi and Stamelos proposed using iterative (i.e. dynamic) Bayesian Nets to represent a RUP. They pointed out the difficulty of building DBNs which fully modelled the whole RUP and showed how this might be simplified. Although examples of NPTs were given, there is no evidence that any of these BNs were ever built or validated.

4.10 Conclusions

We have seen that it is possible to construct modular, causal models of the software development process using BNs. The Fenton, Krause and Neil models have developed techniques for combining large numbers of causal factors into a single coherent model that can perform risk analysis, trade off analysis and aid in decision support. The Siemens model shows that a single size and quality model can be produced, while the Settas et. al anti-patterns model shows how a single XP factor can be modelled.

We will see in Chapter 6 that agile environments present particular challenges when applying some of this existing research. Before doing so however, we take a detailed look at XP, which we will treat as our “typical” agile methodology.

5 Extreme Programming

The aim of Extreme Programming (XP) is to create an agile software development environment that is responsive to changing user requirements. It does this by devoting as much developer effort as possible to coding and testing. Unstable requirements are no longer considered a demon that must be exorcised at all cost. XP recognises that the world changes, and customer requirements change with them. As one of the primary proponents of XP, Kent Beck, says in the subtitle of his book: we must “embrace change” [18].

The philosophy and practical details of XP are outlined together with a review of the relevant literature. The chapter ends by giving a rigorous definition of the main management metric in XP: project velocity (PV).

Novel contributions include a proof that PV can be measured using any absolute or relative measure of program functionality together with a demonstration that PV can be used to predict development timescales and costs.

5.1 *XP and other Development Processes*

Various software process models have been described over the years. The most basic is arguably the “Waterfall” model ([173]). The waterfall model splits the software development process into a series of discrete phases: analysis, design, code, test, maintenance. Each phase is completed before moving onto the next. In each phase, completion is marked by well defined deliverables which act as input to the next phase.

There are obvious problems with this approach.

- Its serial nature increases dependencies between analysts, system architects, developers and testers.
- Mistakes made earlier in the development lifecycle are more costly since they can only be fixed by going back and revisiting all intermediate phases. In the worst case, costs can rise exponentially [24].

In practise, few software projects adopt a strict waterfall approach. It is far too restrictive and inflexible for solving real world problems. Nevertheless, it remains a useful ideal against which alternative models can be compared.

The demands on Software Engineering have changed substantially since the inception of the waterfall method. Software is no longer confined to large mainframes

in specialised IT service departments. It is ubiquitous and often mission critical to a company's business. Competitive pressures squeeze timescales while demanding ever higher degrees of reliability, functionality and user friendliness. By the time traditional analysis and design phases are complete, the business opportunity has often passed [125]. Commercial organisations need, and demand, almost instant results. XP aims to flatten the cost of change curve and be receptive to changing requirements at any point in the project lifecycle: [18] p.23.

Extreme Programming (XP) [19] is, in many ways, the exact opposite of the Waterfall approach. Large scale analysis, requirements and design phases are (apparently) eliminated. After some relatively short consultations with the customer, teams plunge straight into code development. As has been pointed out by Stephens and Rosenberg [196] however, this elimination is more apparent than real. Requirements gathering and design are still present. They are simply spread over the development lifecycle and depend upon the successful implementation of various XP practices.

XP is predicated on a small set of *values* which underlie everything and are assumed to be desirable in any software project. These values are used to derive a set of *principles* which must apply to everything that is done in an XP project. The principles, in turn, are used to derive a set of *practices*. These are the mechanisms that an XP project uses in order to uphold the principles and values.

Discussion of XP is complicated by the fact that Kent Beck has revised his ideas considerably since the first publication of his book [18]. For example, the first edition lists twelve XP practices; the second edition [19] lists thirteen primary practices and eleven corollary practices. It is *not* the case that the later lists are a superset of the earlier list. Some practices are common to both, some have been added, while others have been dropped.

5.2 Values, Principles, Practices

XP values claim to be universal in nature, i.e. they apply to any software development environment. These are:

- communication,
- simplicity,
- feedback,

- courage,
- respect (added in Beck 2nd ed [19]).

These five values lead to a set of principles. Principles are less abstract and less universal than values. Consequently, they tend to be more XP specific. The mapping from values to principles however is not one-to-one, or even one-to-many. It is more the case that all principles arise from, and must be consistent with, all values. The relationship between principles and practices (see below) is similar.

Beck identifies the following XP principles in the two editions of his book:

First Edition	Second Edition
<ul style="list-style-type: none"> • rapid feedback • assume simplicity • incremental change • embracing change • quality work • teach learning • small initial investment • play to win • concrete experiments • open honest communication • work with people's instincts • accepted responsibility • local adaptation • travel light • honest measurement 	<ul style="list-style-type: none"> • humanity • economics • mutual benefit • self-similarity • improvement • diversity • reflection • flow opportunity • redundancy • failure • quality • baby steps • accepted responsibility

Table 5-1 Principles in 1st and 2nd editions of Beck's book

It would be inappropriate to go into detail on each of these. To do so would take up a great deal of space, none of which directly contributes to our models. For more information on XP principles, see [18] and [19]. It is the practices that most people think of as constituting XP. They have the most direct input into the models described here, and so it is these that we choose to describe in most detail.

5.3 XP Practices

XP Practices are the specific working techniques associated with XP. However, as with the values and principles, these have been revised over time by Kent Beck.

The different practices listed in the two editions of Kent Beck’s book are listed in Table 5-2. They are grouped together in three sections. The first section shows practices that are simply renamed or are otherwise directly comparable. In some cases, a single practice in the 1st edition has been split into several practices in the 2nd edition. For example, what is simply the “planning game” in the 1st edition becomes: stories, weekly cycle, quarterly cycle and slack in the 2nd edition.

	First Edition	Second Edition
Comparable practices	<ul style="list-style-type: none"> • pair programming • continuous integration • . • . • collective ownership • planning game • . • . • . • small releases • testing • . • 40-hour week • refactoring • on-site customer 	<ul style="list-style-type: none"> • pair programming • continuous integration • 10 minute build • single code base • shared code • stories • weekly cycle • quarterly cycle • slack • incremental deployment • test-first programming • code and tests • energized work • incremental design • real customer involvement
Differing practices	<ul style="list-style-type: none"> • metaphor • simple design • coding standards 	<ul style="list-style-type: none"> • sit together • whole team • informative workspace • team continuity • shrinking teams • root-cause analysis • daily deployment
Business practices		<ul style="list-style-type: none"> • negotiated scope contract • pay-per-use

Table 5-2 Practices in 1st and 2nd editions of Beck's book

The second section shows practices which appear to be quite different in the two editions. A separate third section lists practices which only appear in the second edition and which appear to define the software business model rather than the software development methodology.

The radical changes between the Edition 1 and Edition 2 practices suggest that some care must be taken when constructing XP models. This is a methodology that is flexible and is still evolving. Models must not be tied too closely to any one set of

practices. At the same time, the models must still capture the “spirit of XP”: its promotion of coding and testing above analysis and design, its iterative nature, its ability to react to changing requirements and expectations. These are properties common to all agile development methods (see for example the Agile Manifesto [9]).

Again, it would be wasteful to discuss all of these practices individually. They are explained in detail in [18] and [19]. Many are simply best industry practices and are neither unique to XP nor universal within XP projects. For example, studies by Aveling [16] and Sfetsos et al. [183] both show that projects tend to pick and choose which XP practices to implement, yet they all consider themselves to be XP projects.

The high degree of satisfaction expressed by participants in the Aveling and Sfetsos studies would also seem to refute claims by the likes of Stephens and Rosenberg [196] that XP has to be adopted in its entirety in order to be successful. They argue, for example, that discarding the traditional analysis phase without compensating with constant customer input, is a recipe for disaster. Stephens and Rosenberg’s views are corroborated to some extent in [131] and [53] where MacCormack et al demonstrate that creating prototypes can compensate for poor functional specifications.

Whether XP should be adopted in its entirety or not, the important point is that XP is often adopted piecemeal in practise. As such, it is important that the models developed here make minimal assumptions about dependencies between XP practices.

Instead, we will give a summary of the XP project lifecycle, stopping to highlight significant practices as they appear in context. This brings out the essentials of XP without spending too much time on (possibly transient) practices.

5.4 The XP Lifecycle.

By examining the XP lifecycle, the practices will be seen in the context in which they are used. This holistic approach gives a much better understanding of how the various practices are interrelated, as well as showing how development roles and management techniques are affected by XP.

This examination is not intended to be complete – this is not a user manual for XP managers. Rather, the aim is to highlight those aspects of XP which differentiate it from more traditional development environments and so indicate how XP models might differ from existing software process models.

5.4.1 User Stories

An XP project begins with a customer who writes the requirements. This will not be in the form of a lengthy requirements specification with voluminous appendices. Instead, it will be in the form of a set of *User Stories*. These are short, well defined pieces of functionality which can be written on a small piece of card. The customer allocates a priority to each user story.

The team may choose to put the user story cards on a wall in their workplace. These are split up into completed tasks, those to be completed in this iteration, those to be completed in this release and those to be done in the future. This is part of what Beck refers to as an *Informative Workspace* – one of the second edition practices [19].

5.4.2 System Metaphor

The project may choose to define a *System Metaphor*. This is often an analogy with a real world system. For example, an online purchasing website might use a supermarket as its metaphor.

The system metaphor fulfils a variety of roles.

1. It provides a consistent story that everyone involved with the project: customers, developers, managers and testers, can all understand.
2. It drives the architecture of the system. In our supermarket example we might decide to add aisles, counters and checkouts that are analogous to their real world equivalents.
3. It provides a means of exploring scenarios in the system and asking questions about failure modes. For example, what if a customer visits an aisle and finds a shelf empty?
4. Classes, objects, methods and attributes can all be named using a naming convention that is consistent with the metaphor.

Metaphors are not trivial constructs, and a suitable metaphor or mixture of metaphors may not always be available. When a team is unable to find a helpful or appropriate metaphor, the *Default Metaphor* (or *Naïve Metaphor*) can be used instead. This is not really a metaphor at all, and views the system as simply a computer system with suitable functional components. In our online shopping example we might create inventory, price and customer objects, gaining no further insight into their behaviour than is determined by the User Stories.

System metaphors have proved to be a difficult concept for many XP teams. Consequently they are poorly rated in terms of their usefulness and the degree to which they are adopted. Aveling [16] performed a review of eight XP studies and included four additional project surveys. Only one of the twelve projects unambiguously used a metaphor.

Beck himself seems to have abandoned the idea in the second edition of his book. The role of developing system metaphors is allocated to “Interaction Designers”, who might otherwise be known as analysts or architects [19] (p.75).

5.4.3 Architecture

At this point in the lifecycle, a project has a set of functional requirements in the form of User Stories with associated priorities. There may also be a rudimentary architecture in the form of a System Metaphor. If the development team have experience of similar projects in the past, and the technologies and components are already familiar, then the Planning Game (see 5.4.4) might commence immediately. Otherwise it is likely to be preceded by a short exploratory phase where technologies are evaluated and prototypes constructed. This adds some meat to the architectural skeleton and gives the project an opportunity to evaluate any obvious skill shortages.

Note that analysis and high level design still take place in XP, they are just not being documented in a structured fashion. The developers have engaged with the customer to determine the functional requirements and the team are acquiring the knowledge necessary to implement those requirements.

This approach imposes obvious limitations on XP when large systems are being developed, especially where multiple locations are involved. There may also be cultural or regulatory difficulties to overcome. If the team is working on government contracts for example, the contracts may stipulate the form and scope of the required documentation.

Cao et al. [35] describe a large (1,000 classes) banking application where XP practices were used to implement modifications to the code. The development team continued to employ a large scale architecture and design phase up front and found that this supported agile practices such as pair programming, refactoring and iterative releases. The use of predefined architectural design patterns was used to promote agility in the design.

Another large (500 KLOC) project involving XP [176][59], found the idea of a system metaphor too simplistic, and lamented the lack of a coherent large scale architecture and design. The lack of such an overall picture, which the whole team could refer to, caused considerable misunderstanding about the scope and interactions of user stories.

5.4.4 The Planning Game

The *Planning Game* attempts to break the project down into manageable releases of a few months duration each. It proceeds with the customer selecting User Stories in priority order. However, there must also be some input from the development team. Riskier requirements may have to be included in earlier releases in order to allow sufficient time to handle the risk. High risk aspects of the project should already have been identified through the Architecture phase.

The development team then estimates how long each User Story will take to complete. This should be no more than a few weeks. If the time required to complete the story is too long then it must be split into several stories.

User stories continue to be selected until the contents of the first release have been determined. The exact criteria for determining the contents of the first release are not strictly specified, but will involve some combination of the following.

1. The first release should be useful. It should contain sufficient features to be deliverable to the client and be used to solve real world problems.
2. The amount of time taken to deliver the release should be several months.
3. The principle constraint on the release contents can either be time based or content based, depending on the preferences of the customer and the developers.

Much of the main effort in the Planning Game may take place in a single Release Planning Meeting. This should take a day or two to complete. However, it is important to realise that the Planning Game is not a one-off exercise, it is an on-going process. As each release is delivered, more User Stories are allocated to the next release. The customer provides feedback about work done so far and the project team assess their forecasting accuracy, their understanding of the customer's requirements and their familiarity with the technology.

One of the few experimental studies of the Planning Game [106] has concluded that it is easier to use and faster to execute than pairwise comparison of requirements

[105], although only where pairwise comparison is bereft of tool support. It also turned out to be one of the least popular aspects of XP among students [134], although this was attributed to an aversion to planning in general. Aveling's study [16] perhaps casts more serious doubts on the Planning Game. His literature review suggests that several companies either explicitly rejected, or were forced to modify the Planning Game because it did not fit in with the local culture.

The aspect of the Planning Game discussed so far is what Beck refers to as the *Quarterly Cycle* in the 2nd edition of his book. This reflects a more prescriptive approach than he adopts in his initial description of XP, where a release is only defined as being one to three months in duration [18] (p.56).

5.4.5 Iteration Planning

Once the contents and timetable for a release have been agreed, the process is repeated on a smaller scale. The release is developed in a series of *Iterations*. Each iteration is a few weeks long – generally no more than three weeks.

User stories are split into *Development Tasks* of one to three days in duration. In order to do this, a developer must have a good understanding of the exact requirements of the story. This may involve further discussions with the customer.

At least one XP team, working on a large project, found this too restrictive and allowed some stories to span multiple iterations [59]. As one of the developers explains:

“Lately we have had some issues that we knew could not be effectively split into iteration deliverables and allowed up to 5 two-week iterations for completion.”

Task estimates are more accurate than story estimates as they are the result of a more detailed breakdown of the requirements. They are also generally estimated by the people who must implement them, so they are tailored to the productivity of the developers concerned.

Iteration planning is called the *Weekly Cycle* in the second edition of Beck's book. As with Release Planning, he is more prescriptive in the 2nd edition. Where the first edition suggested that iterations be one to three weeks in length, the second sets them at a fixed length of one week.

The terminology here, which distinguishes between User Stories, which are requirements at the project Release level, and Developer Tasks, which are more

detailed requirements at the Iteration level is as described by Beck [18]. It is not maintained by all XP proponents and practitioners. Many think of an XP project as consisting of one large collection of detailed User Stories, each of which takes a couple of days to complete. I will use Beck's terminology throughout.

5.4.6 Ideal Engineering Days and Load Factors

Developers must produce estimates of how much effort is required to complete User Stories. Within each User Story, estimates are produced of how much effort is required to complete Developer Tasks. This includes design, coding, testing and integration. In both cases the unit used for estimates is the *Ideal Engineering Day* (IED). This is actually a unit of effort: a developer-day, rather than a unit of time.

An IED is a day spent only on development activities, free from distractions, meetings or any other work. User Story and Development Task estimates therefore represent the number of developer days that it would take to complete the activity if all of the developer's time could be spent on that activity.

For example, if a User Story is estimated as taking 10 IEDs to complete, then it is assumed that it would take a single person 10 days of full time, uninterrupted work to complete the story. If two people were assigned, and their communication overhead was minimal, then it would take five calendar days of uninterrupted work to complete.

In practise, no developer is free to spend all their time on coding and testing. There are meetings, interruptions, learning activities, paperwork etc. All estimates must therefore be multiplied by a suitable value to bring the ideal engineering estimates into line with traditional person-day effort measurements. This value is called the *Load Factor* and normally lies between 2 and 3 [20]. So a ten day user story with a load factor of 2.5 will actually take 25 person-days to complete.

We will give a more rigorous definition of the Load Factor when we come to define our XP models.

There is no real upper bound to the Load Factor. It depends not only on the experience of the developer, but also on how much of the developer's time has been allocated to the project. Anecdotal evidence from websites confirms that the range of values suggested by Beck seems to be typical. Pelrine [162] varies the factor from 1.5 to 3.0 depending on how uncertain the estimate of the user story is, with a greater load factor accompanying a more uncertain estimate.

Although a single load factor is normally attributed to the team as a whole, they normally vary between individuals. The ratio of 28:1 is often quoted as the ratio between the productivity of the best and worst programmers. However this is based on a single set of small experiments described by Sackman et al [175]. As Prechelt has pointed out [165], the ratio of 28:1 actually comes from comparing two distinct sets of data with three of the developers opting to use assembler instead of a high level language. When like is compared with like, this ratio drops to 9.5:1. He goes on to analyse the data from multiple experiments and shows that the ratio is rarely greater than 4:1.

5.4.7 Test Driven Development

With the first iteration of the first release now planned, developers can set to work on development tasks. Test code is written before production code. All of these tests will initially fail, since there is no underlying code to be tested. Gradually, as user stories are implemented, more and more tests will start to pass. Unit tests should be independent and automatic in XP, concentrating on those aspects of the code which are most likely to break (not trivial things). They must always run 100% correctly ([18] p.116).

This approach is called *Test-First Programming* or *Test Driven Development* (TDD) and is seen as an important discipline in XP. It fulfils several functions.

1. It ensures that tests have been written for every development task.
2. It acts as a final detailed specification for the task.
3. The presence of a comprehensive test suite encourages developers to fix any perceived problems in *any* part of the code, even if they were not involved in its development. They can do this knowing that the essential functionality of the code that they modify can be tested for correctness. This is an example of *collective* (i.e. *shared*) *code ownership* – another one of the common XP practises.

Although point one above may seem obvious, it is worth pointing out that in the waterfall approach, tests are written at the end of the development cycle. If deadlines are tight then it is often testing which suffers. There is also the psychological advantage that testing at the end of the cycle is seen very much as a chore, whereas when it is done up front it feels more constructive as it helps to fully define the functionality.

This unwillingness to do proper testing at the end of coding was starkly displayed in an experiment by George and Williams [70], where three groups created a piece of software using TDD, and three control groups created the same software using a test-last approach. Despite explicit instructions, only one of the three control groups wrote any worthwhile test cases. This was reflected in the results where the TDD groups passed 18% more acceptance tests than the control, but took 16% longer to do so.

One of the most dramatic studies into the effectiveness of TDD was performed by Maximilien and Williams [133] at IBM, where a relatively inexperienced team was able to deliver a product with 50% fewer defects than expected when compared to the a more experienced team. The more experienced team however used an ad-hoc approach to testing.

There have been several studies which claim to show that TDD is no better than traditional test last development. Jones [99] summarises eight such studies in various schools and colleges and finds that results are mixed. Experiments with college students, such as Pancur et al [157] and Geras et al. [71], similarly suggest that TDD offer relatively little benefit over more traditional approaches. However all of these studies suffer from a common problem in that they are performed with small scale projects using university or school students. It is clear from the aims listed above for TDD that its true benefit is only likely to be apparent on larger projects.

The study by Geras et al. [71] counts every failure of a unit test as a defect, even if it can be trivially and immediately corrected. Highlighting once again the problem of the definition of a defect. This however, is a particularly poor definition of a “defect”. If the aim of counting defects is to measure software quality, then by this definition, the more defects the developers fix, the poorer their software quality becomes.

In a carefully constructed experiment by Erdogmus, Morisio and Torchiano [60], the experiment of George and Williams was repeated, this time using students and this time ensuring that test last groups performed the necessary testing. They found an increase in testing among the test-first groups and a proportional increase in productivity. They found no statistically significant difference in quality. They conclude by suggesting that the main benefit of TDD is the creation of test asset base which is sometimes skipped in test-last development.

5.4.8 Pair Programming

Programmers traditionally work on individually assigned tasks. They acquire an expertise in particular areas of a project and normally go on to “own” the code that they produce. We refer to this type of programming as Single Programming (SP).

Pair Programming (PP) is one of the most actively researched aspects of XP. The idea is that programmers should work in pairs at a single PC. One programmer is responsible for entering code while the other makes suggestions, looks for errors and thinks more strategically. We will refer to these two roles as the *driver* and the *navigator* respectively. The driver and the navigator are expected to switch roles periodically.

Beck recommends that programming should always be done in pairs and that anyone who refuses should be removed from the team [18] (p. 101). He also recommends switching partners every few hours [19] (p. 43).

The following benefits are claimed for pair programming.

1. Greater communication between team members.

Programmers are constantly exposed to different parts of the system as they swap programming partners. No single person therefore has exclusive ownership of any one area of the project. This has two knock on effects. First, it reduces project risk by minimising the effects of losing a critical team member. Second it promotes collective code ownership by increasing the willingness of team members to fix defects wherever they find them.

Sfetsos et. al. [184] performed controlled experiments with student pairs of the same personality type (as identified by the Keirsey Temperament Sorter [107]) and student pairs of mixed personality types. They found that mixed personalities were, on average, twice as communicative with their partner and that this communication gave rise to an average 30% increase in productivity. Communication volumes were measured by the pairs themselves, using written logs.

2. Constant review of code.

The navigator in each pair is constantly checking the code. Again this has two purposes. First, it enables the navigator to point out potential errors in the code. There may be boundary conditions, object states, or paths through the code that the driver hasn't noticed. The presence of the navigator should therefore reduce coding defects in exactly the same way as a conventional code review does. A study by Muller [140] seems to confirm this, in that he was unable to distinguish between the productivity and cost of pair programming in students and equivalent tasks performed by single student programmers with code reviews.

The second benefit of the navigator checking the code is their ability to think more strategically about the problem. The navigator doesn't have the immediate task of entering the code. They are therefore free to think about how the code resembles, or might take advantage of similar code elsewhere in the system. This might lead to better code reuse or potential refactoring operations which might take place. This should lead to smaller, cleaner code, with better class cohesion and lower coupling between classes (see, for example, Lethbridge and Laganier [123], section 9.2, for a good discussion of cohesion and coupling).

3. More intensive programming.

There are fewer opportunities for a programmer to be distracted when pair programming. Both pair members are keen to use their partner's time efficiently. This, together with the improved coding efficiency mentioned above, may be part of the reason why pair programming does not result in the halving of productivity so often feared by project managers. The effect of pair programming on productivity is discussed further below.

The intensive nature of pair programming is also one of the reasons why Beck recommends strictly limited working hours (the practise of "40 hour week" in

Beck 1st edition [18] and “energized work” in the 2nd edition [19]).

4. Enhanced team spirit.

Everyone in the team must work closely with everyone else. It is also claimed that, once initial scepticism have been overcome, developers often prefer to work in pairs.

5. Shorter learning curve.

Inexperienced developers or new team members are quickly assimilated. They are able to see the work of their more experienced colleagues at close hand and ask questions about the structure of the system and the programming techniques being employed.

The two most studied aspects of PP are its claim to reduce coding defects and its claim that PP is just as productive as individual programming. This latter claim is particularly bold since two people acting on the same piece of code, must now produce that code at twice the rate if they are to match the productivity of two individuals working independently.

A summary of some of the studies relating to PP is shown in Table 5-3. (These papers have been summarised by Williams and Kessler [212]. Williams and Kessler also provide an extensive discussion on different programmer pairing strategies: expert with novice, extrovert with introvert etc.) In terms of programmer productivity, they appear to give contradictory results depending on which study you choose to examine. However, when we look at the studies in more detail, a pattern does begin to emerge. Studies which involve short exercises (Nosek, Nawrocki, Heiberg, Lui, Canfora), tend to produce results which either show no clear pattern or show that PP is worse than SP in terms of total effort required. It is only in the longer term studies (Williams, Hulko, Vanhanen) that we see PP matching, or even exceeding the productivity of SP.

This is consistent with the hypothesis of Williams, Kessler, Cunningham and Jeffries [213]. They point out that it takes time for programmers accustomed to solo

work, to learn to work in pairs. They refer to this process as “Pair Jelling”. Once pair jelling is complete productivity increases dramatically.

Paper	Results	Comments
Nosek [154] 1998	Subjects were all professional programmers. Solos took 42.6 minutes to complete a task. Pairs took 30.2 minutes (so 60.4 mins effort – 50% more than SP).	Produced more functional, more readable code.
Cockburn and Williams [46] 2000	Students in software engineering. Initially PP took 60% more effort (pair jelling). PP required 15% more effort. PP produced 15% fewer defects. 10-20% code size reduction for PP.	90% enjoyed PP more. Also in Williams, Kessler, Cunningham and Jeffries [213] 2000
Nawrocki and Wojciechowski [148] 2001	5 SP, 5 x 2 PP, final year CS students. Found development time identical, so PP took twice the effort.	
Williams et. al [214] 2003	Students achieve 3% to 18% better project scores when PP.	Students more likely to stay with CS major when PP.
Heiberg et. al. [85] 2003	1 st year students. Measured number of test cases passed. PP was better than SP at the start. About the same later. The control group in this case consisted of two programmers who split the task work between them.	SP assumed to be slower at the start because they had to figure out how to split the tasks between themselves.
Lui and Chan [128] 2003	Performed two IQ test problem solving tests in pairs and solos. Also Transact SQL problems in PP and SP. 1 st IQ problem took 20.9% more effort when PP. 2 nd IQ problem took only 5.3% less time when PP, but was 85% correct vs. 51% for singles. SQL PP took same time as best member while poorer member learned.	
Parrish et. al. [158] 2004	Professional programmers working in small teams are roughly 4 times more productive if they do not work concurrently on the same code.	Concludes that driver/ advisor roles are essential to success of PP.
Hulko and Abrahamsson [88] 2005	4 case studies using a mixture of students and researchers. States that no pattern could be detected in productivity, either over time or in PP vs. SP. However figures are close for case 2 (18 LOC/hr PP, 21 SP) and are better in cases 3 and 4 (15 v13, 14 v 5, case 1 not available). States there’s no evidence for reduced defects. This was true in one of the two cases where this was measured, but figures show a 6-fold increase in defect densities in case 4 when SP was used.	Found that PP was worse for coding standard deviations, but better for comment density.
Vanhanen and Lassenius [203] 2005	Students with several years programming experience. PP required 100% more effort initially. PP required 5% less effort in later cases. PP resulted in 8% fewer defects, however 3 times as many defects were found post-delivery.	PP productivity vs. SP productivity not correlated with use case complexity. Not conclusively better in terms of method size and cyclomatic complexity (slightly better for PP case).
Canfora et. al. [34] 2005	Masters students. SP required 61% more “hours” initially. Only 3% more hours on a second run with same students.	Unclear whether this study is reporting effort or elapsed time.

Table 5-3 Pair Programming studies

The same study by Williams et. al. [213] also provides evidence to substantiate the claim that most programmers quickly overcome their suspicions and eventually prefer pair over solo programming. In an experiment with students at the University of Utah, 90% consistently expressed a preference for working in pairs. The programs produced by the pairs passed 10 to 17% more of their test cases than those produced by a control group of solo programmers.

The shorter studies also ignore some of the more intangible benefits of PP, namely increased team communication, common code ownership and enhanced team spirit. The extent to which these reduce project risk does not appear to have been studied so far.

Net Present Value (NPV) models have been created [125] which show that similar returns can be gained from project investments in both the SP and PP cases. Padberg and Muller have created a more comprehensive NPV model [156] which suggests that pair programming is most advantageous when time to market is critical.

5.4.9 Development Practises

XP development can involve a range of additional practises.

Developers are encouraged to constantly *refactor* their code. Refactoring involves identifying inefficiencies in the existing design and restructuring the class and package arrangement in order to remove those inefficiencies. Refactoring can be done a little bit at a time, by gradually reducing the functionality in a class until it eventually disappears. It doesn't have to be done all at once. The result should be smaller code with greater class cohesion and lower inter class coupling. Some evidence for this has been found, for example, by Wood and Kleb [218], by Cockburn and Williams [46] and by Nawrocki and Wojciechowski [148].

Refactoring also helps to promote another XP practise: maintaining a simple design. XP aims to create the simplest design that does the job. Hooks for unspecified features that might come in useful one day are to be avoided in XP.

Code is *integrated frequently*, so that the whole team is always working with the most up to date version of the code. This, together with the other XP practices of pair programming (with rotation) and extensive unit testing, promote a sense of *common code ownership*. All of the code in an XP project belongs to the whole team. No single person "owns" particular parts of the project.

Strict adherence to *coding standards* is expected in such an environment, although curiously, in one of the few studies to measure this Hulkko and Abrahamsson [88], found more deviations from coding standards amongst paired programmers than among single programmers.

5.4.10 Workspace

The XP workspace should be an *informative workspace*. Story cards are placed on walls, neatly arranged into completed, ongoing and unassigned sections. At a glance, anyone entering the workspace can get an immediate overview of project progress. Any metrics being measured should also be shown in public wall charts.

The workspace itself should be open plan with the team all sitting together. Beck claims this leads to more and better team communication, although he recognises the need for privacy and advocates individual cubicles as well [18] p.79.

Between 1984 and 1986, Lister and DeMarco [56] (p. 49) performed a series of programming experiments. Of those who performed best, the majority described their environment as scoring better in terms of privacy, quiet, workspace size and freedom from interruptions. This would seem to be evidence against the use of open plan workspaces. Open plan advocates, including Beck, counter that team spirit and communications are improved in a more open environment. Becker and Sims [21] argue for a balance between these two extremes.

5.4.11 Acceptance Testing and Onsite Customer

User stories are not considered complete until the customer has accepted them. Each user story must pass the functional tests written by the customer. This does not necessarily mean that 100% compliance is required. It is up to the customer to decide whether their requirements have been met.

Unlike unit tests, which are written by developers in order to verify that their code works as expected, acceptance tests are written from the customer's perspective. Acceptance tests are intended to prove that the business requirements expected by the customer have actually been delivered.

User stories that do not pass their acceptance tests must be carried over to the next iteration. The amount of rework effort allocated to the failed user story will obviously depend on how seriously flawed the implementation is. Trivial details about the user

interface will require less rework than major misunderstandings about the business requirements.

Major misunderstandings can be avoided by employing the XP practise of having an onsite customer. XP projects have no extensive requirements specifications to draw upon – only the user stories. This leaves considerable room for ambiguity. The small story cards used to write down user stories do not provide a great deal of space. This gives the author very little scope to write down examples that might illustrate a user story or clarify any subtleties.

Martin, Biddle and Noble [132] have pointed out that the role of the XP customer is an extremely demanding one. They must act as the only conduit between all other stakeholders and the development team. It is their responsibility to refine the requirements into user stories, to write acceptance tests and be available all of the time, at an instant's notice to developers. (As an aside, it is interesting that in all three cases examined in [132], some form of formal requirements and architecture are developed before the XP methodology is applied.)

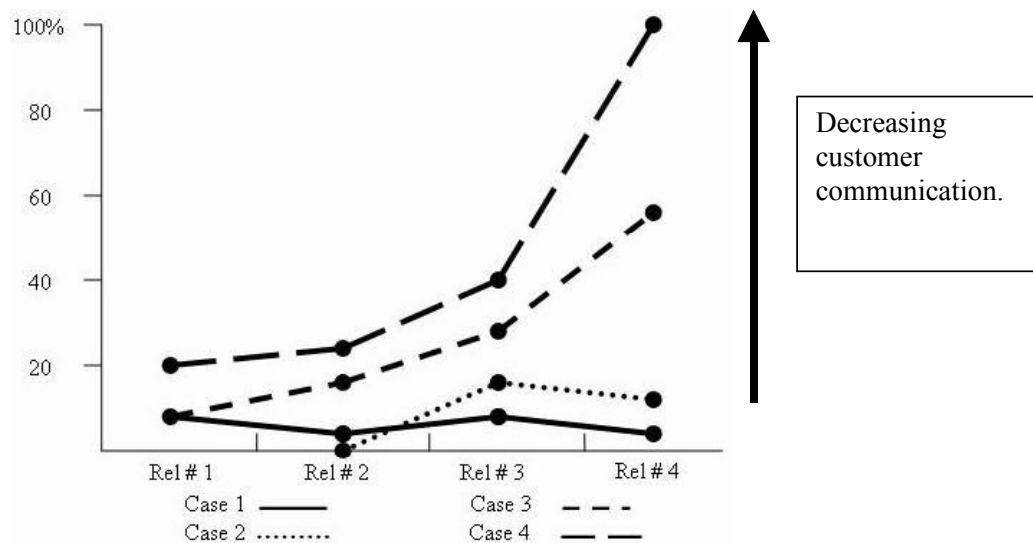


Figure 5-1 Percentage of defect fixing tasks in different projects¹

As Deursen [58] noted, XP provides developers with a whole range of techniques with which to accomplish their role. These include common code ownership, constant refactoring, test first development etc. However, no such guidance is offered for the

¹ This diagram is included with the kind permission of the authors [112]

customer. Deursen's workshop recommends that the onsite customer should actually be an onsite team, capable of fulfilling the multi-faceted role of the onsite customer role. The workshop also recommended the inclusion of a "so that" section in each user story. This should show how the user story helps to meet a particular business requirement.

Despite these difficulties, the role of the onsite customer in XP is essential. Korkola et. al. [112] performed a case study comparing four different projects, with different levels of customer communication. They measured the different rates of rework for each of the four projects over four "releases" ("iterations" in Beck's terminology). The results are shown in Figure 5-1. Case 1 had the customer onsite for 80% of the time; case 2 had the customer onsite at the start and end of each iteration, but was available for face to face communication the rest of the time; case 3 had the customer available only at the start and end of each iteration, and in case 4 the customer was only onsite for the first two weeks of the project.

The percentage of effort devoted to fixing defects, including specification defects, varied greatly in the four case studies. Where customer input was very high, only 6% of effort was spent fixing defects. Moreover this level remained constant across iterations. At the other extreme, when customer input was very low, the time spent fixing defects grew across iterations until it reached about 40% in iteration 3.

5.5 Project Velocity

At the end of the first XP iteration the following should have taken place.

1. The customer will have met the developers and given them an overview of the project.
2. The customer will have written down the project requirements in the form of user stories. The customer will have assigned priorities to each user story; developers will have assigned effort in ideal engineering days (IEDs). Each user story will occupy a couple of weeks of effort.
3. An architecture, possibly including a system metaphor will have been selected.
4. A release planning meeting will have selected the user stories needed for the first release.
5. An iteration planning meeting will have selected the user stories to be completed in the first iteration of the first release. These user stories will have

been split into development tasks. These tasks will have been assigned an effort of the order of a couple of IEDs each.

6. Unit tests for the user stories selected for the iteration will be complete.
7. The customer will have written acceptance tests for the user stories.
8. Some of the user stories selected for the iteration will have passed their acceptance tests, others will not. Those user stories which have passed are deemed to be completed.

A new meeting will now be required in order to plan the second iteration. This has two objectives.

1. Uncompleted stories from iteration 1 are re-examined. The effort required to complete these stories is assessed.
2. New stories must be selected for the second iteration.

XP employs a simple rule in order to meet the second of these objectives: take the estimate for all the stories *completed* in the previous iteration and add them together. This is the *project velocity* [20] (p.59). Uncompleted user stories are selected in priority order until the sum of their estimated IEDs equals the project velocity. These are the user stories that are selected for the next iteration.

If we denote the estimated effort for the j^{th} completed user story in iteration i by U_i^j then the project velocity V_i , for iteration i , is given by:

$$V_i = \sum_{j \text{ completed in } i} U_i^j \quad \text{Equation 5-1}$$

Note that we are using *User Stories* in our definition of velocity. These have IEDs which are measured in weeks, even though iteration length is itself only a week or two (depending on which version of Beck you read). The rationale behind this is not explicitly stated by Beck, nevertheless it does make some sense. User Stories are the unit of functionality – they are the things which are acceptance tested and can therefore be signed off as complete by the customer. Presumably the more accurate IED estimates, arrived at after splitting each User Story into development tasks, are used.

Example

Suppose that there are 8 user stories in total for a project prioritized as shown in Table 5-4.

User Story	Priority	Ideal Engineering Days	Iteration
US1	1	$U_1^1 = 10$	1
US2	2	$U_1^2 = 5$	
US3	3	$U_1^3 = 15$	
US4	4	$U_2^1 = 20$	2
US5	5	$U_2^2 = 10$	
US6	6	$U_3^1 = 20$	3
US7	7	$U_3^2 = 10$	
US8	8	$U_4^1 = 10$	4

Table 5-4 Estimating user stories

Suppose that in the first iteration the developers feel they will have a total of 30 IEDs available. Then they will plan to complete the first three user stories *US1*, *US2* and *US3* in iteration one. If the first three stories pass their acceptance tests in iteration 1, then the project velocity V_1 will be 30.

It is important to emphasize this is an *estimate* of how many IEDs the team thought it would take to complete the work. It is not the number of actual person-days taken, nor is it the calendar time taken.

Assuming that the next iteration, $i + 1$, is the same length as iteration i , the customer selects the highest priority uncompleted user stories whose estimated IEDs sum to V_i . These user stories are then scheduled for iteration $i + 1$. The work scheduled for iteration $i + 1$ therefore has the same estimated ideal effort as the estimates for the actual work completed in iteration i . Expressed more concisely: we can expect to be as productive in the next iteration as we were in the last. So, using the example shown in Table 1 we have $V_1 = 30$, and so we would schedule tasks US4 and US5 for iteration 2 since the estimated IEDs required for these two user stories is 30. If it turns out that in iteration 2 we only manage to complete US4 then $V_2 = 20$.

Note that the actual time taken to complete a user story is not used here. To illustrate why it is unnecessary, let us suppose that the developers working on *US1*, *US2* and *US3* had carefully filled in time sheets and determined that the time spent on those stories was not the 30 days that was estimated, but actually took 36 IEDs, i.e.

there was a bias, b_1 , in their estimates. (Note that the use of the word “bias” here is not intended in the statistical sense of a biased estimator).

If A_1^j are the actual efforts taken then:

$$b_1 = \frac{\sum_j U_1^j}{\sum_j A_1^j} = \frac{V_1}{\sum_j A_1^j} \quad \text{Equation 5-2}$$

In this case, b_1 is 30/36. A project manager might assume that the remaining tasks have been underestimated by the same amount and multiply them by 6/5 to compensate. *US4* and *US5* would then have new estimates of 24 and 12 respectively. The manager knows that the team did 36 actual IEDs work in iteration 1 and, all things remaining equal, are likely to do 36 actual IEDs in iteration 2, so he schedules tasks whose updated estimates add to 36. This would result in *US4* and *US5* being scheduled for the next release - exactly the same as we had before (except that there is now a lot more time tracking and form filling being done).

This scheduling mechanism assumes that the ratio of effort (people × working days) to PV remains constant. This assumption can be justified by examining the two possible estimation scenarios.

1. User story estimates are being consistently overestimated or underestimated. This consistency ensures that any bias in the estimates for the previous iteration will be repeated in the current iteration.
2. If there is no consistent bias in the effort estimations, i.e. there is as much overestimation as underestimation, then these inaccuracies will even themselves out over multiple iterations. This further assumes that teams are able to schedule additional work in an iteration when the effort of existing tasks has been over-estimated and slack time is available.

If the next set of user story estimates do not sum to exactly V_i then various options are possible.

- One of the user stories can be broken down further. In fact, as we have already seen, each user story is broken down into development tasks.
- An alternative user story can be selected, although this breaks the prioritization guidelines in XP.

- Stories with estimates slightly less than V_i can be chosen.

It is recognized that in real world projects there are dependencies between iterations. Efficient implementation sometimes demands that the order of work differ from the customer's desired prioritization. There are also rework tasks which arise when work in a previous iteration must be revisited. We will not explicitly address any of these issues. In the models we develop, inter-iteration dependencies will simply be subsumed within other, more general, overheads.

The introduction of pair programming has no effect on how these calculations are performed. If the developers in our example are paired then the number of user stories completed in iteration one might decrease, say to a set of user stories whose estimates totalled 20 IEDs instead of 30. User stories adding up to 20 IEDs are then scheduled for the next iteration. Similarly, the extent to which user stories are complete serially or in parallel has no effect on project velocity.

5.5.1 Story Points

Story Points provide an alternative way of measuring User Story size [48]. A single story is taken as a benchmark for measuring the relative size of all other stories. If the story chosen is identified as the smallest story then it will be given a story point value of one. A story that is twice its size is allocated a story point value of two and so on. Story points for completed stories are then added up at the end of an iteration and their sum used to allocate user stories for the next iteration.

There is a common misconception that story points are in some fundamental way different from IEDs. Story points are perceived as being on a ratio scale while IEDs are perceived as being absolute. In fact this is not relevant. To see this, consider the IED estimates for stories completed in iteration i as shown in **Equation 5-1**. Now consider the same user stories estimated using story points. Let the story point estimate for story j in iteration i be S_i^j .

If we *assume* that IEDs represent an absolute size measure then they also preserve the relative sizes of the stories. In other words there exists a constant α such that:

$$U_i^j = \alpha S_i^j \tag{Equation 5-3}$$

It follows that the PV measured using IEDs, is just α times the PV measured using story points (call this V_i').

$$V_i = \sum_j U_i^j = \sum_j \alpha S_i^j = \alpha \sum_j S_i^j = \alpha V_i' \quad \text{Equation 5-4}$$

A team measuring PV using IEDs will choose stories for iteration $i + 1$ such that their estimates for the stories, U_{i+1}^k , match the velocity in iteration i :

$$\sum_k U_{i+1}^k = V_i = \sum_j U_i^j \quad \text{Equation 5-5}$$

Similarly, a team measuring velocity using story points will choose stories such that:

$$\sum_k S_{i+1}^k = V_i' = \sum_j S_i^j \quad \text{Equation 5-6}$$

But Equation 5-6 is just Equation 5-5 multiplied by α so both teams will choose identical stories. The reason this happens is that the ratio between the two measures produces the same ratio in their respective project velocities. So even *if* IEDs represent an absolute scale, it is only their measure of relative size that is used in determining which stories are chosen in the next iteration.

The example in the previous section, where it was shown that actual effort measurements result in exactly the same tasks being scheduled, works for exactly the same reason. In that case, both measures are assumed to be absolute, and therefore they too are related by a scaling constant.

We can conclude that *any* relative or absolute measure of user story size will result in the same stories being scheduled in each iteration. We will continue to measure user stories and PV using IEDs, but the reader should bear in mind that everything that is said about PV from now on is applicable regardless of the measurement used.

5.5.2 Project Velocity and Project Planning

We have seen how PV can be used to allocate user stories to the next iteration. Clearly, if we can determine how PV varies over time we will be able to predict the

rate at which new functionality will be delivered. In particular, it allows us to predict the point at which all of the functionality has been delivered and therefore when development is complete. This in turn has a clear impact on project resourcing and cost.

Suppose there are N stories in a project in total. If U_i is the estimated effort in IEDs to complete story i , then the total *development* effort D to complete the project is:

$$D = \sum_{i=1}^N U_i \quad \text{Equation 5-7}$$

If the project velocity V is constant over each iteration then the number of iterations n required to complete the project is:

$$n = \frac{D}{V} \quad \text{Equation 5-8}$$

This formula is essentially what is being used in *burn down* charts [179] which are traditionally used in one of the other agile development methods: Scrum [201]. Burn down charts show the amount of functionality still to be delivered at the end of each iteration (or *sprint* as iterations are referred to in Scrum). A hypothetical burn down chart is shown in Figure 5-2. The slope of the graph is the project velocity and the point at which the graph intersects the horizontal axis is the point at which all of the functionality has been delivered.

There are several problems with this simple approach.

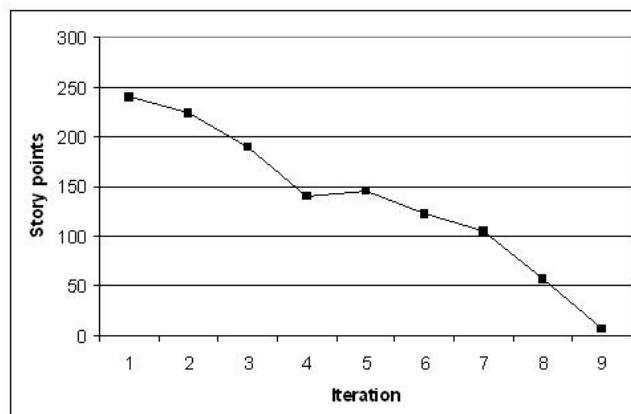


Figure 5-2 A typical Burn Down chart

1. It takes no account of the trends in PV over time. Many projects report low initial productivity, gradually rising on subsequent iterations [3][6][215].
2. It takes no account of the uncertainty in user story estimates. Jorgensen et. al. have shown that there are wide variations in developers' estimates for the same task, and that even a single developer can give widely different estimates for a task on two separate occasions [101][102][103][137].
3. Similarly there is risk associated with the size of the project. In iteration 5 in Figure 5-2 more story points have been added than have been completed causing the number of story points remaining to increase. If the velocity at iteration 4 were extrapolated without taking account of this risk it would produce a highly inaccurate and optimistic prediction for the project delivery time.
4. The simple approach relates development effort, measured in IEDs or story points, to velocity. A manager needs to know how much actual effort in person-days to allocate to the project. Actual effort E is related to development effort D via the load factor l ($E = l \times D$, see section 5.4.6).
5. Changes to the project environment, such as changes in XP practises, are not taken into account.

These problems can all be addressed using BNs. BN learning can be used to identify trends in PV and to determine the load factor. Risk is inherently modelled in BNs by appropriate choice of uncertainty in the priors. Finally, environmental changes can be modelled as causal relationships. Successfully attacking these problems should result in a model which is richer and more accurate than a simple burn down chart.

5.6 XP Models

Most of the effort models in common use on commercial XP projects are little more than variations on the simple burn down chart discussed in the previous section. The lack of formal requirements and a culture of minimising software process overheads, mean that there are relatively few process metrics to model.

Williams and Erdogmus [211] developed a Net Present Value (NPV) [172] model of Pair Programming (PP) – one of the key practices advocated by XP. NPV models take into account the fact that earnings in the future are worth less than the same dollar earnings today. The model combines:

1. productivity rates,
2. code production rates (derived from the literature),
3. defect insertion rates
4. and defect removal rates.

Using empirical values for PP productivity and delivered defect rates [46][154][213], the model predicts that pair programming is a “viable alternative to individual programming”.

Padberg and Müller [156] also created an NPV model of XP. Their model uses market pressure as the principle means of discounting the NPV. The model was tested under various different assumptions about performance and defect rate improvements under PP. The results indicate that the value of both of these parameters is crucial. When market pressure is high, and there is sufficient improvement in both LOC and defect rates, then PP can indeed deliver an advantage.

Several groups have constructed System Dynamics (SD) Models of XP. SD models a system as a collection of stocks, flows and feedback loops, and was first applied to software engineering by Abdel-Hamid [1][2]. Misic, Gevaert, and Rennie [136] attempted to model the interaction of various XP practices. They particularly concentrated on pair programming, refactoring, test driven development and iterative development. Simulation results indicated that XP has an advantage when pairs worked well together and did not swap frequently.

Kuppuswami, Vivekanandan, and Rodrigues [117] also created an SD model. They were able to successfully simulate the flattened cost of change curve claimed by Beck [21] (p. 23).

Cau et al [37] developed a custom simulation to model the XP process, calibrated using data from a real XP project. Once calibrated, their model was able to reproduce empirically derived results [70] about the effects of test driven development (one of the recommended XP practices).

All of the above provide explanation, insight or validation of XP techniques. What none claims to do is offer combined prediction and risk assessment for project managers.

5.7 Summary

Extreme Programming is an agile development method which seeks to flatten the cost of change curve over the development lifecycle. It’s core practices, such as

iterative development, pair programming, test first development and onsite customer interaction have been extensively studied and have been found to be generally effective.

XP is still evolving however. Practices are being added and removed over time, and the effectiveness of core practices continues to be studied. It is imperative therefore that models of the XP development process make minimal assumptions about XP.

PV is the key metric in XP. It can be measured using any ratio scale. If we can model how it changes over time using BNs then we can predict delivery dates with greater accuracy as well as providing estimates of the uncertainty surrounding the predictions.

6 Adapting Causal Models to Iterative Development

We have seen how BNs can be successfully used to model software development processes. There is an obvious way to extend these models to iterative environments: simply take an existing model, such as Fenton, Krause and Neil's Philips model (4.4), and replicate it multiple times. Each copy then becomes a separate timeslice in a DBN (3.2). There are several questions that must be raised when dealing with this approach however.

1. What is the size of the resulting BN and are posteriors computable in an acceptable period of time?
2. Is the data collection and data input required acceptable in agile projects?
3. Can multiple sources of project data lead to inconsistencies in the data?
4. Is an unambiguous interpretation of the project data always possible?
5. In an interactive tool like AgenaRisk [8], used to test the models developed in this thesis, how can we build iterative models efficiently?

I address these questions in this chapter. As a result, we shall see that this approach is unlikely to be practical in an agile environment. Instead, we shall start to develop the case for learning BN models.

Novel contributions include a general purpose data extraction and mapping mechanism for populating BN models with data from heterogeneous databases, and a scripting language for building DBNs which separates out timeslice definition from timeslice connection and observations.

6.1 Model Size

Suppose we use the PHILIPS model (4.4) as a template for building a DBN model of an iterative software development environment. The Philips model contains 63 nodes. A small XP project might contain 10 iterations in a single release, giving a total of 630 nodes for the total model. What impact will this have on the computability of the model if we use the Junction Tree Algorithm (JTA) outlined in Appendix B? We consider each of the stages of the JTA.

Moralisation

Suppose we have a moral graph M of a BN B . If B has n nodes, then adding a new node N will add at most $n - 1$ edges in M as moral connections are established to every

node for which N is a parent. Typically this will only consist of one or two edges however. The moralisation stage therefore grows as $O(n)$.

Triangulation

Optimal triangulation, in the sense of minimising the sum of the edge lengths (in effect the number of edges), is NP hard [141]. Optimal triangulation, in the sense of minimising the weighted state space (i.e. the product of the number of states in each node in a clique), is NP complete [15]. The algorithm used by Agena, is a greedy algorithm (i.e. it makes a locally optimal decision at each stage in the hope that the global solution will be close to optimal). The greedy algorithm works in polynomial time [115].

Message Passing

Let C be the collecting clique in the Collect-Evidence phase. All cliques except C emit a single message. Similarly, during the Distribute-Evidence phase every clique except C receives a message. So if n is the number of cliques in the junction tree, then $2n - 2$ messages will be passed in total. The number of messages passed therefore grows as $O(n)$. The number of cliques will not necessarily be the same for every timeslice, since this depends on the exact operation of the triangulation algorithm. However, the Markov condition limits the connectivity between timeslices, ensuring that the number of cliques will grow as $O(t)$, where t is the number of timeslices [144]. We can therefore expect that the total number of messages passed will also grow as $O(tn)$.

Clique and Message Size

Clique sizes and message sizes all grow as the product of the number of states of the nodes involved. If all nodes have s states, then cliques and messages will have s^n states, where n is the number of nodes. Clique and message sizes therefore grow exponentially with the number of nodes.

Looking at these four items: moralisation, triangulation, message passing and clique and message sizes, it is the size of the cliques which is the most serious concern. Software process models include either effort or defect counts (or both). These are

modelled as either continuous distributions or as integer interval distributions. In both cases a wide range of values with suitable discretisations is required. This results in a large number of states for these nodes. There also tend to be multiple instances of effort and defect nodes as we have already seen in the Philips and Fenton, Krause and Neil Project Level models. Cliques containing these nodes will therefore contain the products of large numbers of states.

The JTA retains all of its cliques throughout the execution of the algorithm (so that it all posteriors can be made available at once). A single junction tree for a software process DBN model will therefore contain multiple copies of these large effort and defect cliques.

Nodes in clique	Number of states
New_defects_in, Pot_defects_given_spec_and_documentation_adequacy, Prob_of_avoiding_defect_in_dev, Quality_of_any_previous_documentation	200,000
Residual_defects_post, Defects_fixed, Total_defects_in	169,785
Defects_fixed, Defects_found_in_testing, Total_defects_in	153,615
Residual_defects_pre, Total_defects_in, New_defects_in	126,500
Inherent_pot_defects, Prob_avoiding_spec_defects, Pot_defects_given_spec_and_documentation_adequacy	61,600

Table 6-1 Largest clique sizes in Philips junction tree

As an example, the number of states in the five largest cliques in the Philips model is shown in Table 6-1. Linked timeslices would likely include even larger cliques. The total number of all states in all cliques in the Philips junction tree is of the order one million. If each of these probability values is held as an eight byte double precision real number, then the cliques require 8 Mbytes of storage. Adding the sepsets nearly doubles this value to 16 Mbytes. If before and after values are held (to prevent recalculation under different scenarios) then this value doubles again to 32 Mbytes. This does not include, priors, marginals, evidence masks and graphical display information. In fact, when run in AgenaRisk, the Philips model has a memory footprint of over 100 Mbytes.

This limits the usefulness of the JTA when constructing DBNs. This does not mean that we cannot use models, such as the Philips model as the basis of a DBN timeslice. However it does mean that we cannot unroll all of those timeslices and evaluate the resultant BN using a single junction tree. We will have to use one of the specialised inference algorithms mentioned in section 3.2.

6.2 Entering Data into BN Models

The Philips model was trialled extensively by Philips. Each project had to provide 29 separate pieces of information [67], of which 27 were qualitative in nature and two were quantitative (both being code size in KLOC). If we used the Philips model as the basis of a DBN model for an iterative environment, and we wished to preserve the Markov property by not sharing common factors across timeslices, then we would have to enter comparable amounts of information in every timeslice. In this section, we examine how this data might be collected and how it might be input into the model.

6.2.1 Quantitative Data Collection

Software process models can potentially combine quantitative data from a variety of sources: defect data can be extracted from bugs databases, scheduling and effort estimates from project plans, usage information from sales databases and web download statistics. Examples are listed in Table 6-2.

Data Source	Examples	Usage
Bugs database	Bugzilla	Previous defect counts. Model calibration. Reliability growth curve estimation.
Project plan	MS Project	Key dates, effort estimates, actual efforts.
IDE	Eclipse	Software size metrics. OO metrics.
Code management	CVS Perforce	Entropy measures.
Sales database		Dates of new operational profiles where software use changes.
Code coverage	Cenqua Coverlipse	Test coverage metrics.
CASE tools	Rose	Object oriented metrics. Function points.

Table 6-2 Model data sources and their uses

Much of this data must be entered into the model manually - a task which is both laborious and error prone. This overhead is particularly unacceptable where metric collection programmes are designed to tune a model to the local environment. If data collected in one project only becomes useful in subsequent projects, after the model has been tuned, then the project that collects the data derives no benefit from the exercise.

Similarly, designers and developers are reluctant to sacrifice valuable time to extract data from their own tools and enter it into the model. If causal models, such as the Philips model, are to be effective as practical agile development tools then automated data collection and analysis would seem to be highly desirable.

As can be seen from Table 6-2, there is a disparate set of data sources, possibly used and maintained by quite separate individuals. This presents a problem since there is no single coherent data dictionary across these diverse data sources. For example, a software release identified as “release 1” in the project plan might be known as “phase 1” in the bugs database.

My initial attempt to build causal models focused heavily on this data collection task. The aim was to extract data from multiple data sources and enter it directly into the Philips, or equivalent, models. Timeslices could then be replicated at will, knowing that the quantitative data required could be easily collected and inserted into the model.

To address this, I developed a general purpose Data Extraction, Mapping And Cleansing (DEMC) capability for use in AgenaRisk. DEMC had the following requirements.

1. To extract data from any of the data sources listed in Table 6-2.
2. To map any value from one data source to any value in any other data source. This acts as a substitute for a common data dictionary and as a partial substitute for the lack of referential integrity.
3. To map query results to nodes in BN models. This enables the model to automatically populate evidence from the data sources.
4. To use the query results from one source as parameters for queries in another data source. For example, the project plan might define the phases of a project. These phases would then be used to query the number of defects recorded in the bug database.

Data sources and queries could be defined interactively via dialog boxes, such as the one shown in Figure 6-1, or via XML configuration files (see Appendix H for an example). A demonstration of these capabilities was accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering [83]. A formal description of the data import and mapping capabilities is given in Appendix E.

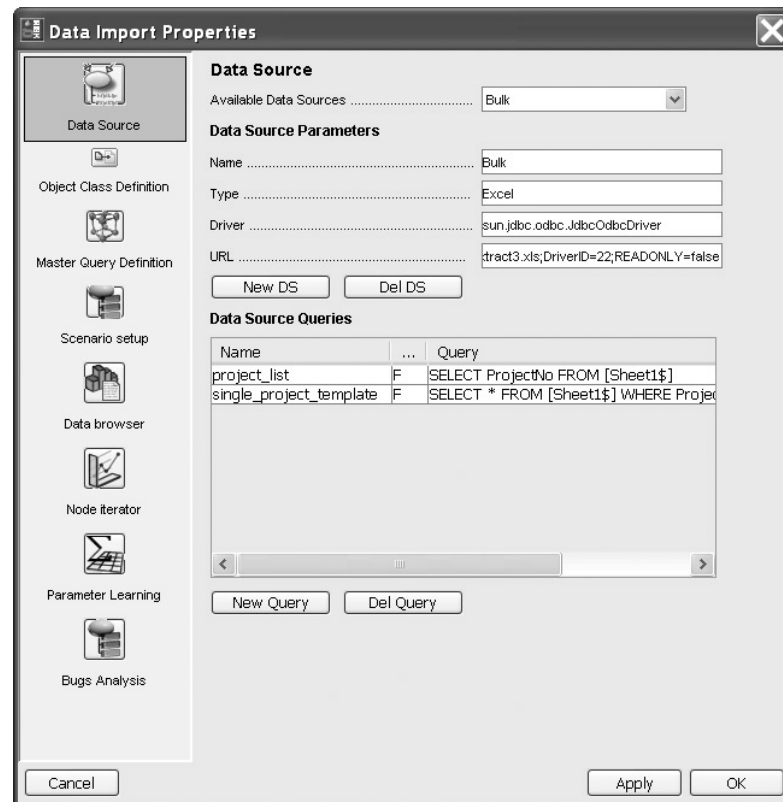


Figure 6-1 DEMC - the Data extraction, mapping and cleansing tool

6.2.2 Problems with Data Collection

The DEMC techniques described in 6.2.1 proved to be sufficiently flexible to extract raw data provided by Philips from a spreadsheet and use it to create multiple copies of the Philips model. It was also used to experiment with extracting and combining data from Microsoft Project and Bugzilla. However, from the XP perspective, it soon became apparent that there were problems with this approach.

1. Qualitative data, other than those items already entered into a spreadsheet, still had to be entered manually.
2. Much of the qualitative data is meaningless in an agile environment.
3. There were problems trying to interpret the meaning of “residual defects”.

The first problem is the simplest to explain. Very little of the qualitative data could be replaced with project metrics. For example, consider the node “Regularity of spec and doc reviews”. This is a ranked node with five possible choices. It specifies whether sufficient specification and documentation reviews were taking place. In order to arrive at this figure in some systematic way, it would be necessary to record the number of such reviews as well as having an algorithm for deciding if that number was sufficient. Further, this data would have to be maintained consistently across all iterations in the agile environment. This is precisely what we wish to avoid: a significantly enlarged metrics collection programme.

Without metric data, most of the data entry in the Philips model remains qualitative and subject to expert judgement. Not only does someone have to enter this data manually (because there are no metrics to import), but the interpretations of terms such as “regularity”, “stability” and “quality” – terms with which the Philips model is replete, must remain consistent across iterations. We cannot make these recurring factors “global” since we wish to preserve the 1st order Markov property for performance reasons.

The second problem is also quite straightforward to explain: much of the qualitative data used in the Philips model simply does not exist in agile projects. The node “Requirements stability” is a good example of this. In a traditional project, stable requirements are a necessary condition for stable, and hopefully more reliable, code. In an agile environment, the requirements are uncertain from the start and are expected to be unstable.

The third problem, the interpretation of “residual defects”, is more complex. There are two reasons why we wish to enter defect numbers into models.

1. In an iterative environment, we may be interested in the residual defect predictions for the most recent phase. This prediction is more accurate if we know the defect counts for previous phases and numbers of defects fixed.
2. Our model may have to learn some parameters associated with defect prediction.

This gives us the problem of interpreting the meaning of “residual defects”. The problem can be broken into two similar sub-problems.

3(a) When do you count a defect?

3(b) When do you stop counting defects?

Problem 3(a) boils down to determining the definition of a defect. Do we only count defects that cause the software to be unusable, or do we also count things that users simply don't like? Does our definition include missing functionality? Are multiple reports of the same defect being included in the data? This is a well-known problem and is discussed in detail in [61]. There are at least two distinct approaches we could take to solving this.

1. We could build additional causal factors into the model to make the definition of a defect unambiguous. For example, we might believe that a poor analysis and design process leads to missing or poorly implemented requirements, whereas inexperienced programmers lead to more instances of program crashes. This would lead us to two distinct types of defects based on different causal chains. Observations would then consist of the two defect types that we had defined and these would have to be recorded separately in the bugs database using criteria defined by the model. This makes the model larger and increases the amount of data that must be collected – both things that we would like to avoid.
2. We could try to learn the definition of a defect from the environment. In this approach, the definition of a defect is “whatever gets recorded in the bugs database”. We then adjust the parameters of the model to automatically make predictions using the users expectation of what constitutes a defect.

This is the first time we have come across this basic choice between causal modelling and learning. It provides a hint that there may be an alternative to building large, comprehensive causal networks to model the agile software development process.

Problem 3(b) is simply a statement of the well-known aphorism that no large piece of software is entirely bug free – there are always more bugs to find. We can however appeal to software reliability theory to perhaps provide us with a solution here.

6.2.3 Software Reliability Theory

The number of residual defects in a piece of software is not easy to estimate. If a piece of software is released, and four bugs have been added to the bug database, this does not mean that there are four residual defects. If the software has received little use then there could be many more bugs remaining. Perhaps if we can model how the

number of defects discovered varies with time, then we can estimate the total number of defects remaining? This is the domain of software reliability theory [146].

The problem of estimating the number of residual defects in software is closely related to the software modality and operational profiles. We adopt the definition of a **system mode** given in [129] p.174:

“A system mode is a set of functions or operations that you group for convenience in analyzing execution behaviour.”

Each mode has a distinctive **operational profile**, where the term “operational profile” is as generally accepted within software reliability theory [129] – essentially this is a probability distribution of the functional usage for a given system mode.

It is useful to introduce three categories of software modality.

- **Single-mode** software has a single operational profile.
- **Multi-mode** software has multiple operational profiles.
- **Modeless** software has no typical operational profile.

With single-mode software, the number of operations it is expected to perform is well-defined and their statistical likelihood is well understood. The operational profile used in testing has a high probability of being similar to the operational profile of the operational system. Because of their well-defined behaviour, they perform the same regardless of who tests the software and regardless of whether the tests are performed in the lab or in the field. Examples of single-mode software might include such elements as software switches, embedded consumer device controllers and avionics measurement devices. A good example of single and double mode systems is given in [121] where Levendel describes an early approach to iterative programming in the telecoms industry and where a major introduction of new functionality lead to a sudden increase in defect reporting.

With their single operational profile, single-mode programs appear to have a finite number of faults where the rate of fault discovery decreases with time. (Note that Musa [146] p.91 recommends that fault recording be measured with respect to program execution time. Outside of controlled laboratory conditions this is almost impossible to achieve. We must normally work with some surrogate, or estimate of execution time). Because of this we might expect them to be ideal candidates for

models such as the Goel and Okumoto (G-O) model [75]. This assumes that software failure can be modelled as a non-homogeneous Poisson process. The resulting estimate for the cumulative number of failures $M(t)$ is given by:

$$M(t) = a(1 - e^{-bt}) \quad (4.13)$$

A brief derivation of this model can be found in [54]. The number a is the total number of faults in the system, while b controls the rate at which the cumulative number of failures asymptotically approaches a .

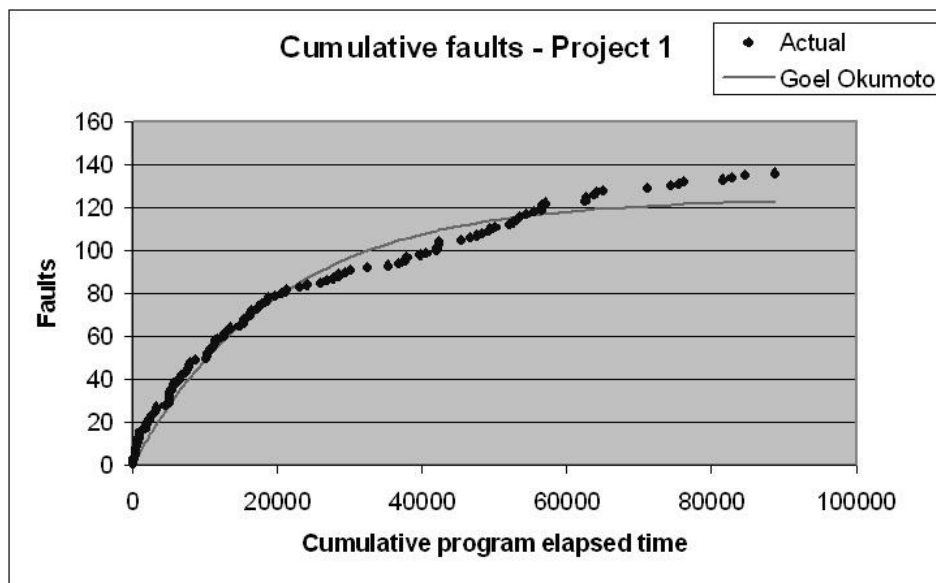


Figure 6-2 Musa fault data - Project 1

Musa [145] collected sixteen high quality data sets for use in software reliability studies. Project 1 (Figure 6-2) shows a clear fit with the G-O model.

The G-O model works reasonably well for single-mode systems. As Whittaker [210] points out:

“Software reliability theory appears to work accurately in telecommunications and aerospace.”

By extension, he seems to imply that its value in other fields is sometimes limited.

Multi-mode programs have multiple uses. For example, an email client will often double as a newsreader. Tests that cover the email operational profile are unlikely to uncover faults in the newsreader functionality. Here, one user of the program might use an entirely different operational profile from another user and so may have quite different perceptions of the program's reliability.

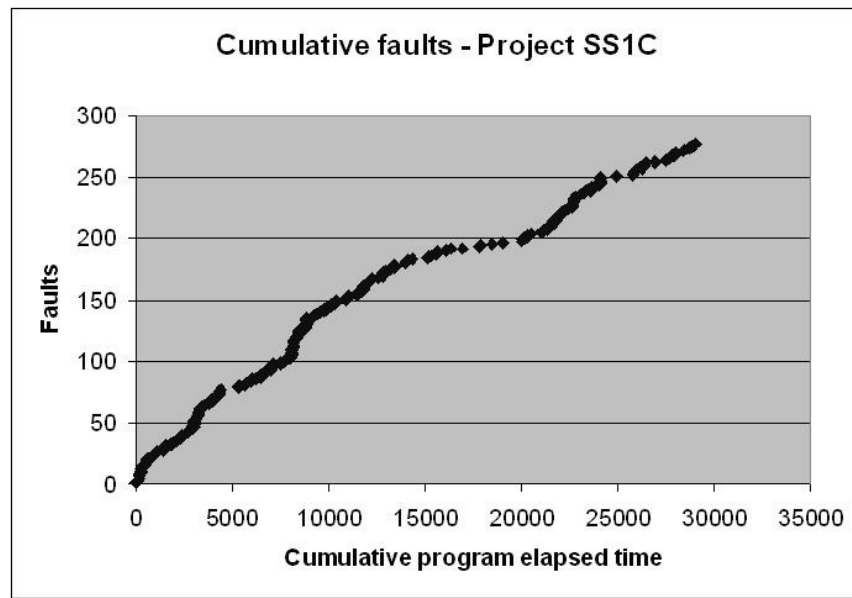


Figure 6-3 Musa data - Project SS1C

Another example of the Musa data is shown in Figure 6-3. In this example it is less clear that the G-O model is being followed. There are several places where the graph looks as if it might be about to level off when the rate of fault discovery suddenly begins to increase again. Failure to recognise the multi-modal nature of this software product could lead to the misinterpretation of any one of these levelling off points as the final value for the post-release defects count.

Modeless programs have no typical operational profiles. Examples include word processors, operating systems, modelling programs etc. Attempts to test such programs in the lab are unlikely to encounter exactly the same conditions as any user experiences in the field. Any given user is equally likely to explore combinations of functionality that have not been explicitly tested. With modeless programs, every new user is likely to experience their own operational profile and consequently to encounter faults which have never been detected before: the more users such a program has, the more bugs will be found.

In building early models of iterative development environments, I had hoped that it would be possible to make estimates of the residual defects using a procedure similar to the following.

1. Use expert judgement to select the modality of the software.
2. Knowing the modality, select an appropriate software reliability model.
3. Having selected a software reliability model, fit it to a time based analysis of the bug database.
4. Use the reliability model to estimate the number of residual defects.

However, it soon became clear that this would be a major area of research in its own right. Limits on the applicability of software reliability models have been known for some time. As long ago as 1987, Dale [54] pointed out:

“Whilst it is possible to estimate current failure rates in fixed usage environments with a fair degree of accuracy on the basis of failure data, it is not possible to relate such estimates to different usage environments.”

Part of the problem lies in the simplistic definition of operational profiles. Whittaker [210] argues that the notion of operational profiles must be extended to take account of software complexity and the operational environment. Gittens, Lutfiyya and Bauer [72] extend this further by including the need to measure the size and complexity of runtime data structures.

In [72] the example happened to be a relational database. Data structures will include results sets, table caches, indexes and so forth. These cannot be predicted in advance and can in principle be of any size and complexity. The same is true of office programs such as a word processor or spreadsheet. Their internal data structures reflect the problem currently being addressed rather than any operational profile which might have been tested prior to release.

Contrast this with single-mode programs. These typically run in embedded devices. Their state is characterised by the state of their internal variables, most of which will have been explicitly created by the programmer. Dynamic data structures are limited both by hardware constraints and by the operational profile of the device.

Even where a program is clearly single mode, it is not always possible to assume a characteristic software reliability model. Figure 6-4 shows the total number of bugs over time in the bug database for a popular open source utility [163]. The utility in

question provides a web based administration utility for the MySQL database. There could be any number of reasons why there is no clear levelling off of this graph. For example, it may simply be that as the tool increases in popularity, many duplicate bugs get recorded (there is no cleansing of the data shown in this graph).

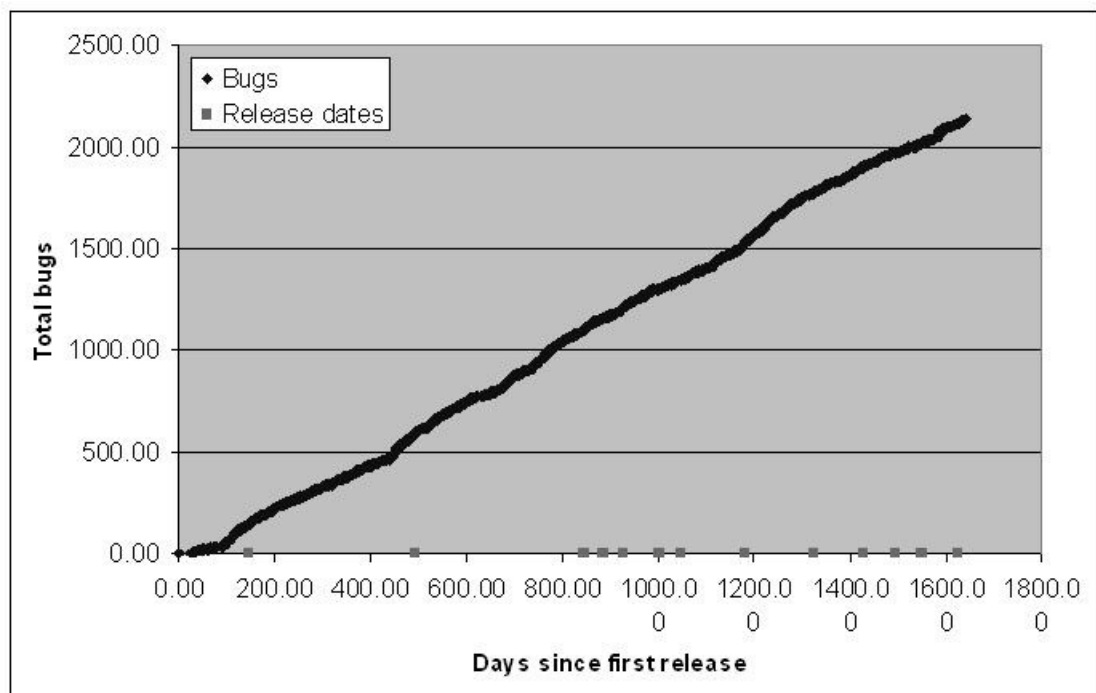


Figure 6-4 phpMyAdmin bug reports

Littlewood argues (see for example [127]) that, given a well defined operational profile and high quality test data, it is possible to retrospectively test reliability models against the data and choose the best model with reasonable accuracy. The problem is that in most situations, neither of these conditions is true. Further, reliability models assume that correcting a software defect does not introduce new defects, an assumption that for the most part cannot be relied upon.

For the moment, software reliability theory does not provide a single well understood reliability model that can be applied to every project, nor does it provide a robust means of *a priori* choosing which reliability model to apply.

6.3 Separating Model and Object Structure

The AgenaRisk tool provides numerous facilities which make it an appropriate tool for building Dynamic Bayesian Nets (DBNs): the easy linking of predefined BNs, its forward only fully factored Boyen-Koller inference algorithm [3.2.4], mixing of discrete and continuous approximation nodes, and its ability to easily incorporate a wide range of distributions and expressions when constructing NPTs.

However, AgenaRisk is primarily a GUI tool - BNs are created and manipulated graphically. This provides a highly intuitive interface which allows BNs to be rapidly constructed by inexperienced users. However, as with all GUI interfaces, it excels when individual changes must be made, but quickly becomes inefficient when many repetitive tasks have to be performed.

The creation of DBNs involves many such repetitive tasks. A typical DBN development lifecycle might look something like this.

1. Create the main timeslice object. This includes creating the basic causal link topology as well as the NPT expressions.
2. Optionally create an initial timeslice to provide some initial input to the model. This is often necessary since the main timeslices contain a set of outputs to feed the next iteration's inputs. The first timeslice can either have default distributions and/or evidence entered, or an initial object can be used to supply these values.
3. Create multiple instances of the necessary Risk Objects and link outputs and inputs together.
4. Test the model.
5. Determine any errors in the mode.
6. Edit the timeslices used in the model and repeat from step 3.

The main problem with this is that changes made to timeslices in step 6 do not automatically get replicated in all timeslices in the model. The model must be rebuilt using the newly edited timeslice. In OO parlance, there is no distinction between classes and objects. We would like to be able to edit a BN class and then have all instances of that class include the new changes.

A similar situation exists when building hierarchical statistical models. Here, a small number of nodes encapsulate parameters which define a statistical distribution. The values of these parameters are learned by creating multiple evidence nodes which

cause the parameters to be updated. Here, many copies of the evidence nodes must be created which replicate their own internal structure as well as their relationships with the parameter nodes and with each other.

Agena have created an API to allow the AgenaRisk engine to be accessed from Java code. This is a powerful facility which makes it relatively simple for Java programmers to create arbitrarily large hierarchical or dynamic BNs. This however suffers from its own problems.

1. The API sacrifices one of AgenaRisk's greatest strengths, namely its powerful user interface. In many cases this may be exactly what is wanted, however to make models that are easily understood by non-specialists the GUI is essential.
2. The amount of code required to create non-trivial models quickly becomes quite large.
3. Users need to know the Java programming language and have a suitable development environment to support it.

To overcome some of these difficulties, I have added a scripting language to the AgenaRisk toolset. Scripts consist of short text files with lists of commands which allow Risk Objects to be created, nodes to be copied, links to be created or deleted, and evidence to be entered. This scripting capability provides many of the facilities needed to construct DBNs and hierarchical models, without the need to either sacrifice the GUI interface or setup a Java development environment.

The DBN development lifecycle remains the same, except that step 3 is now performed by the script. Step 4 can also be placed in scripts. This automates two of the most time consuming aspects of model development, greatly reducing the time required to refine and test models. When a change has to be made to a node in a repeated timeslice, it is a simple matter of editing the script and re-running it. Models can be rebuild in a matter of minutes, rather than hours, and with a greatly reduced probability of mistakes being made.

The full syntax of the scripting language, together with some example scripts, can be found in Appendix C.

6.4 Summary

We have seen that the naïve approach, of simply taking a model like the Philips model and replicating it over multiple timeslices leads to several problems.

1. The model becomes extremely large and memory intensive.

2. Large amounts of qualitative data need to be supplied with each iteration.
3. There are problems in estimating the number of residual defects for use in defect prediction models. Software reliability theory does not help us with this problem.
4. We have highlighted the need to build BN timeslices independently of the DBN which ties them together.

We saw in section 6.2.2, that there were two ways to deal with the ambiguous definition of defects. One was to construct causal models of the various defect types and record them separately in the defects database. The other was to learn what was meant by a defect in a particular project environment. We will see in the next chapter how this latter approach greatly simplifies our software process models and leads to a practical agile development model of XP project velocity.

7 A Learning Project Velocity Model

In Chapter 2 we showed why traditional software process models, particularly regression based models, were often inadequate and provided no natural measure of risk assessment. We then went on to describe Bayesian nets in chapter 3 and explained why they were relevant to software projects. Examples of existing Bayesian net models were then presented in chapter 4. Initial attempts to simply replicate these models in an iterative environment faltered for the reasons described in Chapter 6. However, in the process of trying to develop these early models, it became clear that there was an alternative to purely causal models. That alternative was to create a learning model.

The main novel contribution of this chapter is to introduce and validate dynamic Bayesian nets as a means of modelling iterative software development. PV data is collected from the first iteration in any XP project. This is incorporated into the model, enabling it to learn key parameters and increase the confidence of its predictions in subsequent iterations. We show that, with very little data, it is possible to correct the model's prior assumptions and quickly produce accurate models of PV with associated risk assessments.

First, we must discuss the difference between causal and learning models.

7.1 *Causal vs. Learning Models*

All of the software process models that we have seen so far, attempt to model the causal relationships between different factors in the software development process. So, for example, experienced developers will be more productive, producing more code in a shorter period of time, but they will be less productive if the requirements are sufficiently complex. There are a number of possible problems with this approach.

1. We may get the causal relationships wrong. Causal models rely on the expertise of the modeller to include all of the relevant factors with their proper causal relationships and the strength of those relationships. (Although it should be noted that considerable research has been performed in learning the topology of BNs from databases. See for example Neapolitan [149] (Ch. 8). These techniques are not applied in causal models of the software development process due to the lack of a sufficiently comprehensive database.)

2. Much of the data is subjective. Clear rules must exist for selecting a particular subjective value. Those who enter the observations must be aware of any assumptions that the modeller has made. For example, the node “Programmer capability” in Figure 4-6, is intended to mean “Programmer capability relative to what is required on the project”. If the programmer is inexperienced, but the programming task is simple, then their lack of experience may not be a problem. In a large causal model, such as the Fenton, Krause and Neil models of Chapter 4, there can be a large number of these rules.
3. In an iterative environment the interpretation of a node must be consistent across iterations. If the person who enters data into the model changes from one iteration to the next, then their assessment of some subjective judgements might change also.

An alternative to causal modelling, is to learn about the local environment. So instead of trying to ascertain all the possible influences on productivity, say, we simply measure the actual productivity. This looks like it suffers from the same problem we highlighted in section 6.2.1, namely collecting metrics in one project which are only of value to a later project. However, in an iterative environment this is not the case. We can learn about productivity, for example, from early iterations and then use this to modify our initial assumptions (which can be based on industry averages).

Note that these two approaches, causal modelling and dynamic learning, are not mutually exclusive. We can have a learning model which still includes some causal factors. Conversely, we can have a largely causal model which learns some of its parameters. The models presented later in this chapter are primarily learning models, but they still include some causal elements. A good discussion of causality and learning in probabilistic models can be found in Krause [116]. Before looking at these models in detail, it is useful to consider the learning technique that we will use.

7.2 Why model Project Velocity?

In section 5.5 we defined Project Velocity (PV) as the sum of the ideal engineering time estimates of user stories completed in the last complete iteration. PV is therefore a measure of how much functionality a team can deliver in an iteration. We have shown in section 5.5.2 why PV is so important to project planning and how it can naively be used to predict project timescales. We also pointed out the pitfalls of

ignoring trends in PV and in ignoring project risk. We will now build a DBN model of PV which addresses these concerns.

Project Velocity (PV) is the one management metric that is nearly always available in XP. By modelling PV, as opposed to some other effort based metric, we are therefore leveraging data collection which we know already takes place. We set the following key requirements that the model must satisfy.

1. It must monitor and predict PV, taking into account the impact of relevant process factors.
2. For computability reasons, the core model must be very small. This enables it to be replicated multiple times in order to represent the multiple iterations of an agile development environment.
3. The model must be able to handle different types of data for different environments. In particular, the model must handle key XP practices, while being dependent on none of them.
4. The model must be capable of replicating empirical behaviour. In particular, many projects report low initial productivity, gradually rising on subsequent iterations [3][6][215].
5. The model must learn from data, either as a result of observations or as a result of expert judgment entered as evidence.
6. It must give useful and clear advice to managers.

7.3 Process factors

It is useful to distinguish between *total iteration effort* and *actual productive effort*. Total iteration effort (“total effort”) is simply the product of the number of people available on the project (possibly non-integer) and the number of days in the iteration. Actual productive effort (“productive effort”) is that part of total effort that directly contributes to user story completion. This includes designing, coding and testing activities. Other project overheads, such as team meetings, administrative duties, mentoring and learning, while they may be perfectly constructive uses of time, are not counted in productive effort.

This definition of productive effort is intended to mimic the definition of PV. The sole difference between the two is that productive effort is the *actual* effort which goes towards delivering functionality (just as if it had been measured using timesheets – see section 5.5), whereas PV is the *estimated* effort for the same work.

To model the relationship between total iteration effort and actual productive effort, we introduce a single controlling factor which we call Process Effectiveness, e . Process Effectiveness is a real number in the range $[0,1]$. A Process Effectiveness of one means that all available effort becomes part of the productive effort.

The Process Effectiveness is, in turn, controlled by two further parameters: Effectiveness Limit, l , and Process Improvement, r . The Process Improvement is the amount by which the Process Effectiveness increases from one XP iteration to the next. To allow for failing projects, the Process Improvement can take on negative values.

The Effectiveness Limit recognizes the fact that there are often limits to how productive a team of people can be. Effectiveness Limit is therefore the maximum value which the model allows Process Effectiveness to take.

Note that all of this relies on minimal assumptions: effort either contributes to completed User Stories, or it does not. The ratio between productive effort and total effort exists whether we call it Process Effectiveness or not. This ratio varies between iterations and has a limit, even if the limit is unity. As the core model contains variables based only on these factors, it too is based upon minimal assumptions.

7.4 Bayesian Net Model of Project Velocity

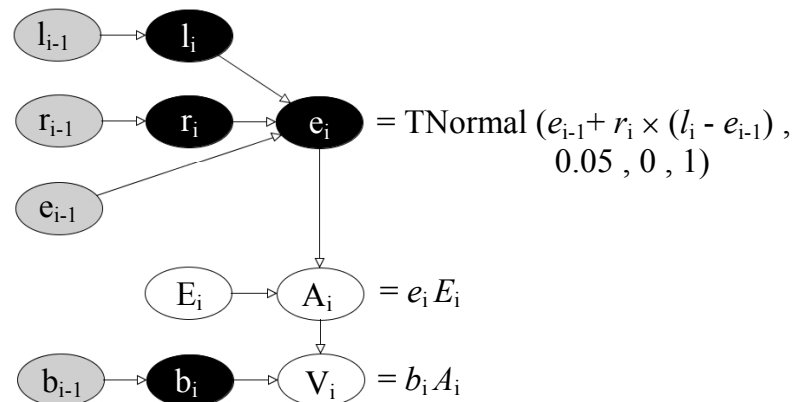


Figure 7-1 Project velocity model

The BN used to model project velocity is shown in Figure 7-1. Table 7-1 summarizes the model variables for the BN. Measures of effort are denoted by capital letters. All other variables use lower case letters. Subscripts are used to denote a specific XP iteration. For example V_2 denotes the velocity in iteration 2. Where the iteration is not important, we drop the subscript and refer simply to V .

Symbol Meaning

d_i	Number of working days in iteration i . $d_i = 0, 1, 2, \dots$ This is an integer value.
p_i	Number of team members in iteration i . This can be fractional if one or more people do not work full time on the project. $e_i \in [0, \infty)$.
s_i	Productive effort to date. $s_i = s_{i-1} + V_i = \sum V_i$, $s_i \in [0, \infty)$.
E_i	Iteration effort in man-days. $E_i = p_i \times d_i$, $E_i \in [0, \infty)$.
U_i^j	Estimated effort of j^{th} user story in iteration i . $U_i^j \in [0, \infty)$.
A_i	Actual productive effort in iteration i . $A_i = E_i \times e_i$, $A_i \in [0, \infty)$.
V_i	Project Velocity in iteration i . $V_i = \sum_j U_i^j$, $V_i \in [0, \infty)$.
b_i	Estimation bias. $b_i = V_i / A_i$, $b_i \in [0, \infty)$.
e_i	Process effectiveness in iteration i . $V_i = E_i \times e_i$, $e_i \in [0, 1]$.
f_i	Load Factor in iteration i . $f_i = E_i / V_i = 1 / e_i$. Used to estimate timescales.
l_i	Effectiveness limit. The maximum value that the e_i can take, $l_i \in [0, 1]$.
r_i	Process improvement. $e_i = e_{i-1} + r_i \times (l_i - e_{i-1})$, $r_i \in [-1, 1]$.

Table 7-1 Symbol definitions

When we wish to distinguish between a model prediction and a measured value, we will use an underscore to denote the measurement. So if V_3 is the predicted value for the velocity at iteration three, then \underline{V}_3 is the measured value.

Not all of the variables shown in Table 7-1 are shown in Figure 7-1. Several of the variables are included only to make the definitions of others more rigorous (d , and p). Some exist to relate the model to XP concepts (f and U), and others to relate the model to management concepts (s).

Figure 7-1 shows a single timeslice but with the link nodes from the previous timeslice shown lightly shaded. The link nodes to the next timeslice are shaded black. Figure 7-2 shows the same model, this time “rolled out” as a three iteration DBN (link nodes are shaded).

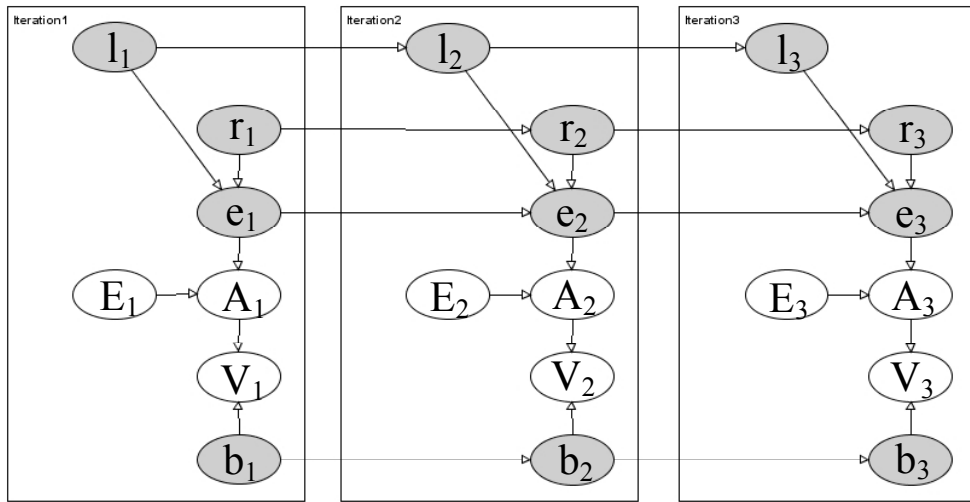


Figure 7-2 PV Model as a DBN

The process effectiveness limit (l_i) and rate of process improvement (r_i) are the key parameters in this model. Between them they control the process effectiveness node, which in turn controls the velocity node. It is important that the model is capable of adjusting these parameters as a result of entering data about the project. In particular, the model must respond to observations of \underline{V}_i .

7.5 Iteration Model

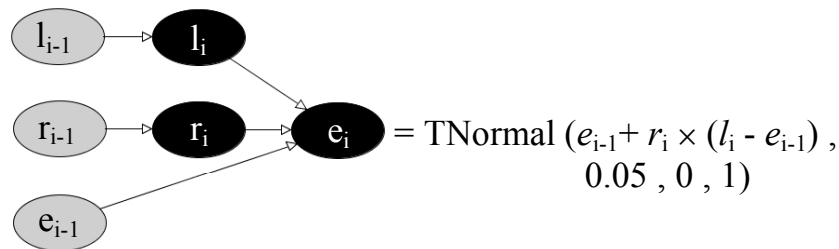


Figure 7-3 Fragment 1 - Process effectiveness nodes

The BN shown in Figure 7-1 is best thought of as comprising three distinct fragments. Fragment 1 controls the Productive Effort (Figure 7-3). A single variable, Process Effectiveness (e_i), is assumed to determine the Productive Effort. High Process Effectiveness means a high Productive Effort and a correspondingly high velocity. Process Effectiveness increases or decreases based on the value of the Process Improvement (r_i). It is constrained to the range $[0, l_i]$.

The CPD of l_i is a function of l_{i-1} . In this case l_i is set equal to l_{i-1} . The process effectiveness limit (l_i) is really a single variable which is global to all timeslices. Copying it between timeslices allows us to preserve the first order Markov property. Similarly r_i is just a copy of r_{i-1} . This is the same process of parameter learning that we saw in section 3.2.3.

The process effectiveness in the current iteration is just the process effectiveness from the previous iteration plus the process improvement times the difference between the previous process effectiveness and the effectiveness limit. For positive r_i this gives a rising e_i which rises asymptotically towards l_i . The whole expression for e_i is surrounded by a TNormal expression with a variance of 0.05. As explained in section 7.8.6, this was needed in order to improve the effectiveness of the approximate inference algorithm.



Figure 7-4 Fragment 2 - Effort nodes

Fragment 2 contains the "effort" nodes (Figure 7-4). It combines the total Iteration Effort (E_i) with the process effectiveness (e_i) to create the actual Productive Effort (A_i). Note that we do not expect A_i to be observed in real projects.



Figure 7-5 Fragment 3 - Project Velocity

Fragment 3 holds the project velocity (Figure 7-5). Velocity can either be predicted by the model (V_i), or once an iteration is completed, it can be entered as evidence (L_i) and used to learn the model parameters. The bias, b_i , allows for any consistent bias in the team's effort estimation. If there was no bias then the productive effort, A , would be the same as V and there would be no need to distinguish between the two.

7.6 Setting the initial conditions

An initial timeslice, Iteration 0 (shown in Figure 7-6), is used to set the initial model conditions.

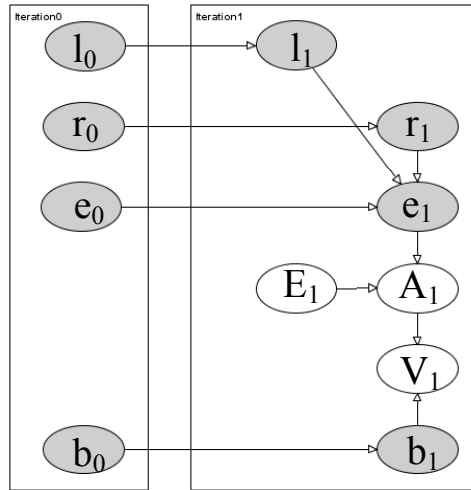


Figure 7-6 Initial Velocity model

For iteration 0, the prior distributions of the input effectiveness limit (l_0), process improvement (r_0) and process effectiveness (e_0) are all set to be normal distributions, with variances of 0.1 and means of 0.8, 0.2 and 0.3 respectively. These values are based on a controlled case study by Abrahamsson and Koskela [3], where process effectiveness varied between 0.4 and 0.75. We have simply extended this range slightly and chosen r_0 so that the lowest to highest transition can take place within four iterations.

The prior of the estimation bias (b_0) is set to a log normal distribution with a mean of approximately 1.0, and a variance of 0.1 (the expression “Log Normal 0 0.3” shown in the script in Appendix F, gives the figures for the underlying normal distribution). The log normal distribution follows from the fact that the bias cannot be less than zero but has no upper bound. For example, a pessimistic bias, where estimates are 2 times the actual, results in a bias of 2, whereas an optimistic bias results in a bias of 0.5. This distribution is confirmed empirically, for example by Little [126].

The estimation bias reflects the cumulative bias of the whole team. In order to significantly deviate from a value of one, there must be a *systematic* tendency for the team to overestimate or underestimate the size of user stories (for example, due to a different choice of measurement units). As such, the variance of this node is not directly related to the estimation bias of individuals, as reported by Jørgensen et. al. [102][103][137]. The variance has therefore been chosen to be considerably smaller than typical values reported by Jørgensen.

The choice of these priors is discussed further in the “Conclusions and Discussion” section at the end of this chapter.

Evidence is entered in all of the E_i nodes so the prior distributions of the E_i nodes have no effect.

7.7 Model Behaviour

Figure 7-7 shows the predicted values of the PV for a hypothetical project with 10 iterations and 50 hours of effort available in each iteration (i.e. $E_i = 50, i = 1, \dots, 10$). The central dotted line is the mean, with the outer dotted lines showing \pm one standard deviation. The solid line is the median value. This is based solely on the model’s initial conditions.

The Process Effectiveness increases with each iteration by an amount equal to the Process Improvement. It flattens out as it begins to hit the Effectiveness Limit. As can be seen from the graph, this leads to the PV starting fairly low and gradually increasing with each iteration. Being able to model and predict this type of behaviour was one of the main objectives of the core model.

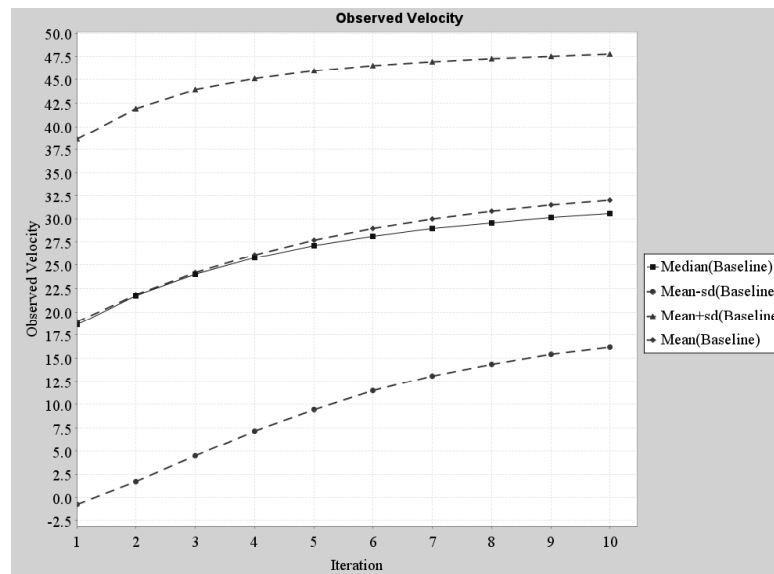


Figure 7-7 Project velocity values V_i – median, mean, mean \pm 1 SD

This is our “Baseline” scenario, with no PV evidence entered into the model. By entering PV evidence, we can construct various alternative scenarios and compare the learned parameters and predicted values of future PV. The values shown in Table 7-2 were used to construct three such scenarios, all based on 50 hours of available effort per iteration. No values were entered for V_9 or V_{10} , allowing the model to predict these

values. The three scenarios represent projects that are: failing, performing as expected, or progressing with great success. We refer to these as the “Failing”, “Average” and “Success” scenarios respectfully.

Note that the “Success” scenario uses deliberately unrealistic figures in order to test the range of the model.

Scenario\PV	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
Failing	2	3	3	4	4	3	4	4
Average	20	25	27	28	28	29	30	31
Successful	200	205	210	215	219	223	225	227

Table 7-2 PV values for three scenarios

7.7.1 Parameter Learning in Different Scenarios

Figure 7-8 shows the resulting distributions of the bias node, b_{10} . There are four distributions, one for each scenario. The “Failing”, “Average” and “Baseline” scenarios have mean values close to one, although both the Failing and Average scenarios have reduced variances compared to the baseline. The reduced variances are to be expected from scenarios where evidence has been entered.

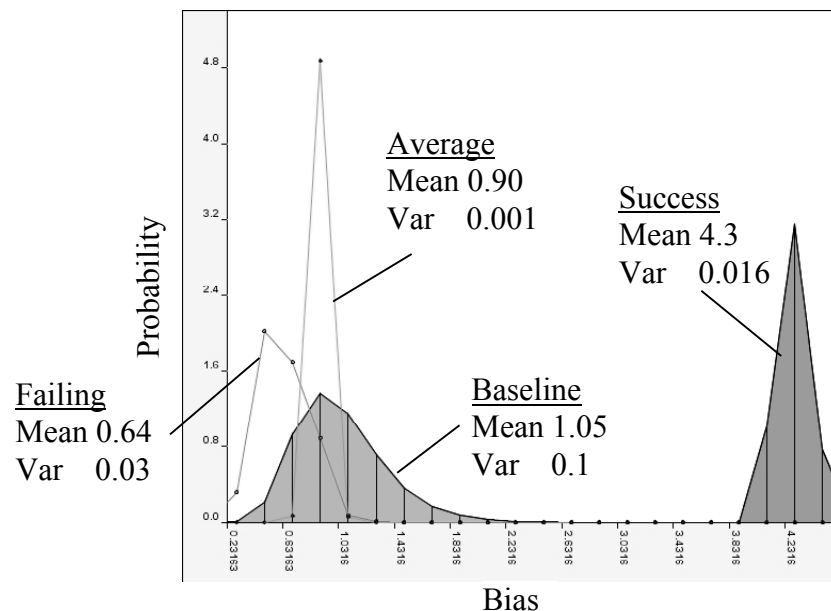


Figure 7-8 Bias distribution iteration 10, b_{10}

In Figure 7-7 the Baseline scenario predicted values for V_1 to V_8 in the range 18-30. However the Success scenario entered evidence in the range 200-227, indicating that

the project team has done 200-227 estimated IEDs in a single iteration with only 50 man-days of effort. Clearly this can only come about if their estimates are significantly biased, and indeed, the model suggests that the bias in this case has a mean value of 4.3. This only accounts for part of the high PV values however. The remainder is accounted for by an increased effectiveness limit (Figure 7-9) which allows a greater process effectiveness.

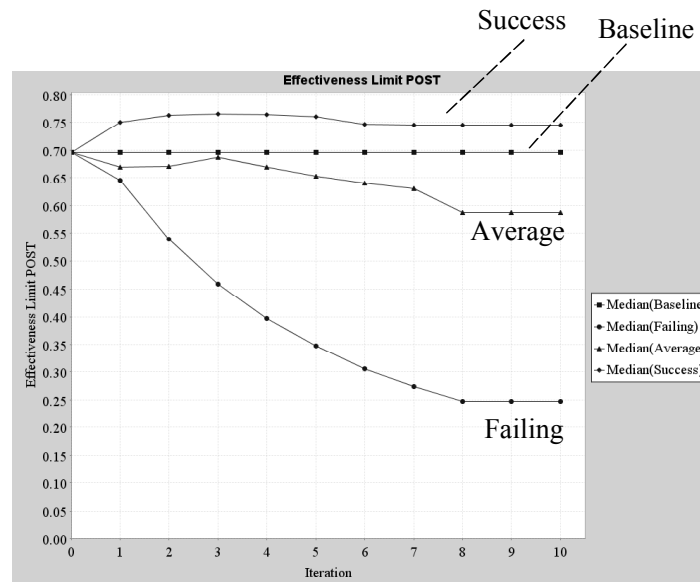


Figure 7-9 Effectiveness Limit l_i , median, 5 iterations

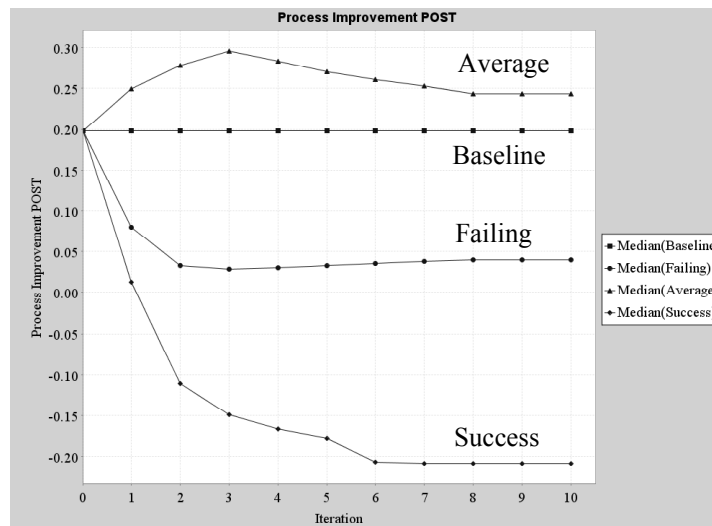


Figure 7-10 Process Improvement r_i , median, 5 iterations

As we might expect, the Failing scenario shows a poor effectiveness limit and a very small improvement in process effectiveness (Figure 7-10). Surprisingly, the success scenario shows an even worse process improvement. However, this is because

the model is forced to assume a very high process effectiveness in the initial iterations. The values provided are so far outside the normally expected range that the model is continually trying to compensate by bringing the process effectiveness back down again. By iteration 6 the process improvement finally begins to stabilize.

Both the Effectiveness Limit (Figure 7-9) and the Process Improvement (Figure 7-10) change as evidence is entered in the first eight iterations. The model therefore learns as new evidence is entered and changes its predictions accordingly.

Figure 7-11 shows the behavior of the Bias node, b_i , in the Average scenario. The central dotted line, which is almost co-incident with the solid line, shows the mean and median values respectively. The outer dotted lines show the mean ± 1 standard deviation (SD). The SD gets smaller as more evidence is entered into the model. This illustrates that, not only does the model learn the values of its parameters, but the uncertainty in those values decreases as more evidence becomes available.

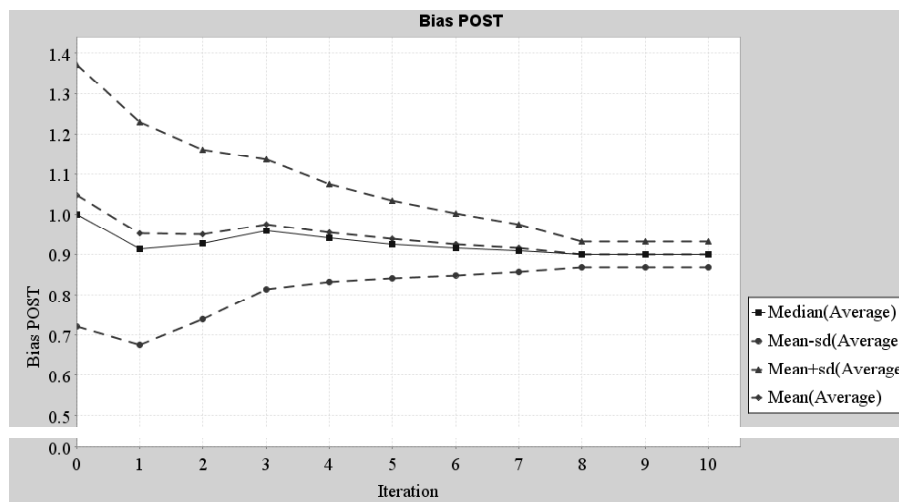


Figure 7-11 Bias b_i , Average scenario, median, mean ± 1 SD

7.7.2 Indicator Nodes

XP practices cannot be categorized as simply being “implemented” or “not implemented”. There are degrees to which various practices are adopted. For example, a team may choose to program in pairs for complex parts of the code and program individually when writing routine code. It is important therefore that XP practices are represented by nodes with a sufficient range of states to reflect the degree of variation of that practice within the project. Boolean nodes are not sufficient for this.

An indicator node for the Effectiveness Limit is shown in Figure 7-12: the

“Collective ownership” node. This is the extent to which collective code ownership is practiced. It is a ranked node, consisting of five discrete values ranging from Very Low to Very High. Ranked nodes allow the user to enter a range of values for “Collective Ownership”. The probability of these five values is derived from a truncated normal distribution whose mean is l_i , and whose variance is arbitrarily set to 0.1. This distribution ensures that a high degree of collective ownership leads to a high effectiveness limit. The variance determines the strength of the relationship. More information on ranked nodes and the use of the truncated normal distribution can be found in section 3.1.2.

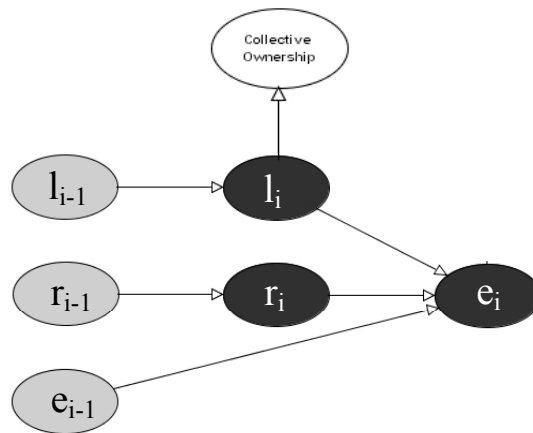


Figure 7-12 The "Collective Ownership" indicator node

With no evidence, the node plays no part in the model, and its parent, l_i , remains constant from one iteration to the next (the “Baseline” scenario). However, when we set the value of “Collective ownership” in each iteration to "Very High" (the “High” scenario) then the situation changes. The evidence back propagates to l_i . Because of the learning mechanism described in section 7.1, the effect is cumulative and the mean value increases across iterations. The difference is shown in Figure 7-13.

Values entered into this node are examples of expert judgment. Just as we saw with the causal BN models in chapter 4, it is easy to combine subjective judgements, such as the extent of “Collective ownership”, with numeric values such as total iteration effort.

Two other scenarios are also shown, one where the Collective Ownership node is always set to “Very Low” (the “Low” scenario) and a slightly more realistic case (the “Mix” scenario). In the Mix scenario, Collective Ownership starts off “Very Low”. However management realise that there is a problem and take steps to improve

collective ownership. By iteration 4 Collective Ownership improves to “Medium” and by iteration 6 it achieves a “High” value.

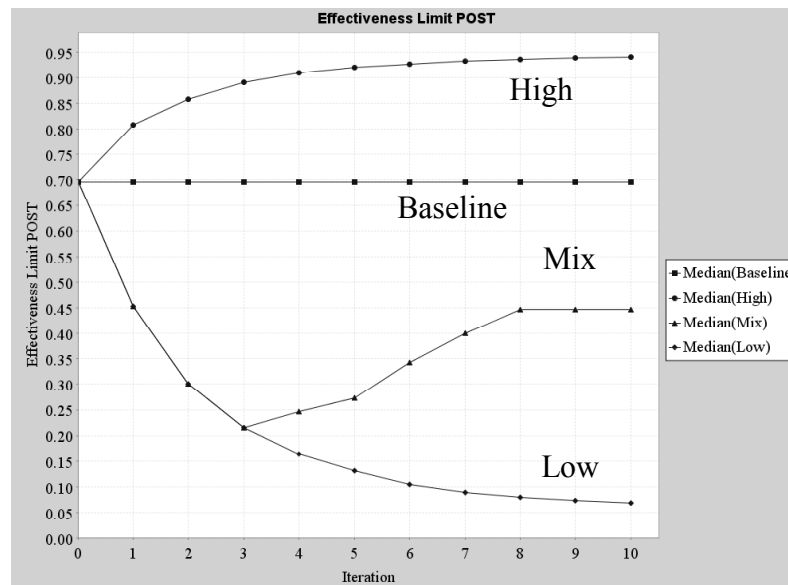


Figure 7-13 Effectiveness Limit I_i with and without indicator node evidence

The extent to which XP practices are implemented can therefore have a dramatic effect on the model parameters, which in turn propagates through to the model’s predictions.

It is not necessary to include all XP practices as indicator nodes in all iterations. If a practice, such as pair programming say, is consistently maintained at the same level in all iterations, then its effect will be included in the learned values of the model parameters. Only practices which affect project velocity and which vary significantly between iterations, need to be included as indicator nodes.

7.8 Model Validation

In this section we apply the model to an industrial case study (section 7.8.1). The model learns from the initial data entered from the project (section 7.8.2) and adjusts its predictions once beneficial XP practices are taken into account (section 7.8.3). Section 7.8.4 provides an example of how the model can be calibrated for a specific XP practice. Finally, in Section 7.8.5 the model provides predictions for the time taken to deliver a fixed amount of functionality. These are in good agreement with the actual functionality delivered.

7.8.1 The Motorola Project

Williams, Shukla and Anton [215] provided a detailed description of an XP project developed at Motorola. The project was developed in a series of eight iterations of between two and three weeks duration. The number of people on the team varied from three to nine over the duration of the project. The full data set is shown in Table 7-3.

i	1	2	3	4	5	6	7	8
d_i	15	15	15	16	12	10	8	10
p_i	3	3	6	6	7	7	9	4
E_i	45	45	90	96	84	70	72	40
\underline{V}_i	9	13	35	30	40	40	36	20

Table 7-3 Motorola project data

The definition of Project Velocity used by the Motorola team corresponds to what we have called Process Effectiveness. We will continue to use the definition given in Equation 5-1. The values for \underline{V}_i given in Table 7-3 have been calculated using our definition.

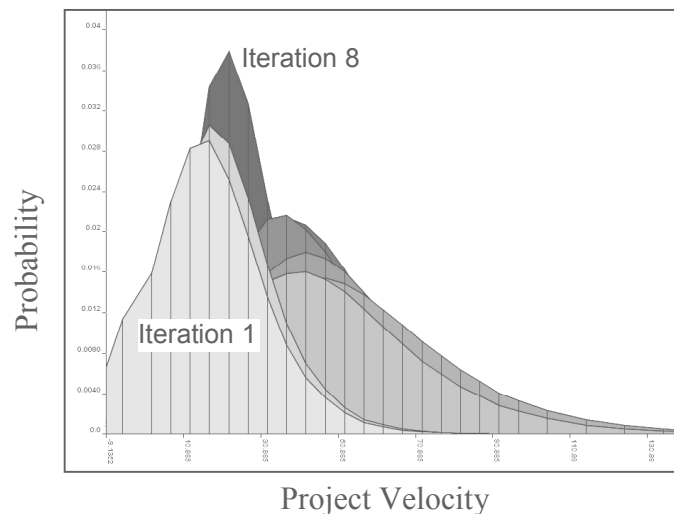


Figure 7-14 Distributions for V_i , one per timeslice

Initially we simply enter values for E_i into the model (no values for \underline{V}_i entered). Figure 7-14 shows the resulting marginal distributions which are generated for the V_i node. There is one distribution for the node in each timeslice.

The median values from the V_i distributions are shown in Figure 7-15 (the “Predicted” graph). Actual values for \underline{V}_i are shown in the same figure for comparison (the “Actual” graph). The large “Actual” dip in iteration 4 is put down to a post-Christmas malaise by the Motorola team. (Surprisingly, given the extent of the dip in

productivity, this is a phenomenon which does not appear to have been extensively studied.)

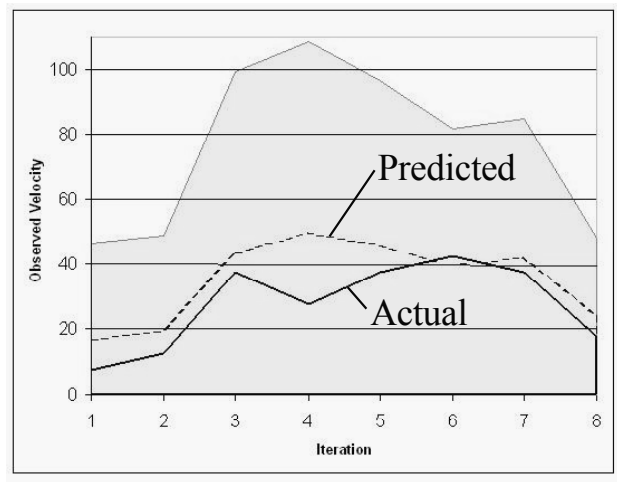


Figure 7-15 Predicted vs. actual Motorola V (medians). Actual values are bold, predicted values are dashed. The shaded area shows predicted medians ± 2 standard deviations.

7.8.2 Parameter Learning

There are a number of problems with the predicted values in Figure 7-15. The most obvious is that, apart from iteration 6, the predicted values are consistently too high. In this section we demonstrate how the model can learn from real project data and quickly improve the accuracy of its predictions.

The effect of this learning process can be seen by taking the “Predicted” scenario and entering V_1 observations for completed iterations. As each new piece of information is entered, back propagation takes place, causing the distributions for the model parameters to be updated. These updated parameter distributions then affect the predictions of future iterations.

The graphs in Figure 7-16 show the change in predicted values when V_1 and V_2 have been entered. The whole of the “Predicted” graph moves to lower values as the model learns from the observations. The predictions for V_3 and V_4 improve as a result. However, the predicted values for V_5 , and V_6 are worse.

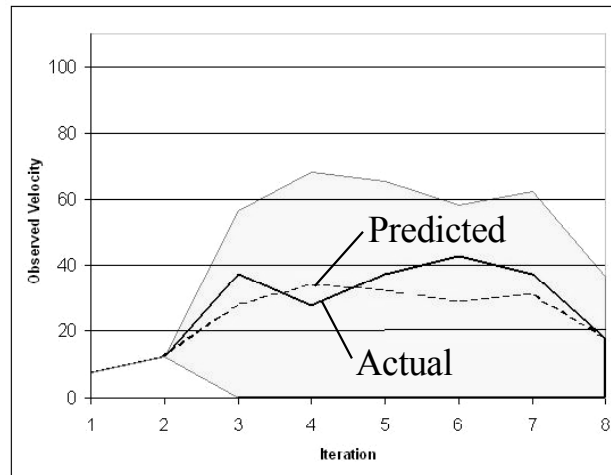


Figure 7-16 Predicted and actual V_i observations. Actual values are bold, predicted values are dashed. The shaded area shows predicted medians +/- 2 standard deviations.

We can examine this more quantitatively by calculating the Magnitude of Relative Error (MRE) of the model before and after learning takes place. The results, calculated from:

$$MRE_i = \frac{|V_i - \text{Median}(V_i)|}{V_i} \quad \text{Equation 7-1}$$

are shown in Figure 7-17. (We use median values from the PV distributions rather than means. The large range of the model causes distributions to grow “tails” which skew the mean values.) Taking the Mean MRE (MMRE), the MMRE improves from 0.34 before learning to 0.16 after learning. The variance of the model’s 8 MRE values also decreases from 0.08 to 0.004, showing that there is greater consistency in its predictions.

The Williams, Shukla and Anton paper [215] points out that various XP practices were implemented more effectively in later iterations. In the next section, we show how this can be incorporated into the model.

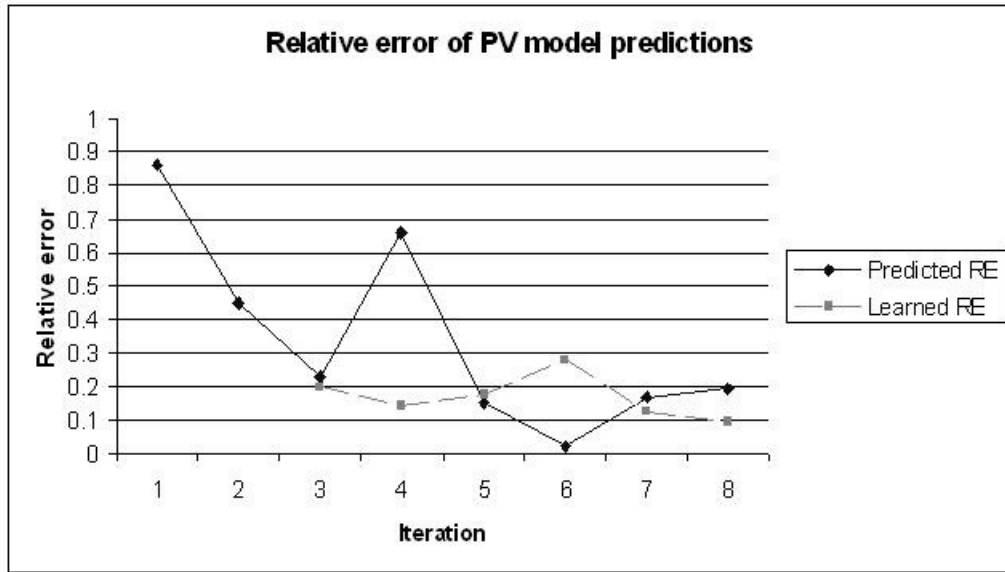


Figure 7-17 Relative error of model before and after learning

7.8.3 “Onsite customer” as an Indicator Node

An indicator node for the Effectiveness Limit is shown in Figure 7-18: the “Onsite Customer” node. This is the extent to which an authoritative customer was available to answer questions about requirements and provide feedback on development. It is a ranked node, consisting of five discrete values ranging from Very Low to Very High. These discrete values define five equal, discrete partitions of the real number range [0,1].

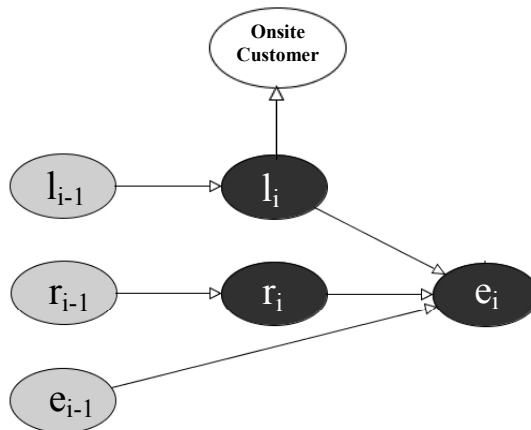


Figure 7-18 The "Onsite Customer" indicator node

The probability of these five values is derived from a truncated normal distribution whose mean is l_i , and whose variance is set to 0.1. This distribution ensures that a

high degree of customer input leads to a high effectiveness limit.

It is important to emphasize that the values entered into the “Onsite Customer” node must be relative to the need for customer input. If the project team have developed similar projects for this customer in the past, or are themselves experts in the application domain, then constant customer input may not be useful. In these circumstances a “Very High” value for “Onsite Customer” might be appropriate, even if the customer is not physically present, but was still able to provide input when needed.

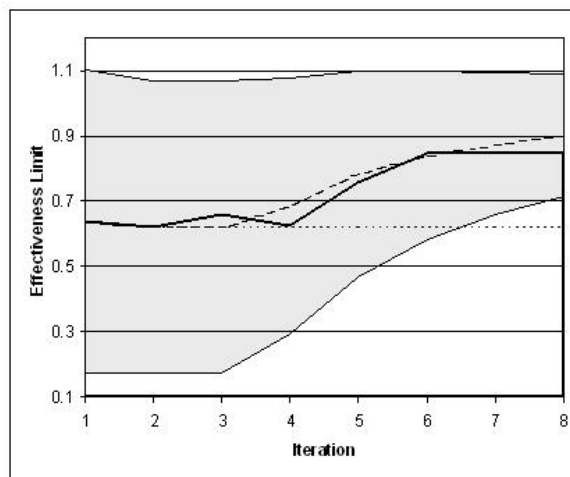


Figure 7-19 Effectiveness Limit with and without indicator node evidence. Actual values are bold, predicated values (after “Onsite Customer” evidence) are dashed. The shaded area shows predicted medians +/- 2 standard deviations. The dotted line shows the learned values without “Onsite Customer” evidence.

Figure 7-19 shows how the indicator node’s parent is affected by changes in its values. The median Effectiveness Limit when only effort data has been entered is simply a straight line, since no learning takes place (not shown). When all the V_i data is entered, then the Effectiveness Limit varies throughout the project (the solid, bold curve). The dotted curve shows the Effectiveness Limit that is learned when only V_1 and V_2 have been entered as observations. This is the curve which is responsible for the modified predictions shown in Figure 7-16.

At the start of the 4th iteration the Motorola team’s access to their customer improved and from the 5th iteration onwards the team had constant access to their customer onsite. The “Onsite Customer” indicator node was therefore set to “High” for the fourth iteration and “Very High” for the subsequent iterations. The result is the dashed curve. It shares the same values for the Effectiveness limit as the “Learned”

curve, until the values for the Onsite Customer indicator node are modified.

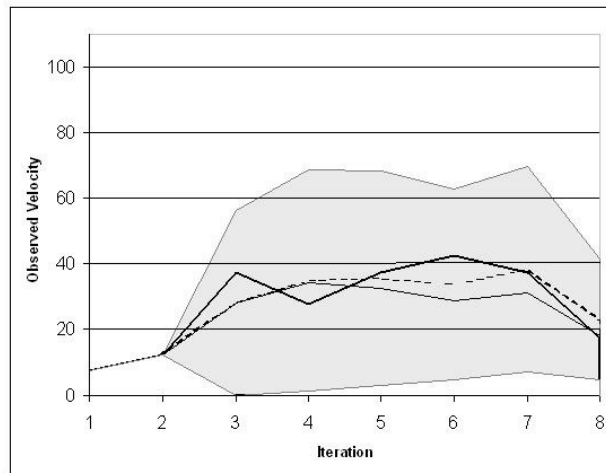


Figure 7-20 V with and without Onsite Customer evidence. Actual values are bold, predicted values are dashed. The shaded area shows predicted medians +/- 2 standard deviations. The solid grey curve shows the predicted values before Onsite customer evidence.

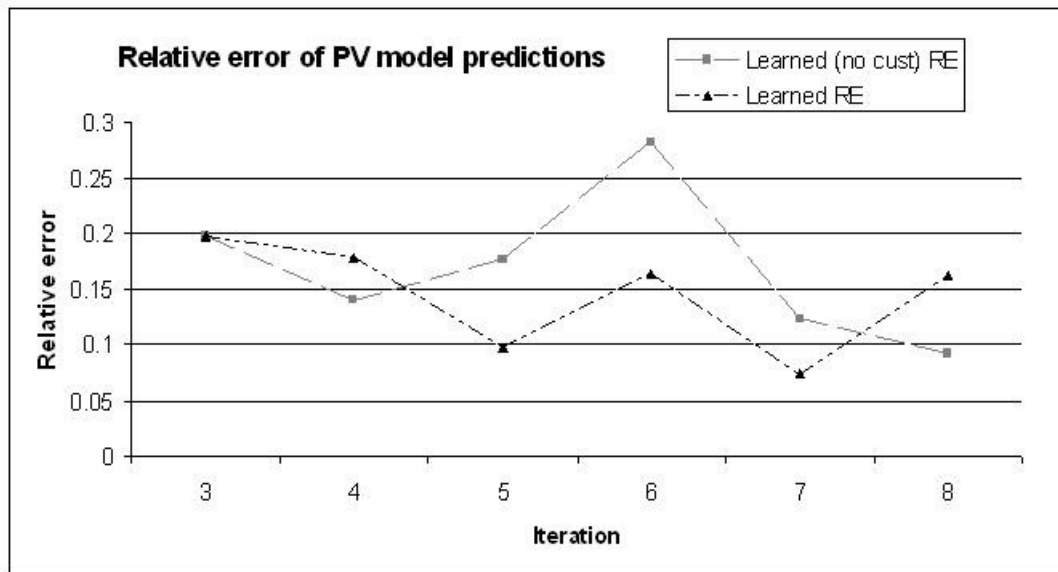


Figure 7-21 Change in relative error with onsite customer

The result of entering indicator node evidence is an improvement in the predicted V_i values, as shown in Figure 7-20. Again, we can graph the relative error of the median predictions as shown in Figure 7-21. In 3 out of 5 iterations, the MRE improves as a result of taking the onsite customer into account. It becomes worse in iterations 4 and 8 however. In iteration 4 it predicts an increase in PV, but it does not take into account the post-Christmas malaise. Iteration 8 was the final iteration. In the words of the paper authors who reported the Motorola project [215]:

“The velocity is only approximate for the last iteration because the team created and destroyed story cards throughout the iteration.”

The MMRE drops from 0.157 without the onsite customer node, to 0.139 with it. The variance of the relative errors also drops from 0.004 to 0.002 demonstrating greater consistency in the model’s predictions.

7.8.4 Calibrating the Onsite Customer Node

The distribution for the “Onsite Customer” node is based on data from Korkala, Abrahamsson and Kyllönen [112]. This paper is discussed in section 5.4.11.

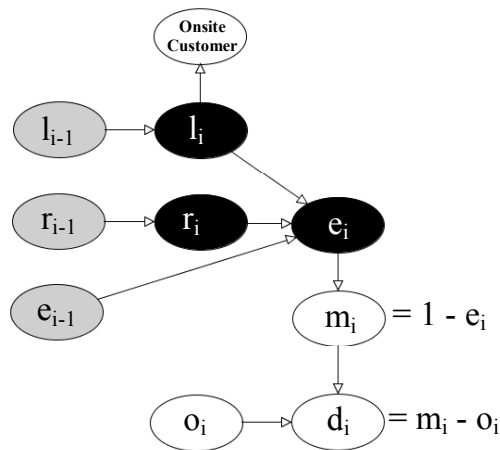


Figure 7-22 BN used to calibrate the Onsite Customer node

Our model does not explicitly include details of defect fixing effort (including requirements defects); they are simply included as effort which does not contribute to V . We therefore make the following definitions and assumptions concerning the relationship between defect fixing effort and non-velocity effort.

1. Define “Miscellaneous Effort”, m_i , to be the fraction of effort that does not contribute to completed user stories: $E_i = V_i + m_i$.
2. Miscellaneous effort is composed of a variable component due to defect fixing effort, d_i , and a set of process overheads, o_i : $m_i = d_i + o_i$. This does not provide a full description of miscellaneous effort, but it is adequate for this model. Note that the process overheads o_i are not necessarily fixed across iterations. Indeed, we might expect these overheads to decrease as the team

become more familiar with the software development process.

3. When the onsite customer input is at its maximum, the defect fixing effort is at its minimum.

With these assumptions in place, we can use the BN shown in Figure 7-22 to calibrate the Onsite Customer node. The algorithm proceeds as follows.

1. An initial guess is made at the Onsite Customer distribution.
2. The values of \underline{d}_i are chosen so that, when the Onsite customer node is set to “Very High”, d_i produces a constant mean value of about 6% across all iterations. The values of \underline{d}_i are entered as observations in the model.
3. Modify the Onsite Customer distribution, with the value set to “Very Low” until the time spent fixing defects in iteration 3 is about 40%.
4. Repeat steps 2 and 3 until both conditions are satisfied simultaneously.

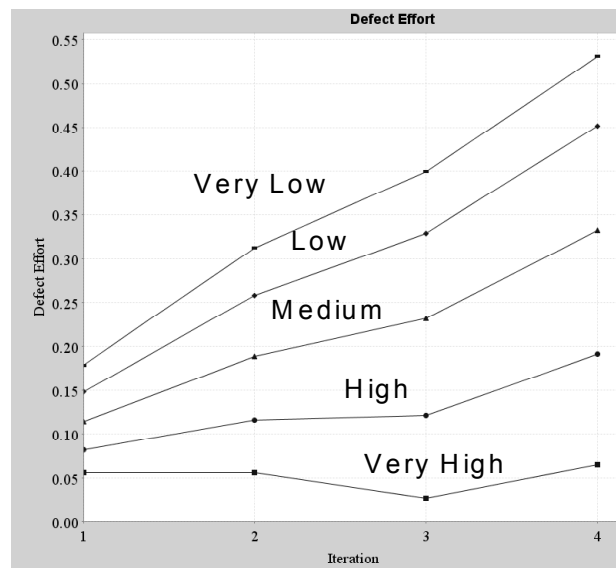


Figure 7-23 Defect effort % for each Onsite Customer setting

The resulting defect effort percentages for each value of “Onsite Customer” across four iterations are shown in Figure 7-23. These are similar to the empirical curves of Figure 3 in [112] and reproduced in Figure 5-1 (p.97). Note that our calibrated node results in less than 55% of time being spent on rework in the case of iteration 4 with a Very Low level of onsite customer communication. Figure 5-1 shows a level of 100% for the corresponding real case. However, as is explained in [112]:

“The last release of Case 4 concentrated solely on defect-fixing,

because all the other scheduled functionalities were cancelled.”

I am therefore assuming, that had further releases been available, some development of new functionality would still have taken place in release 4 of Case 4.

7.8.5 Timescale Prediction

Figure 7-24 shows a slightly modified version of the velocity fragment of the model. This includes an additional link node, s_i , which acts as the cumulative sum of V to date.

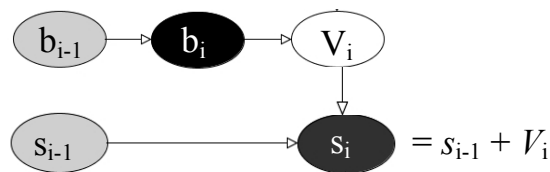


Figure 7-24 Project Velocity summed to date

Plots of s_i for the initial prediction, the learned prediction and the actual scenarios are shown in Figure 7-25. If the total estimate to complete the entire project is, say, 200 IEDs, then we can immediately read off from the graph how long it will take to complete the project.

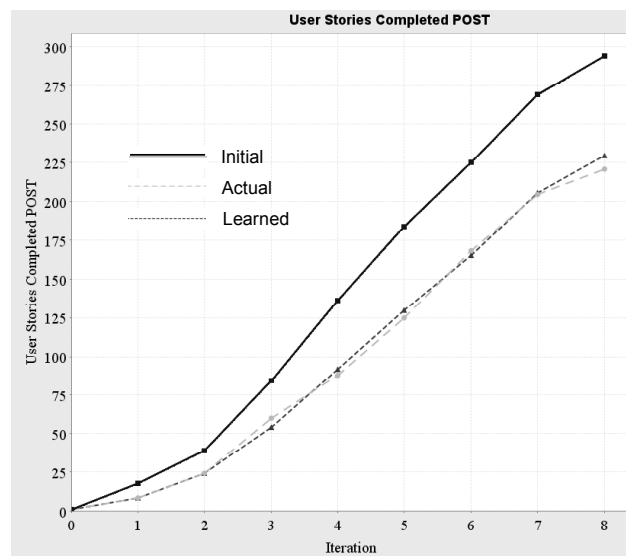


Figure 7-25 Sum V_i to date

The initial predictions of the model are too optimistic. However, once the model has learned from the V_1 and V_2 observations, and account has been taken of the onsite

customer, the predictions are virtually indistinguishable from the actual outcome.

Iteration	s_i	s_i^0	MRE_i^0	s_i^2	MRE_i^2
1	9	17.6	0.96		
2	22.5	39.0	0.73		
3	57.9	84.6	0.46	54.1	0.07
4	87.9	136	0.55	91.4	0.04
5	127.8	184	0.44	129.0	0.01
6	168.4	225	0.34	164.4	0.02
7	204.4	269	0.32	204.4	0.00001
8	224.4	294	0.31	228.4	0.02
MMRE			0.51		0.026

Table 7-4 The true functionality delivered after iteration i is s_i . The initial prediction for s_i is s_i^0 . The MRE for s_i^0 is MRE_i^0 . The prediction for s_i after two iterations is s_i^2 . The MRE for s_i^2 is MRE_i^2 . The Mean MREs are shown at the bottom of the relevant columns.

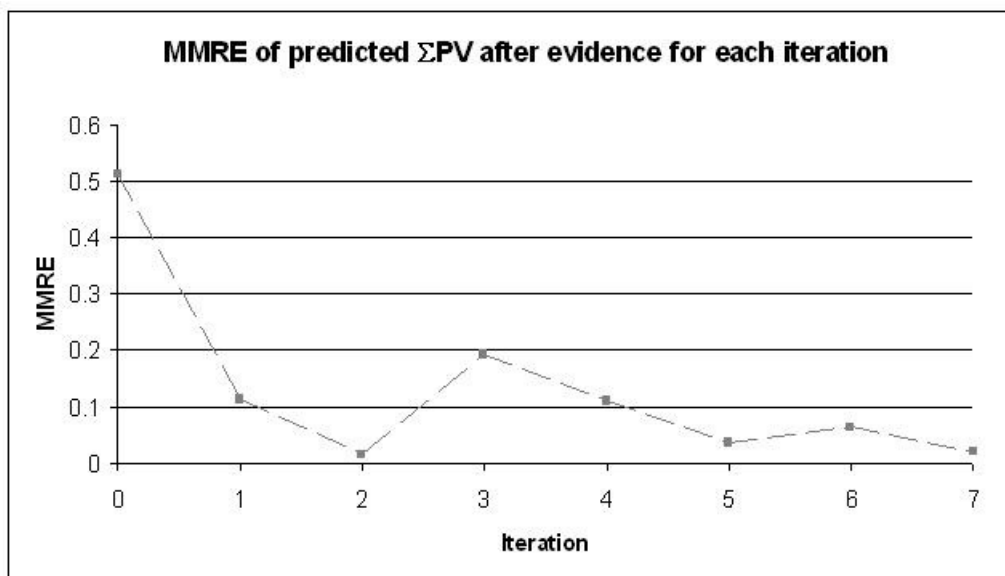


Figure 7-26 Sum V_i to date, Relative Error

The model's initial median predictions have MRE values between 0.3 and 0.96. This range reduces to 1E-5 to 0.07 after learning. The improvement is summarized in Table 7-4. As further evidence is entered into the model however, the accuracy of its predictions do not necessarily improve. For example when V_3 is entered, the model has no way of knowing that the next PV will be affected by the Christmas break. It therefore predicts a much higher value than is actually achieved. This causes the model's accuracy to degrade when V_3 is entered. We can examine this effect quantitatively by taking the Mean of the MRE values (MMRE) for the model's future median s_i predictions after each V_i is entered. After V_3 is entered, this jumps suddenly

before gradually improving once again. This is illustrated in Figure 7-26.

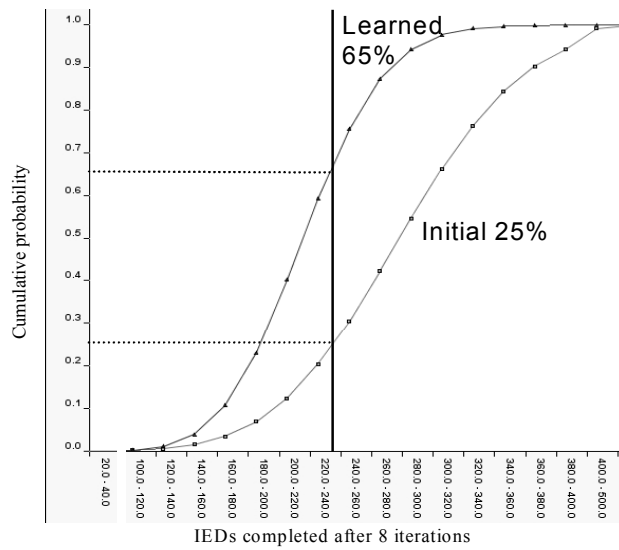


Figure 7-27 Iteration 8 cumulative distributions

The Motorola project completed 224 IEDs of functionality before the project ended. The model can quantify the uncertainty involved in completing 224 IEDs within 8 iterations. Figure 7-27 shows the cumulative distribution functions for the s_i node in iteration 8. The vertical line allows us to read off the probability of completing up to 224 IEDs by the end of the 8th iteration. For the “Initial” scenario, there is only a 25% chance of completing up to 224 IEDs (i.e. there is a 75% chance of delivering more than 224 IEDs). Once the model has learned from V_1 and V_2 , the probability is revised up to a 65% probability. This means that the model was initially too optimistic in its predictions (a 65% chance of delivering up to 224 IEDs means a 35% chance of delivering *more than* 224 IEDs).

7.8.6 Accuracy of the FOFFBK Algorithm

The PV model described in this chapter was built using AgenaRisk, with propagation between timeslices achieved using a Forward Only Fully Factored Boyen Koller algorithm (FOFFBK). As discussed in section 3.2.2, this is an approximate algorithm. If we are to have any confidence in the PV model then we must have some indication of the size of the error introduced by this approximation.

The PV model is sufficiently small that it is possible to run all eight iterations of the Motorola project as a single, unrolled, BN. We can therefore compare the results generated from running the “unrolled BN” model against the FOFFBK model. The

unrolled BN should produce more accurate results since it performs exact inference (subject to the limitations of the model, such as its choice of discretisation).

Care must be taken when comparing the unrolled BN model with the FOFFBK model. When we look at iteration i in the FOFFBK model, it includes information previous iterations, $i - n$, but not from future iterations, $i + n$. This is *not* the case in the unrolled BN. In the unrolled BN, every iteration includes information from all other iterations, including future ones. To ensure that we are comparing like with like we have to run the unrolled model eight times: once with one iteration, then with two iterations, then with three, and so on. In each case, we take the values from the final iteration included in that run of the unrolled BN model.

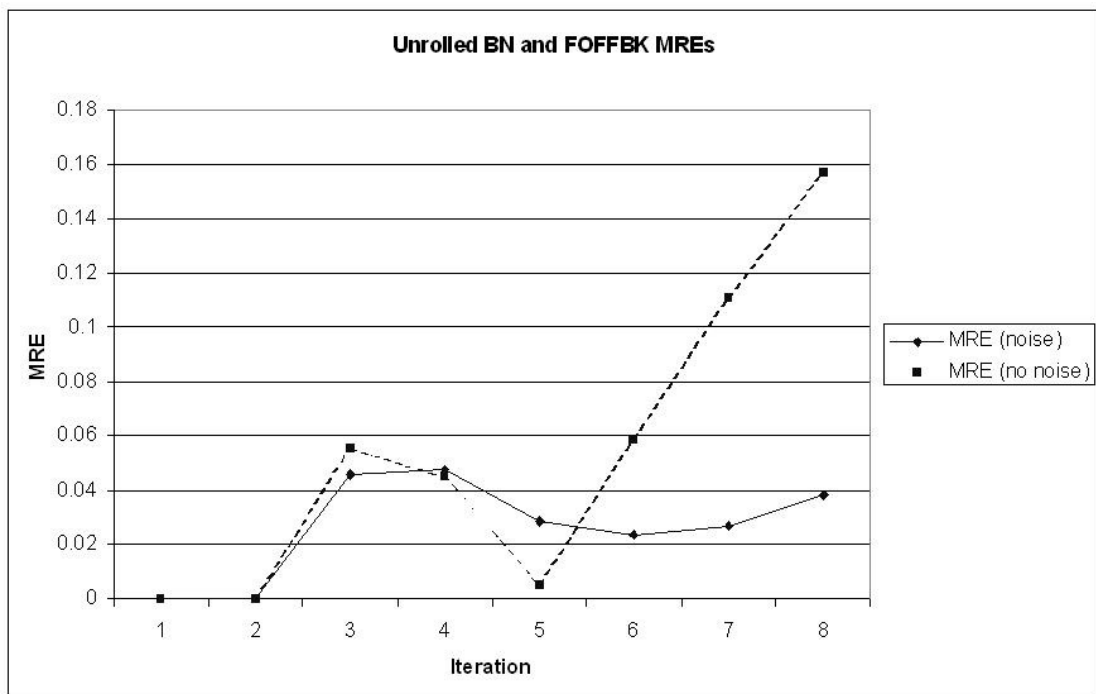


Figure 7-28 Magnitude of relative errors between the exact and approximate inference algorithms' PV predictions (both with and without noise).

Initially, the results of comparing the two models were poor. Early versions of the FOFFBK model did not include the TNormal expression shown in Figure 7-3. Instead, the e_i node had its CPD set to the arithmetic expression: $e_{i-1} + r_i \times (l_i - e_{i-1})$. The only mixing of states, required by all Boyen-Koller algorithms, is due to the priors and the choice of discretisation. The result is shown as the dashed line in Figure 7-28. This shows the magnitude of relative error between the median predicted PV for the unrolled BN model (exact inference) and the FOFFBK (approximate inference). In

both cases, the models included observations of PV for the first two iterations, and included the *Onsite customer* values. As can be seen, after iteration 5, the two models start to diverge, with the approximation errors apparently growing over time.

The PV model described in this chapter included extra noise in the form of the TNormal expression surrounding e_i . The variance of 5% was found empirically by testing the model with different TNormal variances ranging from 1% to 25%. The magnitude of relative errors between the “noisy” FOFFBK model and the unrolled model is shown as the solid line in Figure 7-28.

The mean MRE for the approximation algorithm is approximately 3.5%. However, this does not necessarily translate into a 3.5% error in timescale predictions, as Figure 7-26 shows. The PV approximation errors can be either positive or negative, with a tendency to cancel one another out when summed to give functionality delivered.

The “noisy” FOFFBK algorithm therefore behaves better than the FOFFBK without “noise” when compared to the exact inference algorithm.

7.9 Conclusions and Discussion

We have developed a model of XP project velocity and shown that it reproduces known empirical behaviour from iterative projects.

The model has been applied to a real industrial project. Incorporating data from the early part of the project enabled the model to update its parameters and improve its predictions. When this was combined with knowledge about the presence of an onsite customer, the model was able to make extremely accurate predictions about the level of functionality delivered over time. Other XP practices can be incorporated in the model using similar techniques.

While the model presented here has successfully demonstrated the benefits of using a learning BN model in XP projects, it is recognized that there are a number of threats to its validity.

1. The model relies on having sufficient degrees of freedom to learn from its environment. This is principally accomplished by updating the parameter nodes l_i and r_i . It is possible those are insufficient to accommodate the full range of behaviors of real XP projects, or that some future XP practices cannot be wholly accommodated as indicators of one of these nodes.
2. Only a single industrial test case has been used. Greater confidence in the model will be achieved through exposure to a greater variety of data sets.

3. The example shown had the benefit of real effort data from a completed project. At the start of a project, only projections of available effort are available.
4. No sensitivity analysis has been performed on the model priors in Iteration 0. However, regardless of the initial values, the model will adapt to the current project's local conditions as soon as the first few iterations are completed. Clearly, any change in the means or standard deviations of the priors will affect the model's initial predictions. We would expect that more mature software development organizations would replace the supplied values with distributions based on their own previous metrics programmes.
5. Two XP practices have been included in the model: "Collective ownership", using hypothetical data, and "Onsite customer", using data from a single study. Empirical data on the effectiveness of other XP practices needs to be used in order to calibrate appropriate indicator nodes.

Despite these concerns, there are a number of clear benefits to this approach.

1. Although prior metrics information is valuable, an extensive data collection phase is not essential. The model starts off making generic predictions, but quickly alters them as local data becomes available. Developers tasked with metrics collection therefore see an immediate benefit from doing so: predictions about their own project will improve as a result. Contrast this with traditional metrics collection programs, which often founder because of the need for long-term commitment.
2. Empirical data, project data, prior assumptions and expert judgment are combined in a single intuitive, learning model.
3. The predictions provide probability distributions, not just single values. The model tells you what the chances of various outcomes are.
4. Provided suitable empirical evidence is available, it is relatively simple to add new XP practices or other environmental features, making the model extremely versatile.

The model presented here differs from many of the causal models described in section 4. Rather than trying to construct a complex graph of causal relationships, it opts instead for a very simple structure. This model recognizes that, for a large variety of reasons, software productivity varies throughout the iterations of an agile project. It therefore learns the cumulative effect of these variations rather than trying to model

their interactions explicitly.

Users of the model only need to provide three items of information:

1. available effort over the timescale of the project,
2. measured project velocity as it becomes available,
3. the extent to which XP practices are varying between iterations.

The first two should be available anyway in any XP project and the third can be supplied using subjective judgment. The burden to developers and managers in maintaining this model is therefore minimal. In return for this small overhead, projects get improved PV predictions such as in Figure 7-25 and a quantitative assessment of the risk, as in Figure 7-27.

8 Extending the Model

The previous chapter showed how a very simple DBN can be used to model PV in XP projects. In this chapter I show how this model can be extended in two distinct ways.

1. A measure of quality is introduced into the model. This extension allows the model to make predictions about defects. When evidence is entered for the observed number of defects this enables the model to learn the value of quality parameters.
2. The model gets applied to a second agile environment: Scrum.

The models presented in this chapter should be regarded as “proof of concept” only. Further research is required to externally validate the models. Both of these extensions to the model represent novel contributions.

8.1 Adding Quality to the Model

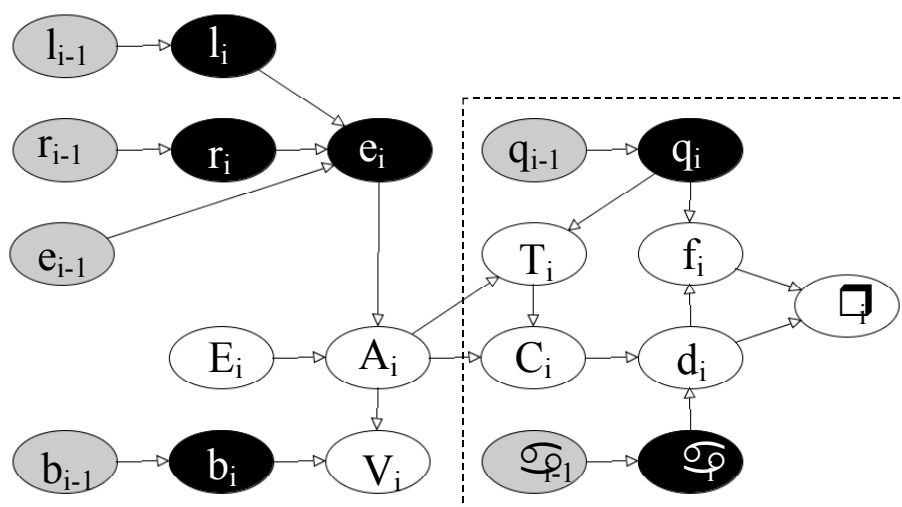


Figure 8-1 - PV model updated to include code quality

In this section we add the notion of quality to the XP PV model. There are two aims in doing this.

1. Knowing the quality of the delivered code allows the model to make predictions about the number of defects present. Entering the observed number of defects allows the model to update its assumptions about code quality and so improve its predictions.
2. Varying the quality enables the model to perform trade-off analysis within

individual iterations.

The model should also show the same useful characteristics that the PV-only model displayed, namely: simple data input, learning across iterations and compatibility with existing XP practises and philosophy. The updated model is shown in Figure 8-1 with the new nodes enclosed in the dashed box. A formal description of the new nodes is given in Table 8-1.

Symbol Meaning

q_i	Quality of code produced in iteration i . This is a ranked node with 5 values ranging from Very Low to Very High. This is not directly observable.
T_i	Test and design effort in man-days. $T_i = A_i \times q_i$.
C_i	Code effort in man-days. $C_i = A_i - T_i$.
α_i	The rate at which defects are inserted per man-day of effort. $\alpha_i = \text{Normal}(1, 0.1)$, $\alpha_i \in [0, \infty)$.
d_i	The number of defects inserted given the code effort and defect insertion rate. $d_i = C_i \times \alpha_i$, $d_i \in [0, \infty)$.
f_i	The number of defects found. $f_i = \text{Binomial}(d_i, q_i)$.
ρ_i	Residual defects. $\rho_i = d_i - f_i$.

Table 8-1 Quality model symbol definitions

8.1.1 Model Structure

For simplicity we introduce a single parameter, q_i , which defines the quality of the code being produced. (Note that this is a *product* quality parameter. *Process* quality is effectively defined by e_i .) This is a five state ranked node. We take a high value for q_i to imply the following.

1. A greater proportion of the actual productive effort will be taken up by design and testing.
2. There is a higher probability that testing will discover defects.

In the actual model shown I have used the simplest possible implementation that is consistent with these assumptions. The “Test” Effort, T_i , is taken to be directly proportional to q_i , with $T_i = A_i \times q_i$. Similarly, the probability of discovering a defect is taken to be exactly q_i . Both of these are possible because the AgenaRisk toolset implements the ranked node as an underlying real value in the range [0,1].

For the purposes of this model, “test” effort also includes effort spent on the detailed design of components. Ideally this would be a separate node. In this idealised model it is combined with test effort because they have similar relationships with q_i . As the quality goes up, we expect the effort spent on design to also go up. As the design effort increases, the chance of finding and removing defects in the design also increases.

It is tempting to think that, as quality goes up, the number of user stories delivered in an iteration goes down and that the PV therefore goes down too. If this were true then we would need to add a link from our new quality parameter, q_i , to the process effectiveness, e_i . However, this is not the case, as the following example illustrates. Suppose V_i is observed to have a value of 80. Four user stories: US6, US7, US8 and US9, are scheduled for iteration $i + 1$. All four have been estimated to take 20 IEDs to complete. During the planning meeting for iteration $i + 1$, the team decide that stories US6 and US7 are particularly crucial parts of the system. They should therefore be produced to a particularly high level of quality. The team revise their estimates for US6 and US7, up from 20 IEDs to 40 IEDs each, so only US6 and US7 are included in iteration $i + 1$. The total estimate for the next velocity remains the same, even though the number of user stories has halved.

In each iteration, the node A_i determines the amount of effort available for development. This can be split between the design/test effort T_i and the code effort C_i . It is this split between code and design/test that allows the model to perform trade-off analysis. More effort can be allocated to generating poorly thought out and poorly tested code, or less code can be generated but of a higher quality with a greater chance of removing defects.

Measuring design, code and test effort explicitly is precisely the type of project overhead that we do not wish to impose on agile projects. We therefore need some proxy for these measurements. Code and test effort could possibly be indicated by the LOC measures of the relevant packages, with the ratio of production code to test code matching the ratio of code to test effort. Measuring design effort is more problematic. Possibilities include the number of code management check-ins (with more check-ins implying more revisions of the design), or the number of fully specified method definitions prior to coding.

Whatever metrics are chosen, there is a clear need for model integration with IDEs, code management systems, bug tracking systems and other development tools. This

querying across heterogeneous data sources implies the kind of integrated approach to data extraction and normalisation specified in section 6.2.

The model assumes a fixed distribution of defect insertion for each man-day of effort, α_i . Multiplying this by the amount of code effort C_i , gives the number of defects inserted into the code d_i .

Interpreting the code quality q_i as being numerically equal to the probability of finding a defect allows the model to use the binomial distribution with d_i as the number of trials and q_i as the probability of success. This is exactly the same as the approach used in the Modist model (4.3.2) and the Philips model (4.4).

8.1.2 Validating Model Consistency

We can test the defects model by looking at a typical iteration. Table 8-2 shows the result of a single iteration, where E is set to 50, V is set to 25 and q is varied through each of its values from Very Low to Very High. All values are median values.

q	0.1	0.3	0.5	0.7	0.9
e	0.465	0.465	0.465	0.465	0.465
T	2.73	7.28	11.8	16.3	20.7
C	20.1	16.07	11.4	6.85	3.26
d	19.4	15.3	10.8	6.35	3.14
f	1.92	4.34	5.17	4.24	2.69
α	1	1	1	1	1
ρ	17.1	10.7	5.34	1.83	0.682

Table 8-2 Test results when $E = 50$, $V = 25$ and q is varied from Very Low through to Very High (0.1 to 0.9 in steps of 0.2).

Test and design effort T and code effort C both vary linearly in the expected direction. The process effectiveness e and defect insertion rate α both remain constant. The defects inserted d , defects found f and residual defects are more interesting.

As expected, as the product quality increases, the number of defects inserted decreases. This is primarily due to less code being generated. However, even though the number of defects generated decreases, the probability of detecting an error initially goes up. This causes the number of defects found to also go up, even though there are fewer defects to be found. It is only when the number of defects generated becomes sufficiently small that the number of defects found begins to decline again.

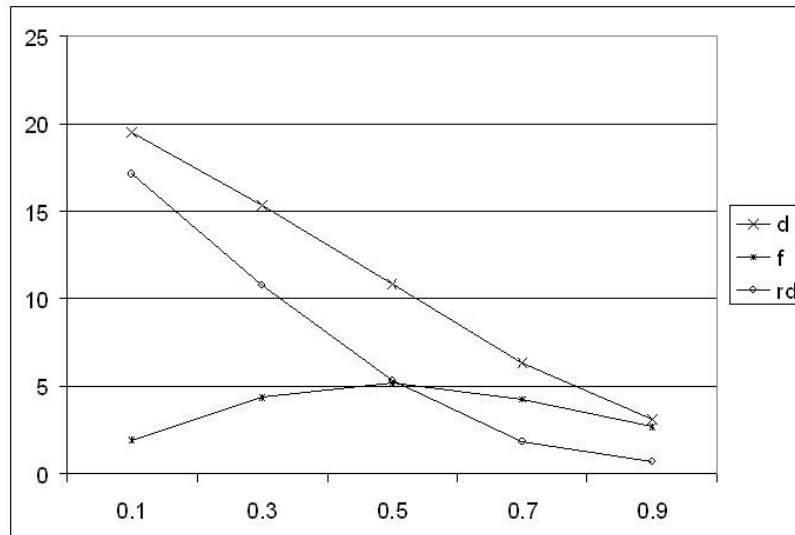


Figure 8-2 Defects inserted d , defects found f and residual defects rd , as q is varied from Very Low (0.1) to Very High (0.9), with $E = 50$ and $V = 25$.

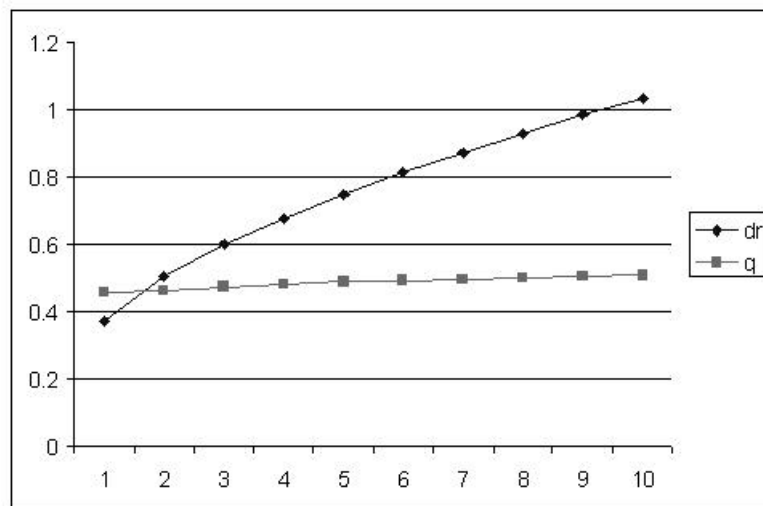


Figure 8-3 Defect insertion rate dr (α) and product quality q as number of defects found varies from 1 to 10. $C = 20$, $T = 20$.

We can perform a similar test to see the effect of changing f_i (defects found). Figure 8-3 shows the result when $C = 20$, $T = 20$ and the number of defects found is varied from 1 through 10. The product quality increases slightly. The bulk of the increase in defects found is explained by an increase in the number of defects being inserted. This is as we would expect, all other things being equal.

As a final test we can keep the code effort constant, the number of defects found constant and increase the amount of testing. As the amount of test code, relative to the amount of production code is increasing we would expect the product quality to

increase. The product quality is the same as the probability of finding a defect in this simple model. If the probability of finding a defect is increasing, but we are still finding the same number of defects, the defect insertion rate must be going down. This is exactly what the model does, as shown in Figure 8-4.

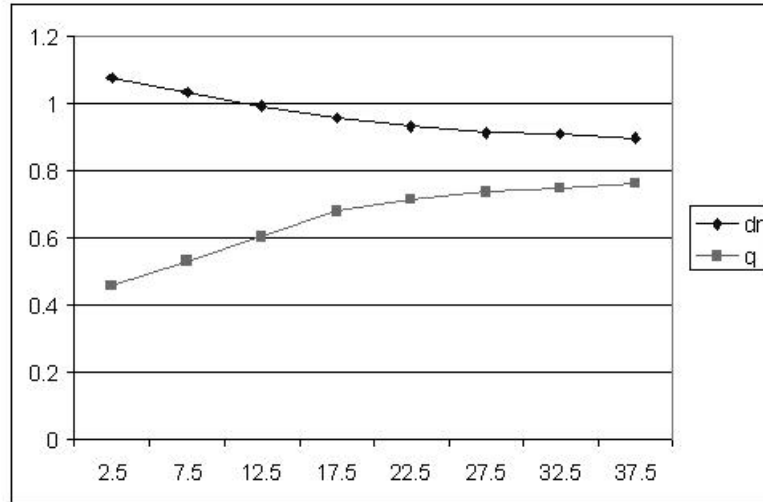


Figure 8-4 Defect insertion rate dr and product quality q with $C = 10, f = 5$ and rising value of T .

8.1.3 Model Learning and Prediction

The previous section shows that the defects model behaves largely as our intuition expects when we vary the evidence within a single iteration. We must now see how the model behaves when multiple iterations are linked together to form a dynamic model.

	E_i	V_i	T_i	f_i
Scenario name				
Baseline	$\underline{E}_i = 50$			
High test low faults	$\underline{E}_i = 50$	$\underline{V}_1 = 15$	$\underline{T}_1 = 10$	$\underline{f}_1 = 3$
		$\underline{V}_2 = 20$	$\underline{T}_2 = 15$	$\underline{f}_2 = 3$
		$\underline{V}_3 = 24$	$\underline{T}_3 = 18$	$\underline{f}_3 = 4$
		$\underline{V}_4 = 27$	$\underline{T}_4 = 20$	$\underline{f}_4 = 4$
High test high faults	$\underline{E}_i = 50$	$\underline{V}_1 = 15$	$\underline{T}_1 = 10$	$\underline{f}_1 = 5$
		$\underline{V}_2 = 20$	$\underline{T}_2 = 15$	$\underline{f}_2 = 6$
		$\underline{V}_3 = 24$	$\underline{T}_3 = 18$	$\underline{f}_3 = 6$
		$\underline{V}_4 = 27$	$\underline{T}_4 = 20$	$\underline{f}_4 = 7$
Low test low faults	$\underline{E}_i = 50$	$\underline{V}_1 = 15$	$\underline{T}_1 = 5$	$\underline{f}_1 = 3$
		$\underline{V}_2 = 20$	$\underline{T}_2 = 6$	$\underline{f}_2 = 3$
		$\underline{V}_3 = 24$	$\underline{T}_3 = 7$	$\underline{f}_3 = 4$
		$\underline{V}_4 = 27$	$\underline{T}_4 = 8$	$\underline{f}_4 = 4$
Low test high faults	$\underline{E}_i = 50$	$\underline{V}_1 = 15$	$\underline{T}_1 = 5$	$\underline{f}_1 = 5$
		$\underline{V}_2 = 20$	$\underline{T}_2 = 6$	$\underline{f}_2 = 6$
		$\underline{V}_3 = 24$	$\underline{T}_3 = 7$	$\underline{f}_3 = 6$
		$\underline{V}_4 = 27$	$\underline{T}_4 = 8$	$\underline{f}_4 = 7$

Table 8-3 Dynamic quality model test scenarios

To construct a dynamic quality model we make the same linkages as we did for the PV model. In addition, we link q_i from iteration i to q_{i-1} in iteration $i + 1$ and α_i in iteration i to α_{i-1} in iteration $i + 1$.

We can examine the behaviour of the dynamic quality model by constructing different scenarios. These are shown in Table 8-3. All of the scenarios have 8 iterations. The *Baseline* scenario sets all of the \underline{E}_i values to 50. No other evidence is entered. The *Baseline* scenario provides a comparison against which the other scenarios can be measured. All of the alternative scenarios also have each \underline{E}_i value set to 50. The alternative scenarios also have the same values for \underline{V}_1 to \underline{V}_4 , however they have different values for \underline{T}_1 to \underline{T}_4 and for \underline{f}_1 to \underline{f}_4 . There are four alternative scenarios and their names are self-explanatory: *High test low faults*, *High test high faults*, *Low test low faults* and *Low test high faults*. (In all scenarios, the term “faults” means “defects found through testing”).

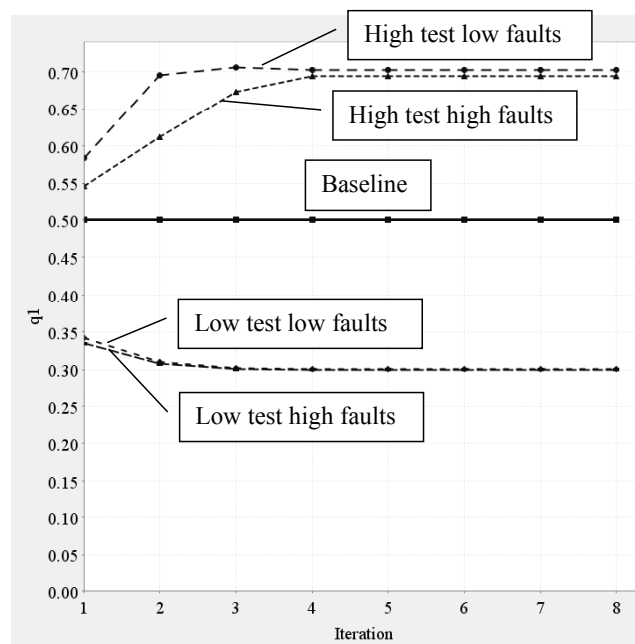


Figure 8-5 Learned values for q under different scenarios.

Figure 8-5 shows how product quality q is learned as the amount of testing varies. Although the number of defects found has some impact on q it is the amount of testing which has the biggest effect. As expected, a high level of testing leads to a high product quality.

Figure 8-6 shows the corresponding graph for the defect insertion rate α . The dominant factor here is the number of defects discovered. However, the amount of testing is also significant. To understand this we have to take two similar cases where

the same number of defects are discovered. In one case the defects are discovered after very little testing, suggesting that there are a large number of defects present and so a high defect insertion rate. In another case the same number of defects are found, but after much more rigorous testing, suggesting that few defects remain and so implying that the defect insertion rate must be lower.

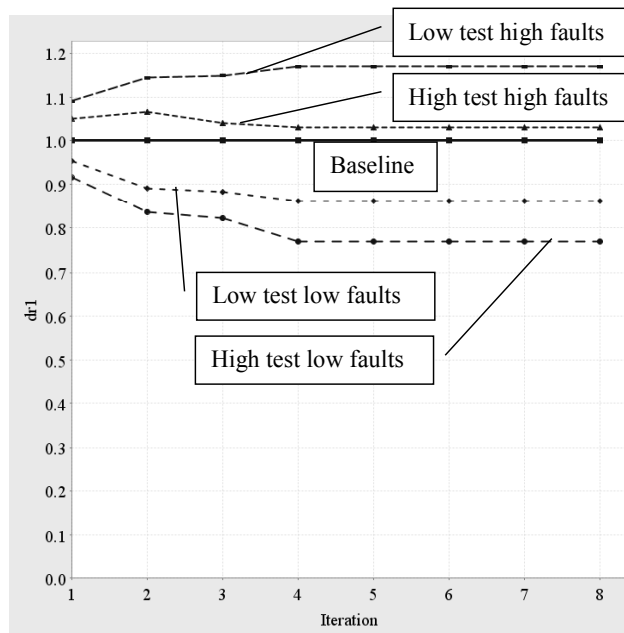


Figure 8-6 Learned values of alpha in varying scenarios.

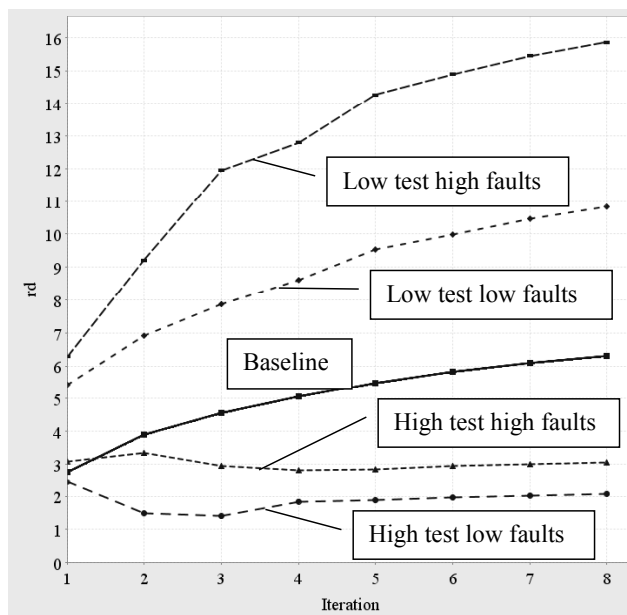


Figure 8-7 Residual defects in each iteration for different scenarios.

Figure 8-7 shows the number of residual defects predicted. The residual defect prediction curves are ordered as we would expect: low test scenarios predict larger number of defects and low defect discovery scenarios predict lower residual defects for the same amount of testing.

8.1.4 Future Model Work

The simple model demonstrated in the previous sections shows that it is possible to construct a learning model of product as well as process quality that is consistent with our common-sense notions of how a software project “ought” to behave. However there are many outstanding problems which require further research.

- Proper priors for the quality parameter q and the defect insertion rate α need to be established.
- The single quality parameter q needs to be mapped to the proportion of effort devoted to testing and separately mapped to the probability of finding a defect.
- Test and design effort need to be included separately.
- We need to find metrics to measure design, code and test effort.
- Establish data sources and automated data extraction for above.
- The model needs to be validated against a real project.
- The model currently takes no account of the effort required to fix defects. At the moment this is simply subsumed within the process effectiveness e . It is an open question whether there is any benefit to be gained from modelling rework effort.
- The PV model assumes a trend in process quality. Is there a similar trend in product quality q ?

8.2 *Creating an Iterative Model for Scrum*

We briefly mentioned Scrum in section 5.5.2. Scrum is an agile development method which bears a close resemblance to XP. The name “Scrum” derives from a rugby scrum where a small number of players act together to achieve a well defined goal. The term was first applied as a project methodology by Takeuchi and Nonaka [201]. It was subsequently applied independently in software projects by Schwaber [178] [179] and Sutherland [199].

Scrum places less emphasis on individual programming practises than XP, although the short scale iterative nature of the method inevitably leads to a large overlap. As with XP, a product owner is responsible for creating a prioritised list of requirements. Initial effort estimates are allocated to these requirements by the development team. In scrum this list is called the *Product Backlog*. Each iteration (a *Sprint* in Scrum terminology) is preceded by a sprint planning meeting where the contents of the next sprint are decided. This involves a discussion between the product owner and the development team, who must decide how much of the product backlog they can implement in the next sprint. The length of each sprint is usually 3-6 weeks and, as with XP, is expected to deliver a functioning piece of software.

Once the contents of a sprint have been agreed they cannot be altered. The agreed contents are then transferred from the product backlog to the *Sprint Backlog* where they are broken down into smaller tasks which can be measured in hours. The total number of hours on uncompleted tasks are then plotted daily on a *Burndown chart* Figure 5-2.

Every day there is a team meeting (the *Daily Scrum*) where each team member provides three pieces of information.

- What they did yesterday.
- What they will do today.
- What problems they face.

Anecdotal evidence (see for example [171]) suggests that the daily scrum can act as a clearing house for code refactoring where small changes in one task can unblock problems being encountered elsewhere, or where code to solve common problems can be shared. This in turn can lead to the kind of “hyper-productive” state described by Sutherland [199]. Other teams report similar feelings of increased productivity [139], although to this author’s knowledge there has been no systematic attempt to demonstrate this.

There are clear parallels between Scrum and XP, even the low productivity in initial iterations seems to be shared between the two [178] (which is hardly surprising given their similarity). Detailed project management issues are not addressed by Scrum. However, in practise most Scrum projects use the definition of project velocity which has been used in this thesis. The units vary, Schwaber recommends function points [180], but it is easy to find examples where story points [198] or IEDs are used

instead. As we have already shown in 5.5.1, provided the units used to measure PV form a ratio scale, the exact nature of those units are immaterial. We can therefore take the PV model unaltered across to a Scrum project. However we can also do more than that.

Scrum typically uses burn down charts (Figure 5-2) to report progress. Using the information from the Motorola project (7.8.1) we can construct real and predicted burn down charts for that project. Burn down charts are a more natural tool for Scrum projects than PV. It is extremely simple to adapt the PV model described in 7.8.5. Instead of taking the sum of the project velocities to date (which shows the amount of functionality delivered to date), we define s_i to be the amount of functionality still to be completed at the end of sprint i and initialise s_0 with the total amount of functionality to be delivered by the project. When sprint i is completed, we deduct V_i from s_{i-1} to give the functionality remaining at the end of sprint i . The revised node is illustrated in Figure 8-8.

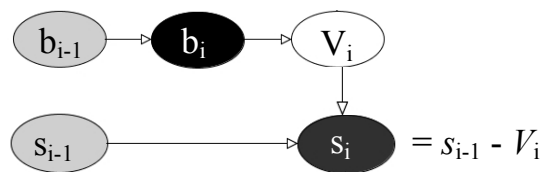


Figure 8-8 Modelling burn down

In the scrum model we do not have to enter values for V_i . Evidence is entered for s_i instead. As burn down values become available after each sprint, the model adapts its predictions for V_i and e_i leading to improved predictions for future iterations. Figure 8-9 shows three versions of the burn down chart for the Motorola project. In all three cases the project begins with 224 IEDs to be delivered (this is the amount that the project actually delivered before being cancelled). The “Initial” graph shows the predicted burn down chart when only effort values E_i have been entered. The “Actual” graph shows the final burn down chart as it would actually appear in the Motorola project. Values for s_i were calculated by simply subtracting V_{i-1} from s_{i-1} .

The “Learned” graph shows the predicted burn down graph after s_1 and s_2 have been entered and with appropriate values for the *onsite customer* node. As with the PV model, the burn down model has learned from the evidence entered and improved its predictions as a result.

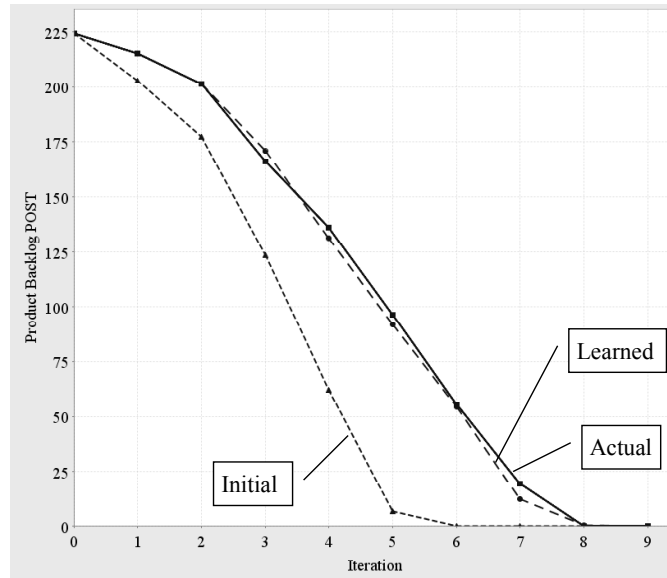


Figure 8-9 Burn down graphs for the Motorola project.

8.3 Extending BN Models to Other Agile Methodologies

The previous sections have shown how the PV model of chapter 7 can be extended to encompass quality in an XP project and how a simple modification can be made to make it more applicable to Scrum. Although XP and Scrum are two of the most popular agile methods in use, they are not the only ones. In 2006 Dr. Dobb's Journal published the results of a survey [12] into the adoption of agile methods. Over 54% of the survey respondents who used agile methods used either XP or Scrum, with XP alone accounting for over 36% of agile method usage. The models shown so far can therefore claim to be applicable to the majority of the agile community. In this section we look at some of the other agile methodologies available.

A good overview of 9 of the most popular agile methods can be found in [4]. The authors describe the relationships between each method and explore them through five distinct perspectives: software lifecycle coverage, project management, abstract principles versus concrete guidance, universal versus situation specific and empirical evidence. We will restrict ourselves to the three further methodologies which, together with XP and scrum, account for nearly 90% of the Dr. Dobbs survey responses.

8.3.1 FDD

The second largest agile methodology in the Dr. Dobb's survey, accounting for nearly a fifth of respondents, was Feature Driven Development (FDD) [45]. FDD is different from XP and Scrum in that it is quite prescriptive in defining the tools, roles and processes that should be used on a software project. It begins with large upfront processes that create an overall design in UML (modified to include a colouring convention for class diagrams), builds a "feature" list (with a feature being essentially equivalent to a user story in XP) and create a project plan. It then launches into an iterative cycle of detailed design and feature development. The big difference is that teams are fluid, being created as needed for a given feature, with each team working to its own start and end dates. There are also many minor differences from XP, such as prescribing class ownership rather than common code ownership. The processes involved in FDD have spawned a small industry of support tools.

The issue as to whether FDD is truly "agile" or not arises from time to time on web forums. Ron Jeffries, one of the founders of XP, seems to think not [5]. If we take the definition of "agile" to be the Agile Manifesto [9] then we can test FDD against the various statements from that manifesto. FDD clearly provides some of the iterative and people oriented processes that the manifesto demands. It also concentrates on software production with the minimum overhead required for upfront modelling (FDD is geared towards larger projects where some architectural and overall design work is essential). It is less clear that FDD can handle fluid requirements and constant customer feedback.

The overlapping development cycles and fluid development teams of FDD clearly present a problem for the type of model developed so far in this thesis. The controversy over whether FDD constitutes an "agile" method in the first place suggests that FDD is significantly different from other agile methodologies. If this is the case, then it is not surprising that models designed to fit other agile methodologies are less applicable to FDD.

8.3.2 AUP

The Agile Unified Process (AUP) [13] was used by just over 8% of the Dr. Dobbs' respondents. AUP tries to combine the Rational Unified Process (RUP - see section 4.9) with the practises common to agile development. RUP is really a framework from which a more project specific process can be constructed. AUP reflects this by

defining a wide range of activities and artefacts which a project may choose to implement. As with RUP, AUP makes the distinction between project phases: inception, elaboration, construction and transition, and the disciplines undertaken in those project phases: model, implementation, test, deployment, configuration management, project management, environment.

At the heart of the construction phase, the implementation discipline in AUP is highly iterative. The project management discipline specifically lists collection of project velocity as one of its responsibilities. Clearly, the PV model can be directly applied in AUP.

8.3.3 Agile MSF

Microsoft Solutions Framework (MSF) for Agile software development, is Microsoft's contribution to agile methodologies. It defines a set of roles, principles, work items and reports for use in agile projects. Each role is associated with a well defined set of work streams. The *Project Manager* role, for example, is associated with the *Plan an Iteration* work stream. Each work stream is then subdivided into a set of work items each of which has a corresponding database record which includes the status of the work item and the effort involved. This database can then be used to generate a set of standard reports showing project activity in the form of burn down charts or project velocity.

Project managers are expected to use PV in their iteration planning. Work item sizes are measured in different units depending on the type of work unit. *Task* work items are measured in hours, although it is not clear from the Microsoft documentation if this is elapsed time or effort. Regardless of whether elapsed time or effort is used, they both form a ratio scale and so can be used in the PV model presented in chapter 7.

Scenario work items, which roughly correspond to use cases or user stories, are measured on a ranked scale with values 1, 2 or 3. The value 1 corresponds to items requiring up to 12 calendar days; 2 corresponds to 12 to 24 calendar days, and 3 corresponds to more than 24 calendar days (with a recommendation that items with a value of 3 be split if possible). The PV model could be used with *Scenarios*, but clearly it will be more accurate once a *Scenario* has been broken down into tasks.

9 Overall Summary and Conclusions

This section provides a summary list of the main points of this thesis. In the conclusion, the summary points that support the hypothesis are highlighted.

9.1.1 Summary

1. Chapter 2 showed why existing, mainly regression or multivariate based models are unable to capture the full complexity of software development projects.
2. Chapter 3 showed how learning can be extended from a BN to a DBN.
3. Chapter 4 showed how existing Bayesian Net (BN) models can address software processes by building causal models of the software development environment.
4. Chapter 5 provided a review of the literature on XP and showed that any ratio scale could be used to measure Project Velocity (PV).
5. Chapter 6 showed why we would have difficulty applying the same techniques in iterative development environments.
6. Chapter 7 demonstrated a learning Dynamic Bayesian Net (DBN) model of Extreme Programming's (XP) key PV metric.
7. The model exhibited key characteristic properties of real world XP projects, including slow initial productivity and the ability to vary productivity by changing XP practices.
8. By entering PV data, the model learned about its local environment, altering its future PV predictions.
9. Varying the PV data supplied to the model showed that the model was capable of operating in a consistent manner over a wide range of conditions.
10. Using data from a real XP project at Motorola, the model's predictions were verified to a good degree of accuracy.
11. Providing only one or two measurements of real PV from the Motorola project was sufficient to calibrate the model. Since these PV measurements would be made anyway, the model essentially places zero overheads on the management of the project.
12. The PV model in this thesis was developed using PV measured in Ideal Engineering Days. It was shown in section 5.5.1 that using PV to plan

iterations can be used with any measure of PV, provided that measure forms a ratio scale. This model therefore works equally well with Story Points and Function Points.

13. The model uses a forward only, fully factored, Boyen-Koller DBN inference algorithm. As was shown in section 3.2.4, forward-only learning is faster than full forwards-backwards learning and has descriptive and diagnostic advantages in that we can easily show the history of learned parameters. Forward only has the disadvantage that it prevents DBN smoothing, limiting the usefulness of the model as a decision support tool.
14. Chapter 8 has shown possible ways in which the XP PV model might be extended. We have seen how quality measures might be introduced and suggested topics for further research such as finding adequate proxy measurements for design, code and test effort.
15. The need to access measures of effort and quality, together with the requirement to minimise project overhead, implies the type of tool integration and heterogeneous query capability described in section 6.2.
16. The PV model was easily modified to use Scrum's burndown metric instead of XP's PV metric. The Scrum model showed similar learning capability using minimal data and demonstrated the same high level of accuracy once calibrated.
17. Applicability of the PV model to other popular agile methodologies was discussed. This concluded that the PV model could be applied in AUP and MSF. However, in FDD development teams are mutable in nature and irregularly stagger iterations across teams. The model is not directly applicable to such an environment. Although FDD is widely used, there is controversy in some quarters as to whether it constitutes an agile method at all.
18. A lightweight scripting language was defined to create repetitive Bayesian nets. The implementation of this language is specific to the AgenaRisk development tool. However, the principle of using an essentially declarative programming language to define BNs and DBNs is generally applicable.

I claim novelty for points 4 through 10, and 12 through 16.

9.1.2 Conclusions

The hypothesis given in section 1.1 states the following.

H1.It is possible to construct learning models of iterative agile development environments.

H2.These models require minimal “training” and minimal data collection programmes.

H1 is supported by summary points 6, 7, 8, 9, 10, 12, 14, 16 and 18. H1 is partially supported by 17, although we have identified at least one agile development method, FDD, to which the models developed here cannot easily be applied.

H2 is supported by 11, 15 and 16.

Appendix A - An Introduction to Bayesian Nets

A Bayesian Network (BN) is a Directed Acyclic Graph (DAG), where the nodes represent random variables and the arrows represent causal influences. Nodes without parents are defined by a random variable's probability distribution. Nodes with parents are defined by Conditional Probability Distributions (CPDs). Probability distributions and CPDs are collectively known as Node Probability Tables (NPTs).

The theory of BNs has been developed mainly since the early 1980s, by Pearl [160], Neapolitan [149], Jensen [93], Lauritzen and Spiegelhalter [120].

Example A-1

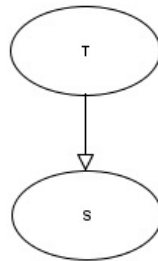


Figure A-1 A simple "Temperature and snow" BN

The T node represents the probability, on any given day, that the outside temperature is 0°C or below. The S node gives the probability that it will snow or hail on a particular day.

The probabilities assigned to these random variables are shown below.

T	$P(T)$	T	true	false
true	0.01	S	$P(S T)$	
false	0.99		true	0.05
		false	0.95	0.99

As can be seen from the tables, a low probability (1%) is assigned to a sub-zero temperature. If the temperature is above zero then we assign a small probability of snow or hail, but increase this by a factor of 5 when the temperature is sub-zero.

Note that, in Example A-1, given a particular value of T , the probabilities for S must add up to one:

$$\sum_S P(S | T) = 1. \quad \text{Equation A-1}$$

From elementary probability theory we get:

$$P(S | T) = \frac{P(S, T)}{P(T)}. \quad \text{Equation A-2}$$

This can be justified on the basis of frequentist arguments (see for example [23]) or taken as an axiom. From this we can see that we can easily calculate the joint probability $P(S, T)$. Given a Joint Probability Distribution (JPD) of two variables S and T , we can find the probability $P(S = s_1)$, where s_1 is one possible state of S , by summing over all of the n mutually disjoint states of T .

$$P(S = s_1) = \sum_{i=1}^n P(S = s_1, T = t_i). \quad \text{Equation A-3}$$

We refer to this process as *marginalisation* and call the probability distribution $P(S)$ the marginal of S with respect to $P(S, T)$.

Using simple symbol substitution, we can derive a second expression for $P(S, T)$:

$$P(S, T) = P(S | T)P(T) = P(T | S)P(S). \quad \text{Equation A-4}$$

Rearranging gives us Bayes Theorem:

$$P(T | S) = \frac{P(S | T)P(T)}{P(S)}. \quad \text{Equation A-5}$$

Bayes Theorem allows us to calculate a CPD that we may not otherwise have access to. In Equation A-5, $P(T)$ is called *prior* probability of T . In Example A-1 it is our belief that it will be sub-zero outside given no other

information. The left hand side, $P(T|S)$ is the *posterior* : our belief that it is sub-zero given our knowledge of whether it is snowing or not.

Hard evidence, otherwise known as an *observation*, is entered into a node, A , in a Bayesian Network when we know its value, a_i , for certain. This is equivalent to setting its posterior distribution to:

$$\begin{aligned} P(A = a_i) &= 1 \\ P(A = a_j) &= 0, i \neq j \end{aligned} \qquad \text{Equation A-6}$$

When evidence is entered into a node, the distributions of all other nodes must be recalculated to take account of this change in its probability distribution.

Example A-2 (Taken from the Stanford Encyclopedia of Philosophy [195])

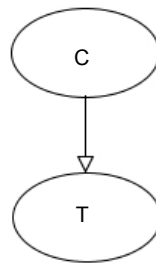


Figure A-2 Test for Drug use

In this example C represents the probability that an individual in a population is a cocaine user. We will assume that it is 3%. The T node represents the probability, that a particular test will reveal a cocaine user's habit. It correctly identifies cocaine users 95% of the time and correctly identifies non-users 90% of the time. Given that an individual has tested positive, what is the probability that they are a cocaine user?

The NPTs for the two nodes are shown below.

C	$P(C)$
true	0.03
false	0.97

C	true	false
T	$P(T C)$	
true	0.95	0.10
false	0.05	0.90

We want $P(C = \text{true} | T = \text{true})$. Using Bayes Theorem.

$$P(C = \text{true} | T = \text{true}) = \frac{P(T = \text{true} | C = \text{true})P(C = \text{true})}{P(T = \text{true})}$$

We can find $P(T)$ by calculating $P(T,C) = P(T | C)P(C)$ and marginalising out C .

$$\begin{aligned} P(C = \text{true} | T = \text{true}) &= \\ &= \frac{P(T = \text{true} | C = \text{true})P(C = \text{true})}{P(T = \text{true} | C = \text{true})P(C = \text{true}) + P(T = \text{true} | C = \text{false})P(C = \text{false})} \\ &= \frac{0.95 \times 0.03}{0.95 \times 0.03 + 0.10 \times 0.97} = 0.227 \end{aligned}$$

This highlights the very real misperception that often arises, even among qualified physicians, that a seemingly reliable test can have quite counter-intuitive implications. In this case, the number of false positives from the non-drug users (0.10×0.97) is significant compared to the number of positives from the small number of drug users.

A.1 Conditional Independence and Bayesian Nets

Let A , B and C be random variables. If, for all values of A , B and C :

$$P(A | BC) = P(A | B). \quad \text{Equation A-7}$$

then A is said to be *conditionally independent* (CI) of C given B .

There is a notational ambiguity in Equation A-7. The discrete probability distribution on the left hand side is indexed by three nominal variables A , B and C , whereas the discrete probability distribution on the right hand side is only indexed by two variables, A and B . The notation is really just a short hand for:

$$P(A = a | B = b, C = c) = P(A = a | B = b), \forall c \in \text{dom}(C). \quad \text{Equation A-8}$$

where $\text{dom}(C)$ is the domain of the random variable C .

The topology of a BN defines a set of CI conditions. These are summarised by the *Markov condition*: for each node A_i in the network:

$$P(A_i | \pi(i), \varphi(i)) = P(A_i | \pi(i)). \quad \text{Equation A-9}$$

where $\pi(i)$ denotes the parents of A_i , and $\varphi(i)$ denotes its non-descendants. This means that, knowing any information about the non-descendants of a node tells us nothing about the likely value of a node if all of its parents are specified.

A.2 The Chain Rule for Bayesian Networks

The “chain rule” for a set of random variables $R = \{X_1, \dots, X_n\}$ can be stated as:

$$P(X_1 \dots X_n) = \prod_{i=1}^n P(X_i | X_1 \dots X_{i-1}). \quad \text{Equation A-10}$$

This can be proved by examining the case for two variables: $P(X_1 X_2) = P(X_1 | X_2) \times P(X_2)$ and extending it by induction. By combining this with the Markov condition, we are able to derive the chain rule for Bayesian networks.

Theorem A-1 (Chain Rule for Bayesian Networks)

Let $R = \{X_1, \dots, X_n\}$ be a set of random variables, and let $G = (R, E)$ be a directed acyclic graph with edges, E , that satisfies the Markov condition. Then:

$$P(X_1 \dots X_n) = \prod_{i=1}^n P(X_i | \pi(X_i)). \quad \text{Equation A-11}$$

Note: as a notational convenience we represent the prior probabilities of nodes without parents as $P(X_i) \equiv P(X_i | \phi)$, where ϕ is the empty set.

Proofs of Theorem A-1 can be found in any standard text on BNs. Neapolitan [149], for example uses the Markov condition to prove the theorem. Jensen [93] gives a similar proof, but uses d-separation (see below) instead of the Markov condition.

Theorem A-1 is critical to understanding Bayesian Nets. Equation A-11 provides us with the mechanism to calculate the Joint Probability Distribution (JPD) of any BN. If we view the numbers on the right hand side of Equation A-11 laid out in a hyper-cube, with each axis of the hypercube defined by the states of one of the random variables X_i , then we can include evidence in this JPD by masking out hyper-planes corresponding to impossible values and normalising the result. However, the number of calculations involved is exponential in the number of nodes. Much of the research in BNs is therefore directed at calculating marginal distributions without ever having to manipulate the full expression in Equation A-11. These inference algorithms employ two general techniques in order to make the calculation manageable.

1. They take advantage of the CI relations inherent in a BN.
2. They marginalise variables that have been fully incorporated into the calculation but are no longer part of the desired marginal distribution.

Conditional independence in BNs is sometimes expressed in terms of the concept of d-separation, which is introduced in the next section.

A.3 D-Separation

The Markov condition leads to the concept of *d-separation* [149][159]. According to Jensen [93]:

Two distinct variables A and B in a causal network are d-separated if, for all paths between A and B, there is an intermediate variable V (distinct from A and B) such that either
- the connection is serial or diverging and V is instantiated
or
- the connection is converging, and neither V nor any of V's descendants have received evidence.

By “instantiated”, Jensen means that hard evidence has been applied to a node. The definitions of serial, diverging and converging connections are

illustrated in Figure A-3. Two nodes are *d-connected* if they are not d-separated.

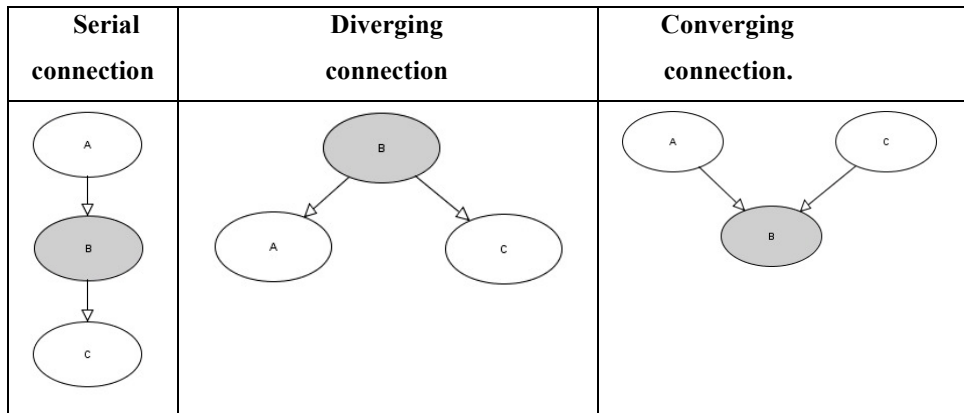


Figure A-3 Node connections

Figure A-3 show the three ways that nodes can be connected in a BN. In a serial connection, evidence entered into *A* will affect *B*, which will in turn affect *C*. *A* and *C* are d-connected. However, if there is evidence entered in *B* then *A* and *C* become d-separated. Changing *A* will have no effect on *C*.

We can see how the Markov condition implies this as follows. Using the chain rule (and therefore the Markov condition):

$$P(A, B, C) = P(A)P(B | A)P(C | B) = P(A, B)P(C | B). \quad \text{Equation A-12}$$

From the definition of conditional probability (Equation A-2), we also have:

$$P(C | A, B) = \frac{P(A, B, C)}{P(A, B)} = \frac{P(A, B)P(C | B)}{P(A, B)} = P(C | B). \quad \text{Equation A-13}$$

i.e. *C* is CI of *A* given *B*. (The above demonstration and those that follow in this section are due to Jensen [93]). Exactly the same argument applies to diverging connections:

$$P(A, B, C) = P(B)P(A | B)P(C | B) = P(A, B)P(C | B). \quad \text{Equation A-14}$$

For converging nodes the situation is different. In Figure A-3, A and C both contribute to B ; they can vary independently and so are d-separated:

$$\begin{aligned}
 P(A,B,C) &= P(A)P(C)P(B | A,C) \\
 \therefore P(A,C) &= \sum_B P(A,B,C) = P(A)P(C) \sum_B P(B | A,C) = P(A)P(C)
 \end{aligned}
 \tag{Equation A-15}$$

Notice that this argument depends on the CPD of B adding to one when summed over B . Adding evidence to B is equivalent to masking off some of the possible states of B , thus removing this property. So if evidence is present in B then A and C lose their independence and become d-connected.

For example, suppose B is a Boolean node, with states $\{b, \underline{b}\}$. With no evidence in B :

$$\begin{aligned}
 P(A,C) &= \sum_B P(A,B,C) = P(A)P(C) \sum_B P(B | A,C) \\
 &= P(A)P(C)(P(b | A,C) + P(\underline{b} | A,C)) = P(A)P(C)
 \end{aligned}
 \tag{Equation A-16}$$

Setting $B=b$, gives the following:

$$\begin{aligned}
 P(A,C) &= \sum_B P(A,B,C) = P(A)P(C) \sum_B P(B | A,C) \\
 &= P(A)P(C)P(b | A,C)\kappa
 \end{aligned}
 \tag{Equation A-17}$$

where κ is a suitable normalising constant. Since $P(b | A,C)$ is not a constant (it varies with A and C), $P(A,C)$ is no longer a simple product of the priors of A and C and so they are not independent.

Example A-3

This a modified version of Example A-1. Three new boolean nodes have been added: W , which says whether it is winter; F , to indicate if there is a frost and R , to say if precipitation is likely.

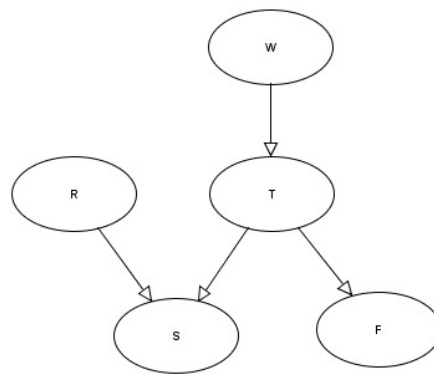


Figure A-4 Updated Winter Snow example

If evidence is entered in T , then the Markov condition ensures that any change in W or F has no effect on S . Normally, precipitation and temperature are independent. However, if we know that the temperature is below zero, and we also know that there is snow on the ground, then it must be true that there has been precipitation. Entering evidence in S has made R and T d-connected.

There is a further subtlety in Example A-3. Suppose we are locked in an office basement and unable to see outside. We know it is cold outside because it was cold when we arrived, but we can't see outside to see if it has been snowing. A colleague then arrives with snow on their boots. From this we infer that it is snowing outside, and therefore that there must have been precipitation. We can represent our new evidence in the BN if we had a further node, B , representing “snow on colleague's boots”, as shown in Figure A-5. This illustrates that it is not only evidence in S which connects R and T , but also evidence in any of S ' descendents.

The importance of d-separation is that it encapsulates our common sense understanding of how causal relationships “ought” to work. This correspondence between d-separation and our expectation of causal reasoning is part of what makes BNs so simple and intuitive to use.

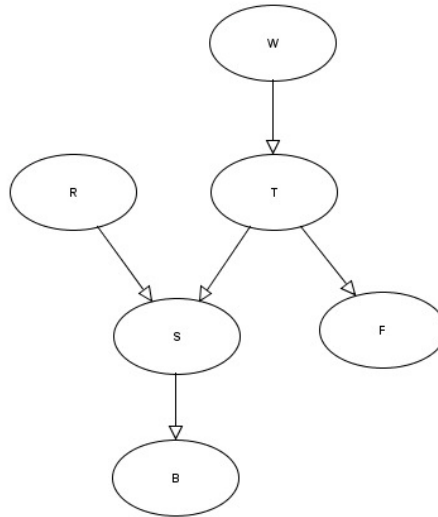


Figure A-5 "Snow on Boots" Node

A.4 Algorithms Used

Most of the models described in this thesis have been evaluated using an exact inference algorithm for discrete BNs called the *Junction Tree Algorithm* (JTA). This algorithm is capable of calculating all the marginal distributions in a large BN without having to store the complete JPD. The algorithm is described using a detailed example in Appendix B.

The AgenaRisk toolset [8] was used to build the agile models described later and to run the JTA. AgenaRisk was chosen for a number of reasons.

1. It provides a wide variety of built-in statistical distributions.
2. CPDs can be constructed as analytic expressions based on a node's parents.
3. The toolset uses a simple point and click interface.
4. Access to the source code was available. This allowed the creation of the custom extensions necessary to build models suited to agile environments.

The extensions mentioned in point 4 are discussed in some detail in the main text, but they can be summarised as follows.

- 1 Database connectivity between the AgenaRisk tool and a series of diverse data sources. This allows the tool to create complex queries using mappings between heterogeneous databases. When combined with these query results, BN models can be generated automatically using some simple rules. Nodes can be updated with evidence using similar query results.
- 2 A scripting capability has been added. This allows both the definition of models and the creation of links between models to be handled outside of the GUI

provided by AgenaRisk. As with all point and click interfaces, what is convenient for one-off tasks, is not always convenient for repetitive tasks. Clearly Agile models, with their inherently iterative nature, involve some degree of repetitiveness. The scripting capability also adds many other highly desirable features, such as common discretisations across many instances of the same model. (The importance of discretisation is discussed in section A.5.6).

- 3 The iterative nature of agile models means that aspects of the model change over time. It is highly desirable to be able to capture these time dependent aspects and view them visually. The ability to extract and display such aspects graphically has also been added to the AgenaRisk tool.

A.5 Advantages of Bayesian Nets

In this section we cover the main benefits of BNs.

A.5.1 Intuitive Models

One of the most obvious benefits of BNs is their intuitive graphical nature. With the correct tools, models can be created manually using a simple point and click interface. The domain expert does not require any understanding of conditional independence, probability theory or Bayes Theorem. All that is required is a qualitative understanding of the variables involved, the causal relationships between them and some means of judging the relative strengths of those relationships.

Causal reasoning is highly compatible with the way humans construct arguments. We naturally regard events as being the result of a network of cause and effect relationships. What is more, humans realise that an event can have more than one possible cause: the grass can be wet because it is raining *or* because of the lawn sprinkler. Similarly, some events can only happen when the correct combination of causes are present: it only snows if there is cloud precipitation *and* it is sufficiently cold. This kind of reasoning is easily handled by BNs as shown in the example below.

Example A-4

We can model multiple alternative causes of the same outcome. Suppose A and C are Boolean variables. If either is true then so is B . This is expressed by the CPD shown on the right below, which in this case is simply the truth table for the Boolean *OR* operation.

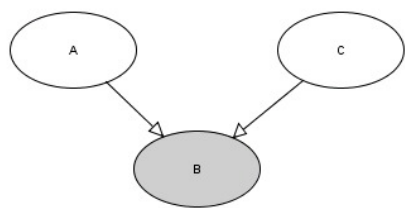


Figure A-6 Boolean OR as a BN

A	T	T	F	F
C	T	F	T	F
B				
T	1	1	1	0
F	0	0	0	1

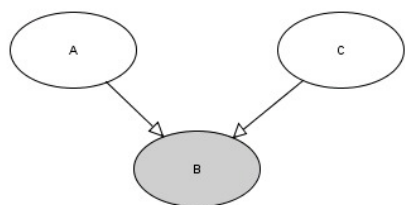


Figure A-7 Boolean AND as a BN

A	T	T	F	F
C	T	F	T	F
B				
T	1	0	0	0
F	0	1	1	1

Similarly, we can model the case where multiple causes must all be present for an outcome to occur. Again, let A and C be Boolean variables. Both must be true for B to be true. This is expressed by the CPD shown on the right, which in this case is simply the truth table for the Boolean *AND* operation.

BNs can encapsulated more complex logical operations than simple AND and OR models. Appendix D, for example, explains the concept of *Noisy Or*.

A.5.2 Learning

In cases where there is sufficient data, it is sometimes possible to learn the structure of the network from a database. Various algorithms exist to determine the best fitting structure for a dataset (see for example [149], [170], [38] and [84]). This can rely on either a searching and scoring approach, or a search for statistical relationships between variables. In general, learning the structure of a BN using a search and score algorithm is NP-hard [41]. Nevertheless, where the data is complex, structure learning may turn out to be more practicable than the domain expert approach. There is rarely sufficient data available in software engineering to perform automated structure learning. For this reason it plays no part in the models discussed here.

Parameter learning, as distinct from structure learning, occurs when the distributions for one or more random variables is unknown and must be learned. Sometimes this uses algorithms such as Maximum Likelihood Estimation based on

tools and data external to the BN model. However it is also possible to get the BN itself to learn its parameters by entering evidence into the model.

Example A-5

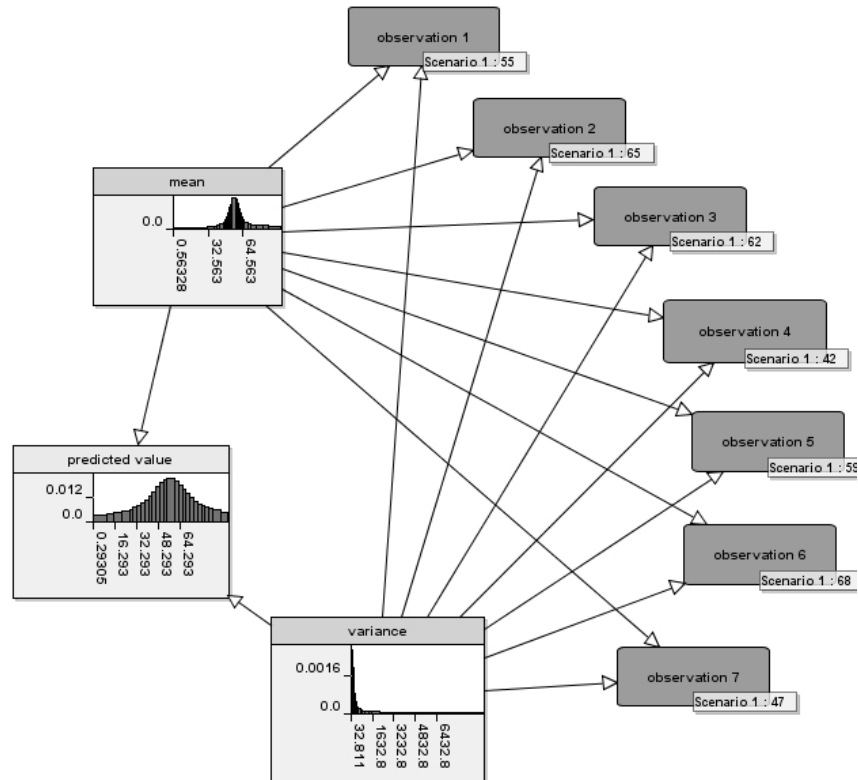


Figure A-8 Learning the parameters of a Normal distribution

Figure A-8 shows how we can learn the parameters of a normal distribution. We have a series of observations which we assume are normally distributed, however the mean and variance of their distribution is unknown. In the model shown, the ‘mean’ node and the ‘variance’ node are set to uninformative priors, which in this case are just uniform distributions. (We have chosen a uniform distribution for simplicity in this example. It should be noted that an uninformed prior is not always best represented by a uniform distribution. See section 3.2.3 for a better real life choice.) Each ‘observation’ node has a CPD which is set to:

$$\text{Normal}(M, V).$$

Where M is the value of the mean node and V is the value of the variance node. Let us look at a single observation variable – call it O . The prior of the observation node is:

$$P(O | M, V).$$

Entering hard evidence into the model masks off all of $P(O | M, V)$, except for the observed state, call this o (lower-case letter “o”). The only non-zero states that remain are:

$$P(O=o | M, V).$$

We can now construct the JPD for the whole model. Let the i 'th observation variable be O_i , with evidence o_i , then the JPD (excluding the predicted variable) is:

$$P(M)P(V)\prod_i P(O_i = o_i | M, V)$$

The distributions $P(M)$ and $P(V)$ are both uniform, so they can be treated as constants and ignored, this leaves only the product of the observation variables. We marginalise each O_i by summing over the states of O_i , but only one of these values is non-zero: the one with the selected evidence, so summing has no effect. Subject to a constant, we therefore have:

$$P(M, V) = \prod_i P(O_i = o_i | M, V)$$

Those combinations of values of M and V with the highest probability of explaining the evidence o_i (as determined by the CPD of will O_i) will dominate this distribution. I.e. We have learned the most likely values of M and V .

We return to parameter learning in section 7.1 where we show how to learn parameters in Dynamic Bayesian Nets (discussed below).

A.5.3 Reasoning Under Uncertainty

One of the problems with simple regression based models is their inability to deal with uncertainty. This is something that human decision makers must deal with on a daily basis. We rarely have complete evidence in any given situation and what evidence we do have will normally have an uncertainty attached to it. BNs have the ability to deal with both missing data and uncertain data.

When evidence is missing, the prior distributions of each node are propagated through the network. Where no knowledge whatsoever is known about the prior state of a node then the prior distribution can be set to an uninformed prior. In many cases however, more detailed information will be available. This may involve the exact form of the distribution as garnered from either theoretical or empirical studies, or it may be as simple as a hunch, based on experience, that some values are more likely than others.

We have already seen how hard evidence is incorporated into the model in Example A-5. Some soft evidence, such as *findings* (where certain values are excluded), can also be incorporated using this mechanism. Arbitrary distributions can also be entered by setting each value of the evidence potential, Λ , to appropriate real number values.

A.5.4 Heterogeneous Modelling

By *heterogeneous modelling* we mean the ability to include multiple types of data in the same model. The data can be said to be heterogeneous in at least three distinct senses.

1. It may belong to numeric, ordinal or nominal scales.
2. If numeric, then it can be continuous or discrete (mixed continuous and discrete models are referred to as *hybrid* models).
3. Data can include observations, prior knowledge or beliefs.

Point 3 has been largely covered above in *Reasoning Under Uncertainty*. Although we shall mix continuous and discrete nodes in the models described here, the continuous distributions are in fact always approximated by splitting the distribution into discrete ranges. This leaves us only point 1 left to consider.

None of the theory described so far, including the propagation algorithm, depends on any properties of the states of nodes. Provided a (conditional) probability distribution can be defined across the complete set of discrete node states, we can construct a consistent BN and propagate any evidence entered. We will therefore illustrate with some examples how CPDs can be generated for various combinations of data types.

Example A-6, Continuous Variable with Ordinal Parent

Let A be a continuous variable with a single ordinal parent, B . Let B have the set of states:

$$b_1, \dots, b_n, i < j \Rightarrow b_i < b_j$$

and let A be discretised into the ranges:

$$[a_1..a_2), [a_2..a_3), \dots, [a_{m-1}..a_m), i < j \Rightarrow a_i < a_j.$$

For each value b_i and range $[a_k..a_{k+1})$ we can define a probability $P(A=[a_k..a_{k+1}) | b_i)$, such that

$$\sum_k P(A=[a_k..a_{k+1}) | b_i) = 1.$$

$P(A=[a_k..a_{k+1}) | b_i)$ therefore forms a CPD for A . We can model the ordinal variable by another continuous variable C , discretised into ranges:

$$[c_1..c_2), [c_2..c_3), \dots, [c_n..1), i < j \Rightarrow c_i < c_j, c_{i+1} - c_j = 1/n.$$

The ranges preserve the ordering of the ordinal variable B and each range $[c_i..c_{i+1})$ is then associated with the ordinal value b_i . With this association in place we can use any function $\varphi : A \times C \rightarrow \mathfrak{R}$, suitably normalised, as a shortcut to setting up the CPD for A . For example:

$$P_k^i = \varphi\left(\frac{a_k + a_{k+1}}{2}, \frac{c_i + c_{i+1}}{2}\right).$$

I.e. we can use the value of φ defined at the midpoint of the corresponding ranges. The net effect of this is that we can define the CPD of a continuous node with an ordinal parent in terms of standard statistical distributions. For example, if $\text{Normal}(\mu, \sigma)$ represents a Normal distribution with mean μ and standard deviation σ , then we can define:

$$P(A|B) = \text{Normal}(B, B/2).$$

This creates a Normal distribution over the values of A for a given value of B with the mean and standard deviation specified and suitably normalised so that the probability mass sums to 1.

Clearly the technique used in Example A-6 to associate a continuous variable with an ordinal node can be used wherever an ordinal node appears. This allows us to assign standard statistical distributions to ordinal nodes themselves, or to ordinal nodes with numeric or ordinal parents. Wherever a statistical distribution is specified over a continuous variable with an ordinal parent, a procedure similar to the above should be assumed.

Example A-7, Continuous Variable with Nominal Parent

Let A be a continuous variable with the states shown in Example A-6. Let A have a single nominal scale parent, B , with the states:

$$b_1, \dots, b_n$$

The technique used in Example A-6 can no longer be used. The states of B are no longer ordered, therefore it is impossible to setup a unique isomorphism between B and an ordered partition of the range $[0..1]$. We can still define continuous distributions over A however. Now we use a set of functions $\varphi_i : A$

→ \mathfrak{R} , $i = 1..n$, one for each value of B . These are assigned to probabilities as follows:

$$P(A = [a_k..a_{k+1}] | b_i) = \varphi_i \left(\frac{a_k + a_{k+1}}{2} \right)$$

We call φ_i a *partition of B over A* . The set of functions $\{\varphi_i\}$ is known as a *partitioned expression*.

Partitions can be defined using ordinal as well as nominal variables. This is particularly useful when the relationship between the parent ordinal variable, as represented by its associated partition of the $[0..1]$ range, and the continuous child variable is not easy to express as an analytic function.

A.5.5 Induction and Abduction

The ability to perform abductive, as well as inductive reasoning is closely related to parameter learning. Abductive reasoning occurs when an event is known to have taken place and we wish to ascertain the probable cause. We illustrate this with an example of “explaining away”.

Example_A A-8

Consider the case of a student sitting an exam. Assuming that appropriate precautions exist to prevent cheating, there are two possible ways to get a good score. The student can be hard working, or alternatively, they might be lucky and just happen to have read up on the subjects that appear in the exam paper.

Let E be the event “Exam Passed”, with possible causes “Good Student”, S , and “Luck”, L . We assume that we know nothing about the student in advance and that they are equally likely to be hard working and lucky. Both causes are therefore given uniform priors. If the student is hard working ($S = \text{true}$) then they will normally get a good score. However, it is possible that they are unlucky that day ($L = \text{false}$) and they feel unwell. In this case there is a small probability that they will get a poor exam score.

If the student is lazy ($S = \text{false}$) then they will almost certainly get a poor exam score, unless they happen to be lucky on the day, in which case there is

a small possibility that they will get a good score. This information is summarised in the probability distributions shown below.

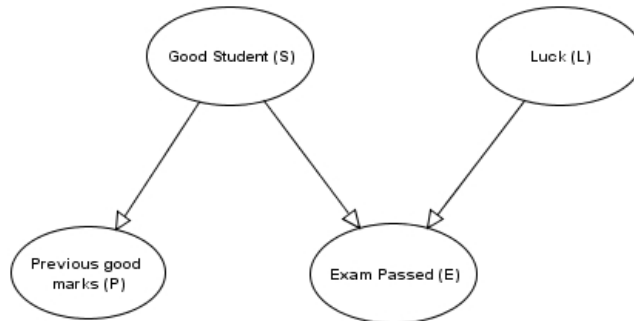


Figure A-9 Explaining away

	$P(S)$	$P(L)$
F	0.5	0.5
T	0.5	0.5

S	F	F	T	T
L	F	T	F	T
P(E S,L)				
E = False	1	0.9	0.1	0
E = True	0	0.1	0.9	1

The distribution $P(E|S,L)$ is symmetric with respect to S and L , so initially the model shows a 50/50 chance of the student achieving a good mark. Suppose the exam is now marked, and the student achieves a good score. This is equivalent to zeroing out the $E = \text{false}$ line in the $P(E|S,L)$. We can get from the CPD, $P(E|S,L)$, to the JPD $P(E,S,L)$ as follows.

$$P(E,S,L) = P(E|S,L)P(S)P(L)$$

Since $P(S)$ and $P(L)$ are equal, the $P(E,S,L)$ will have the same “shape” as $P(E|S,L)$. The bulk of the probability mass is therefore concentrated in the bottom right corner of $P(E,S,L)$ where $S = \text{true}$ and L remains almost evenly split between true and false:

S	F	F	T	T
L	F	T	F	T
P(E,S,L)				

E = False	0	0	0	0
E = True	0	0.1	0.9	1

This is as we would expect: if the student achieves a good score then they are probably a hard worker, with luck playing little part on the exam day.

Now suppose we have another indicator of whether the student is a good student. The node “Previous good marks”, P , has the table shown below.

S	F	T
P(P S)		
P = False	0.9	0.1
P = True	0.1	0.9

If we now discover that the student had previously gained good marks then the row corresponding to $P = \text{False}$ gets zeroed out in the above table. It is now almost certainly the case that the exam has been passed because the student is hard working, and not because of luck. We have “explained away” the “lucky” explanation.

A.5.6 Discretisation

The propagation algorithm outlined in Appendix B produces exact results for discrete variables. Exact solutions are also possible in hybrid models (where discrete and continuous variables are intermixed) but only by restricting the continuous variables to Gaussian distributions [119]. This is too strong a restriction for most models.

Example A-6, in the section on *Heterogeneous Modelling*, showed how we could approximate a continuous distribution by splitting it into a finite set of discrete ranges. Other approximation techniques have been proposed, and these are discussed in [153], however discretisation remains the only simple, general-purpose algorithm available.

Clearly, getting the discretisation right is crucial to the success of any model. Figure A-10 shows the marginal distributions for three nodes initialised with the same distribution: Normal (5, 1). The leftmost graph illustrates the result of an inappropriate choice of discretisation, while a better choice of discretisation results in the middle graph.

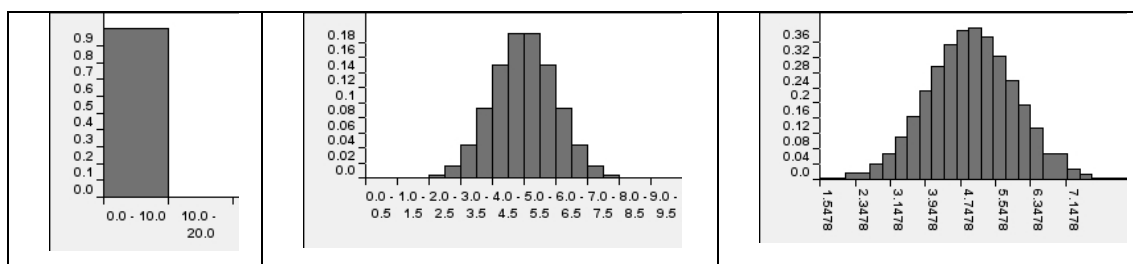


Figure A-10 Bad, good and dynamic discretisation.

Various undesirable effects can flow from a poor discretisation.

1. The shape of the distribution can be entirely misleading, as is the case in Figure A-10.
2. Summary statistics, such as mean, median and variance, become inaccurate.
3. Evidence entered into a poor discretisation becomes less precise. For example, if a variable is numeric and covers the range $[0, 100]$ and it is discretised into 5 equal ranges, then entering a value of 10 (say) means that the $[0, 20)$ range will be selected. A more refined discretisation would result in a much smaller range being selected.

In iterative models, such as dynamic BNs or object oriented BNs, these errors are cumulative, leading to serious distortions in the calculations.

This would all seem to argue that higher resolution discretisations will result in more accurate models, and indeed this is the case. However there is a computational penalty to pay for this higher resolution. A large number of states in a node results in a similarly large number of states in every clique which includes the node, causing the multiplication of potentials and their marginalisation to take longer. There is therefore an incentive to only use as many states as are necessary, concentrating the high resolution states at locations of highest probability mass.

The problem facing modellers is that it is not always clear in advance where the main body of the probability mass will reside. This can be particularly problematic for dynamic BNs (discussed later), where the model includes a timeseries element. In this case, the probability mass will often move across timeslices, so that a discretisation which is appropriate for one timeslice turns out to be inappropriate for another.

A possible solution, involving dynamic discretisation, has been proposed by Neil, Tailor and Marquez [153]. The dynamic discretisation algorithm begins with a very small number of states, and by continually splitting and combining states, arrives at a

discretisation which minimizes the relative entropy error [113]. A marginal distribution resulting from this algorithm forms the third graph in Figure A-10.

A.5.7 Discretisation and Ranked Nodes

In Example A-7 we introduced the concept of an ordinal variable represented by an ordered set of discrete ranges which partition the range $[0, 1]$. We refer to a node containing such a variable as a *ranked node*. Ranked nodes are discussed extensively in [65]. They typically include a set of 3, or 5 values with labels such as

{Very Low, Low, Medium, High, Very High},

although other values and numbers of labels are possible.

As pointed out in the previous section, entering evidence into any discrete partition of a continuous variable results in the entire discrete range containing the evidence being selected. In effect, the model introduces an extra level of uncertainty in the measurement.

Ranked nodes are typically used to express expert judgement. This judgement is usually subjective and uncertain. If this were not the case, if it were objective and definite in value, then we would use a continuous variable instead. In the case of ranked nodes therefore, this uncertainty is a desirable attribute. More information on ranked nodes and the important role that they play in eliciting CPDs from experts is contained in the section on Ranked Nodes, TNormal Distributions, and Weighting Functions.

Appendix B – The Junction Tree Algorithm

Note – because of the size of some of the probabilistic expressions in this chapter, I am using a more compact notation for probability distributions. The notation is summarised in Table B-1.

	Standard Notation	Component Notation
Probability distribution of a random variable A .	$P(A)$	P_A
Conditional probability distribution of A given B .	$P(A B)$	P_A^B
Probability that variable A takes the value “ a ”.	$P(A = a)$	$P_{A=a}$ or P_a
Joint Probability of a set of random variables $\{X_1, \dots, X_n\}$	$P(X_1, \dots, X_n)$	$P_{1..n}$
Potential indexed by a random variable.	$m(A)$ or $\psi(A)$	m_A

Table B-1 Notation used in this appendix.

This appendix provides an outline of an algorithm that allows evidence to be propagated consistently while allowing marginal probabilities to be calculated from much smaller arrays of numbers. This is often called the “Hugin” algorithm after a piece of software in which it was first implemented. Hugin is an extension of the Lauritzen and Spiegelhalter algorithm [120]. Another alternative is due to Shenoy and Shafer [186]. All share the property that a secondary graph is created from the BN, with local potentials updated via message passing. The three methods are compared in [122].

Rather than provide a formal proof, we will illustrate the method using an example from Huang and Darwiche [87]. A rigorous derivation of the algorithm can be found in [94][95]. The example BN is shown in Figure B-1. For simplicity, each node is a boolean variable with two states: true and false.

We could calculate any probability distribution for the example by first creating the JPD:

$$P_{ABCDEFGH} = P_A P_B^A P_C^A P_D^B P_E^C P_F^{DE} P_G^C P_H^{EG} . \quad \text{Equation B-1}$$

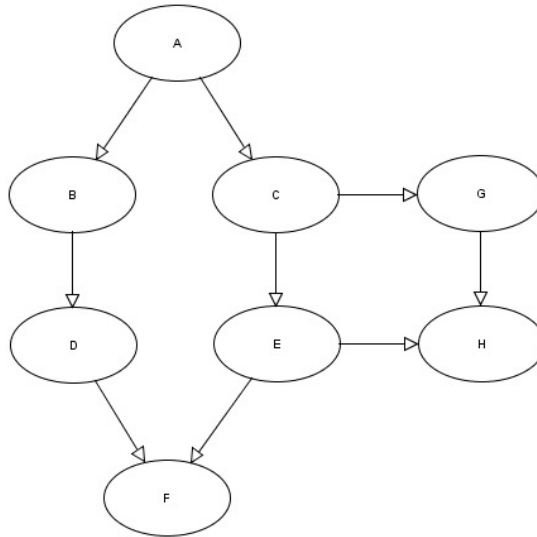


Figure B-1 Huang and Darwiche example

We can then marginalise to calculate the marginal probability distributions for each node. In this case, with only $2^8 = 256$ entries, this might be a practical approach, but in general this is not the case. The Joint Probability Distribution (JPD) of a ten node network, with each node having ten states, is a ten dimensional array of numbers containing a total of 10^{10} entries. Clearly, constructing the full JPD for any non-trivial BN is not a practicable task.

The key to simplifying the task is to realise that each random variable only appears in a small number of potentials. Summing over all variables that do not appear in those potentials results in an expression that is much simpler. For example, suppose we wish to calculate the marginal distribution for P_H in Equation B-1. We would have to sum over all the remaining variables:

$$P_H = \sum_{ABCDEFG} P_A P_B^A P_C^A P_D^B P_E^C P_F^{DE} P_G^C P_H^{EG} . \quad \text{Equation B-2}$$

Most of the distributions in Equation B-2 do not involve H . Consequently, we could pre-compute many of these values to create the potential:

$$m_{EG} = \sum_{ABCD} P_A P_B^A P_C^A P_D^B P_E^C P_F^{DE} P_G^C . \quad \text{Equation B-3}$$

Calculating P_H then becomes much simpler:

$$P_H = \sum_{EG} P_H^{EG} m_{EG} . \quad \text{Equation B-4}$$

We would therefore like to create a second graph, derived from Figure B-1, which has a node with the potential:

$$m_{EGH} = P_H^{EG} m_{EG} = P_{EGH} . \quad \text{Equation B-5}$$

This potential can also be summed over either of its other dependent variables to give P_E and P_G . i.e. $m_{EGH} = P_{EGH}$.

B.1 Moralising

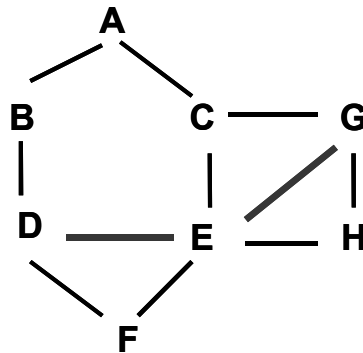


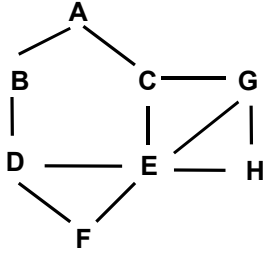
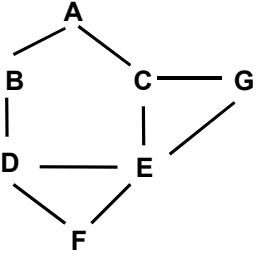
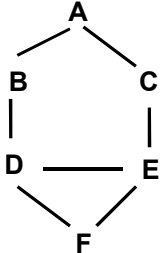
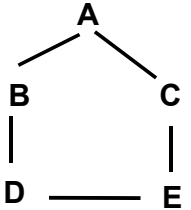
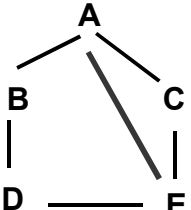
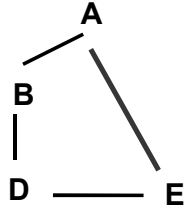
Figure B-2 Moralised graph, G_M

We begin by “moralising” the graph, G , to produce a new graph G_M (Figure B-2 – this figure and many of those that follow for this example, are reproduced with the kind permission of Prof. Martin Neil). We moralise the graph by removing the directions and linking together all parents of a common child. The need to moralise the graph arises from the fact that there are domains for potentials in Equation B-1 which are not represented by edges on the graph. For example P_F^{DE} has domain $\{D,E,F\}$. Moralisation guarantees that these three nodes will end up as part of the same *clique* in the (non-unique) triangulation step explained below. This in turn ensures that there will be a clique to which the potential P_F^{DE} can be assigned.

B.2 Triangulation

Once the moral graph has been obtained, the next step is to *triangulate* the graph. We do this by selecting a node V , and pairwise connecting it and all of its neighbours. This set of neighbours forms a *cluster*. V is chosen so as to minimise the number of connections added to create the cluster and to minimise the total number of states of the cluster. The domain of each cluster is added to a domain list. The node, V , is then removed from the graph.

The order of node removal is not unique. One possibility in the example is as : shown in Figure B-3.

 <p style="text-align: center;">1</p> <p>Node H is already pairwise connected to all its neighbours. Together with its neighbours it forms a cluster with domain {G,E,H}. Remove H.</p>	 <p style="text-align: center;">2</p> <p>Node G is already pairwise connected to all its neighbours. Together with its neighbours it forms a cluster with domain {C,E,G}. Remove G.</p>	 <p style="text-align: center;">3</p> <p>Node F is already pairwise connected to all its neighbours. Together with its neighbours it forms a cluster with domain {D,E,F}. Remove F.</p>
 <p style="text-align: center;">4</p> <p>Pairwise connect node C with its neighbours A and E.</p>	 <p style="text-align: center;">5</p> <p>Node C is now part of a cluster with domain {A,C,E}. Remove C.</p>	 <p style="text-align: center;">6</p> <p>Pairwise connect node B with its neighbours A and D.</p>

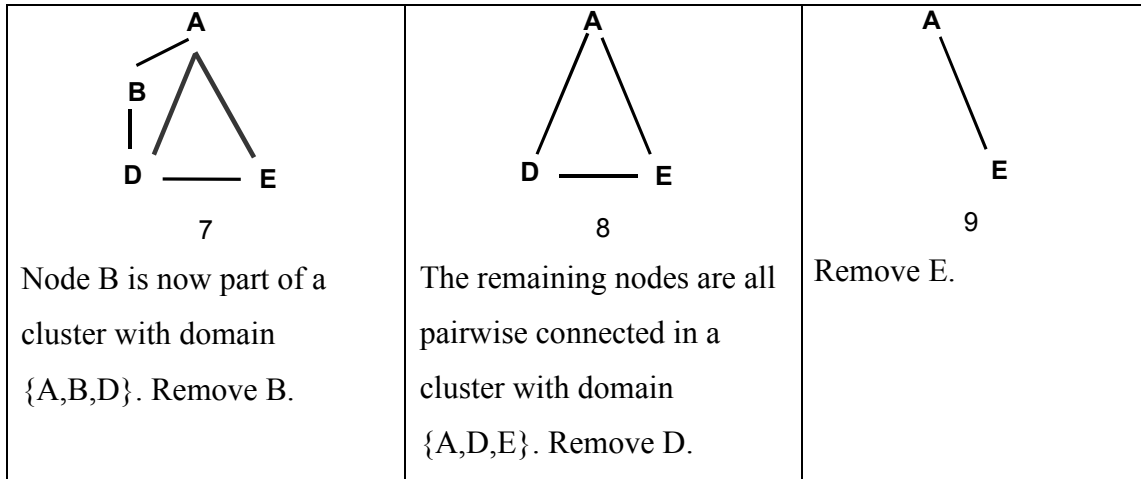


Figure B-3 Triangulation and node elimination

The remaining domain sets, {A,E} and {A} are excluded because they are subsets of {A,D,E}. This last step ensures that only maximal complete sets, or cliques, are included in the domain set. This results in the domain set: GEH, CEG, DEF, ACE, ABD, ADE. If we include all the links added during triangulation to the moral graph, we obtain the graph shown in Figure B-4, the triangulated graph.

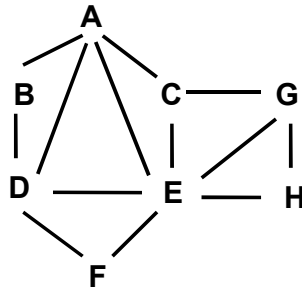

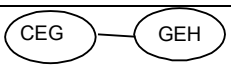
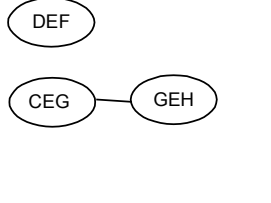
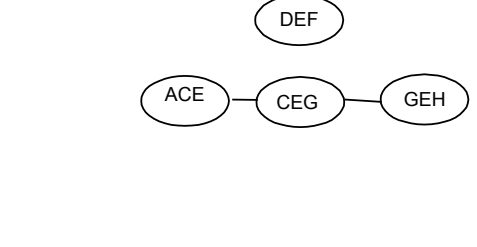
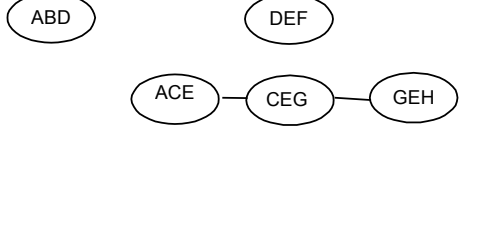
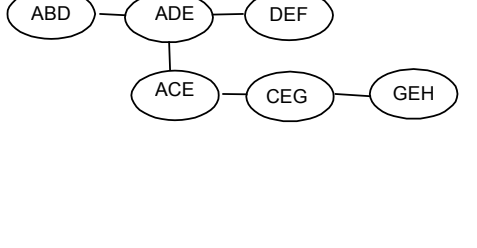


Figure B-4 The triangulated graph.

B.3 Join Trees and Junction Trees

Next, we build a join tree, J . Each node in J corresponds to one of the cliques identified during the triangulation stage. We build the join tree one clique at a time in the order in which we discovered them.

	The first clique we discovered had domain {G,E,H}, so we create a node in J corresponding to that clique.
	The next clique discovered had domain {C,E,G}. This shares an edge in the triangulated graph with clique {G,E,H}. In J , this shared edge becomes a link between the nodes corresponding to the two cliques.

	<p>The next clique discovered had domain $\{D,E,F\}$. This did not share any edges in the triangulated graph with any of the cliques added to J so far. For the moment this clique remains disconnected.</p>
	<p>The next clique discovered had domain $\{A,C,E\}$. This shared an edge in the triangulated graph with clique $\{C,E,G\}$. In J, this shared edge becomes a link between the nodes corresponding to the two cliques.</p>
	<p>The next clique discovered had domain $\{A,B,D\}$. This did not share any edges in the triangulated graph with any of the cliques added to J so far. For the moment this clique remains disconnected.</p>
	<p>Finally we discovered the clique with domain $\{A,D,E\}$. This shared edges in the triangulated graph with the cliques $\{A,B,D\}$, $\{D,E,F\}$, and $\{A,C,E\}$. Each of these edges are represented by links in J.</p>

The join tree J has the property that, given two cliques, say DEF and GEH in J , all nodes in the path between DEF and GEH contain $DEF \cap GEH$. We call this the *join tree property*. A corollary of this is that there exists a set of cliques for any given node (E say), such that E appears in all paths between any two cliques in the set and nowhere else. In other words, E exists exclusively in one “joined up” section of J . This property is crucial for the message passing algorithm explained later.

Having obtained the join tree, we now separate each clique by a separation set, or *sepset*. Sepsets contain the intersection of neighbouring cliques. The resulting join tree, referred to as a *Junction Tree*, is shown in Figure B-5.

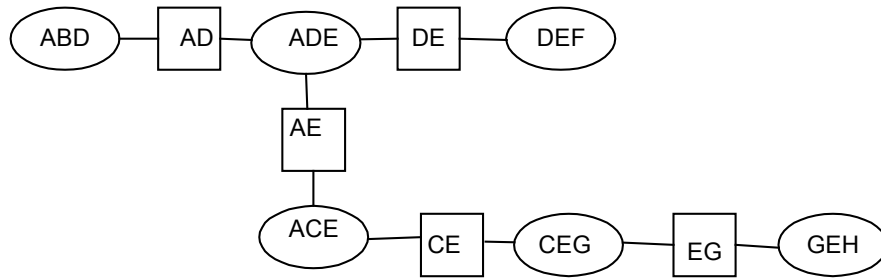


Figure B-5 Junction tree with cliques and sepsets

Each node, V , from the original BN is assigned to a clique in J which contains both V and $\pi(V)$ (the latter can be empty if V has no parents). The moralisation step guarantees that such a clique exists. The potentials associated with all variables assigned to a clique are then multiplied together to form the initial clique potential. This assignment is not necessarily unique. An example is given in Table B-1. P_A could have been assigned to any of the cliques containing A . Sepsets and unassigned cliques are initialised to a potential with all values set to one.

Clique X	ASSIGNED NODES	m_X
ABD	B, D	$P_B^A P_D^B$
ADE	A	P_A
DEF	F	P_F^{DE}
ACE	C	P_C^A
CEG	E, G	$P_E^C P_G^C$
GEH	H	P_H^{EG}

Table B-1 Initial clique potentials

B.4 Message Passing

Let X and Y be adjacent cliques separated by a sepset S . Let them have potentials c_X , c_Y and r respectively. Denote the modified versions of these potentials by c'_X , c'_Y and r' respectively. A *message* passed from X to Y consists of the following steps.

1. $r' = \sum_{X/S} c_X$. i.e. The potential c_X is projected onto S by marginalising out all variables in X that are not in S .

2. $c'_Y = c_Y \frac{r'}{r}$. i.e. The projection from \mathbf{X} is multiplied into \mathbf{Y} , but with the previous sepset potential divided out. Initially the sepset potentials are all set to one.

A single clique is selected. Messages are then passed from all outer cliques to the selected clique, starting from the cliques that are farthest away from the selected clique. This is called the *collect evidence* phase. Messages are then sent from the selected clique to all other cliques. This is called the *distribute evidence* phase.

In what follows, we need to keep track of multiple versions of various potentials. We denote the version of a potential with a numeric superscript. So c_x^1 is version one of a potential, c_x^2 is version 2 and so forth. The subscript is simply the set of nodes assigned to a clique or a sepset. These are always unique. We shall also distinguish between clique potentials, c , and sepset potentials, s .

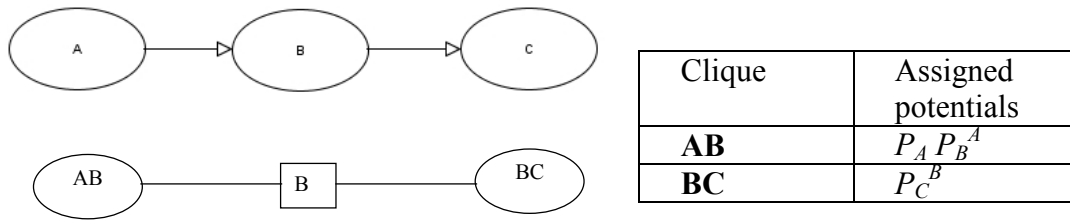


Figure B-6 A simple BN with its junction tree

The easiest way to see how the algorithm works is to give a very simple example. Figure B-6 shows a simple BN, together with a valid junction tree and the potentials assigned to its cliques.

We choose clique AB to be the starting clique. Messages are then sent in the following order.

1. AB to BC: $s_B^2 = \sum_A P_A P_B^A$ $c_{BC}^2 = c_{BC}^1 \frac{s_B^2}{s_B^1} = P_C^B \sum_A P_A P_B^A = \sum_A P_{ABC} = P_{BC}$
2. BC to AB: $s_B^3 = \sum_C c_{BC}^1 s_B^2$ $c_{AB}^2 = c_{AB}^1 \frac{s_B^3}{s_B^2} = P_A P_B^A \sum_C c_{BC}^1 = \sum_C P_{ABC} = P_{AB}$

As an optimisation, we can remove the division by s_b^l in stage one above.

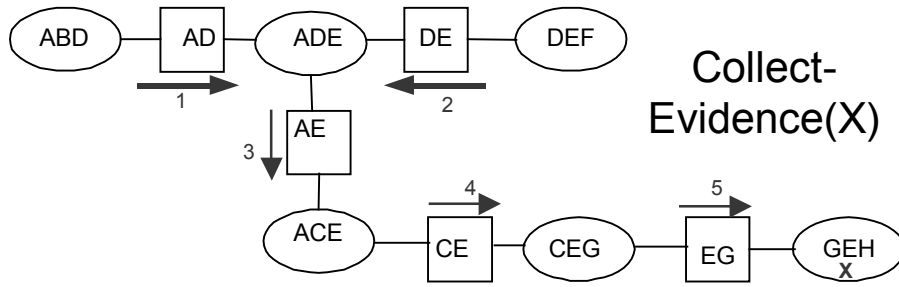
Theorem

The message passing algorithm sets the potential of each clique equal to the JPD of that clique.

Justification

A rigorous proof of the above theorem is available in [94][95]. Instead, we illustrate the algorithm by applying it to the example used throughout this section. It should be apparent that the mechanism is sufficiently general to cope with any junction tree.

We choose the GEH clique as the starting clique. Messages are then passed from outer cliques in the following order:



1. ABD to ADE:	$s_{AD}^2 = \sum_B c_{ABD}^1$	$c_{ADE}^2 = c_{ADE}^1 \frac{s_{AD}^2}{s_{AD}^1} = \sum_B c_{ADE}^1 c_{ABD}^1$
2. DEF to ADE:	$s_{DE}^2 = \sum_F c_{DEF}^1$	$c_{ADE}^3 = c_{ADE}^2 \frac{s_{DE}^2}{s_{DE}^1} = \sum_{BF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1$
3. ADE to ACE:	$s_{AE}^2 = \sum_D c_{ADE}^3$	$c_{ACE}^2 = c_{ACE}^1 \frac{s_{AE}^2}{s_{AE}^1} = \sum_{BDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1$
4. ACE to CEG	$s_{CE}^2 = \sum_A c_{ACE}^2$	$c_{CEG}^2 = c_{CEG}^1 \frac{s_{CE}^2}{s_{CE}^1} = \sum_{ABDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1$
5. CEG to GEH	$s_{EG}^2 = \sum_C c_{CEG}^2$	$c_{GEH}^2 = c_{GEH}^1 \frac{s_{EG}^2}{s_{EG}^1} = \sum_{ABCDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1$

The point at which the summations take place is important. Take, for example, the summation over A in step 4. This happens because the algorithm says to sum over all nodes included in the clique **ACE** which are not included in the sepset **CE**. The join

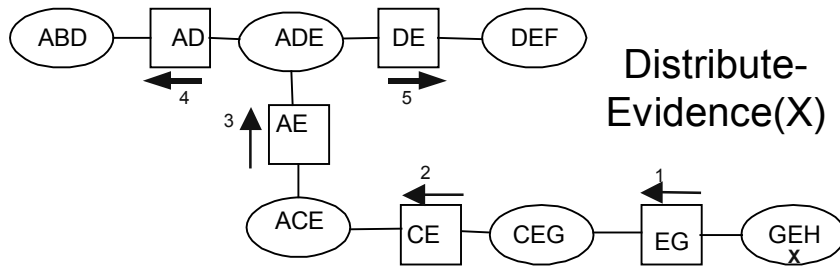
tree property ensures that A can appear only in cliques and sepsets which have already passed a message. I.e. The summation is taking place over *all* potentials that include A . In particular, it guarantees that A does not exist in the clique selected as the origin for message passing, whose nodes we must not sum over.

The final potential, c^2_{GEH} , contains the product of all the initial potentials, and so contains the product of all the probability distributions from the model. So the product of all the probability distributions has been summed over all nodes which are not included in the selected clique. I.e.

$$c^2_{GEH} = \sum_{ABCDF} P_{ABCDEFGH} = P_{GEH}$$

We could repeat this procedure for each clique in the tree. However as we shall see below, the distribute-evidence phase is sufficient to create JPDs in all of the remaining cliques.

Having collected evidence from all other nodes, we now distribute messages outwards from GEH in the following order.



1. GEH to CEG $s^3_{EG} = \sum_H c^2_{GEH}$

$$\begin{aligned} c^3_{CEG} &= c^2_{CEG} \frac{s^3_{EG}}{s^2_{EG}} = \sum_{ABDF} c^1_{ADE} c^1_{ABD} c^1_{DEF} c^1_{ACE} c^1_{CEG} \frac{\sum_{ABCDFH} c^1_{ADE} c^1_{ABD} c^1_{DEF} c^1_{ACE} c^1_{CEG} c^1_{GEH}}{\sum_{ABCDF} c^1_{ADE} c^1_{ABD} c^1_{DEF} c^1_{ACE} c^1_{CEG}} \\ &= \sum_{ABDFH} c^1_{ADE} c^1_{ABD} c^1_{DEF} c^1_{ACE} c^1_{CEG} c^1_{GEH} = P_{CEG} \end{aligned}$$

There is another way to construct this:

$$m^1 = \sum_H c_{GEH}^1$$

$$c_{CEG}^3 = c_{CEG}^2 m^1 = \left(\sum_{ABDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 \right) \left(\sum_H c_{GEH}^1 \right) = P_{CEG}$$

This has removed the division. Division takes twice as much time as multiplication, so this calculation should be three times as fast. It has also removed the need to store s_{EG}^2 . Although not described in Huang and Darwiche [87], this optimisation is described in Jensen [93].

2. CEG
to ACE

$$s_{CE}^3 = \sum_G c_{CEG}^3$$

$$c_{ACE}^3 = c_{ACE}^2 \frac{s_{CE}^3}{s_{CE}^2} = \sum_{BDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 \frac{\sum_{ABDFGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1}{\sum_{ABDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1}$$

$$= \sum_{BDFGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1 = P_{ACE}$$

ALTERNATIVELY:

$$m^2 = \sum_G m^1 c_{CEG}^1$$

$$c_{ACE}^3 = c_{ACE}^2 m^2 = \sum_{BDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 \left(\sum_{GH} c_{CEG}^1 c_{GEH}^1 \right) = P_{ACE}$$

We have replaced one multiplication and one division by two multiplications. This should run 50% faster. We no longer need s_{CE}^2 .

3. ACE
to ADE

$$s_{AE}^3 = \sum_C c_{ACE}^3$$

$$c_{ADE}^4 = c_{ADE}^3 \frac{s_{AE}^3}{s_{AE}^2} = \sum_{BF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 \frac{\sum_{BCDFGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1}{\sum_{BDF} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1}$$

$$= \sum_{BCDFGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1 = P_{ADE}$$

Alternatively:

$$m^3 = \sum_C m^2 c_{ACE}^1$$

$$c_{ADE}^4 = c_{ADE}^3 m^3 = \sum_{BCFGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{CEG}^1 c_{GEH}^1 c_{ACE}^1 = P_{ADE}$$

This calculation will typically be 50% faster. No longer need s_{AE}^2 .

4. ADE
to ABD

$$s_{AD}^3 = \sum_E c_{ADE}^4$$

$$c_{ABD}^2 = c_{ABD}^1 \frac{s_{AD}^3}{s_{AD}^2} = c_{ABD}^1 \frac{\sum_{BCEFGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{CEG}^1 c_{GEH}^1 c_{ACE}^1}{\sum_B c_{ABD}^1}$$

$$= \sum_{CEFGH} c_{ADE}^1 c_{DEF}^1 c_{CEG}^1 c_{GEH}^1 c_{ACE}^1 c_{ABD}^1 = P_{ABD}$$

Alternatively:

$$s_{AD}^3 = \sum_E c_{ADE}^1$$

$$s_{AE}^3 = m^3$$

$$c_{ABD}^2 = c_{ABD}^1 s_{AD}^3 s_{AE}^3 s_{DE}^2 = c_{ABD}^1 \left(\sum_E c_{ADE}^1 \right) \left(\sum_H c_{GEH}^1 \sum_G c_{CEG}^1 \sum_C c_{ACE}^1 \right) \left(\sum_F c_{DEF}^1 \right)$$

$$= \sum_{CEFGH} c_{ADE}^1 c_{DEF}^1 c_{CEG}^1 c_{GEH}^1 c_{ACE}^1 c_{ABD}^1 = P_{ABD}$$

We have multiplied the potentials of the **ABD** clique, the **ADE** clique and the **AE** and **DE** branches. This has 3 mults instead of 1 mult + 1 div. Should take roughly the same time to compute. Note that we do have to retain s_{DE}^2 . We also have to retain s_{AD}^2 for the next branch below.

5. ADE
to DEF

$$s_{DE}^3 = \sum_A c_{ADE}^4$$

$$c_{DEF}^2 = c_{DEF}^1 \frac{s_{DE}^3}{s_{DE}^2} = c_{DEF}^1 \frac{\sum_{ABCGH} c_{ADE}^1 c_{ABD}^1 c_{DEF}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1}{\sum_F c_{DEF}^1}$$

$$= \sum_{ABCGH} c_{ADE}^1 c_{ABD}^1 c_{ACE}^1 c_{CEG}^1 c_{GEH}^1 c_{DEF}^1 = P_{DEF}$$

Alternatively:

$$\begin{aligned}
s_{DE}^3 &= \sum_A c_{ADE}^1 \\
c_{DEF}^2 &= c_{DEF}^1 s_{DE}^3 s_{AE}^3 s_{AD}^2 = c_{DEF}^1 \left(\sum_A c_{ADE}^1 \right) \left(\sum_H c_{GEH}^1 \sum_G c_{CEG}^1 \sum_C c_{ACE}^1 \right) \left(\sum_B c_{ABD}^1 \right) \\
&= \sum_{ABCGH} c_{ADE}^1 c_{DEF}^1 c_{CEG}^1 c_{GEH}^1 c_{ACE}^1 c_{ABD}^1 = P_{DEF}
\end{aligned}$$

B.5 Evidence Propagation

Evidence propagation in junction trees is accomplished by setting to zero all values in a clique potential which do not correspond to observed values and re-running the message passing algorithm. We can denote this by introducing additional potentials called *evidence potentials*. These can be set to:

1. all ones, for no evidence;
2. one for a particular node state and zero otherwise (*hard evidence* or *observation*);
3. a set of values in the range [0, 1] (*soft evidence*).

Clique potentials are then multiplied by the evidence potentials before message passing begins. A normalisation stage must be added after message passing completes.

We will normally denote an evidence potential by an uppercase Greek letter, indexed by the random variable that the evidence applies to. Thus, evidence in node C in Figure B-1 might be denoted by Λ_C . The clique potential c_{CEG} then becomes:

$$c_{CEG} = P_E^C P_G^C \Lambda_C$$

Note that it is only necessary to update one of the cliques with an evidence potential as the message passing algorithm will cause all other potentials to update as well. The net effect of these manipulations is to mask off impossible values.

Appendix C – AgenaRisk Scripting Language

Section 6.3 describes why a scripting facility needed to be added to the AgenaRisk toolset [8]. This section describes this scripting language in detail.

Scripts consist of text files where each line of text begins a new command. Empty lines are ignored. All other lines begin with a command keyword followed by a set of command parameters.

Keywords, command parameters and syntactic punctuation are all separated by space characters. Parameters which include spaces can be included in double quotes.

The case of keywords is not case sensitive. However they are shown in capitals in the descriptions that follow. Command parameters are shown surrounded by angle brackets: <>.

C.1 Risk Object Commands

These are commands which are used to load and manipulate Risk Objects (RO). Each RO is a Bayesian Net in its own right. However, by defining input and output nodes on ROs, they can be linked together to form large, complex networks.

C.1.1 NEW_MODEL

Creates a new model.

C.1.2 LOAD <Risk object filename>

Loads a RO into the model. When constructing a DBN, this command will often be issued several times in order to create multiple instances (usually) of the same Risk Object. It is the same as executing AgenaRisk's **File** → **Import Model** menu command.

When AgenaRisk imports a model it usually prompts the user to ask if the imported model's graph defaults should be extended to the whole model. This is undesirable when a script is being executed. To prevent this, edit the file:

C:\Documents and Settings\\AgenaRisk\minerva.properties.

and add the line:

```
uk.co.agena.minerva.askAboutGraphSettings=false
```

Example

```
LOAD "C:\Local Data\XP_models\Effort\effort_only2.cmp"
```

C.1.3 MOVE_OBJECT <Object name> <x> <y>

Move the RiskObject to the specified position. Must be in the MDI view for this to work (e.g. after an object load).

C.1.4 RENAME <Old object name> <New object name>

When a RO is loaded using the LOAD command, it has a default name as stored in the file it was loaded from. DBNs often need multiple copies of the same Risk Object, and scripts need a way to uniquely identify each copy. Typically, a piece of a script will load a RO, link it to some other ROs, and then rename it.

Example

```
LOAD "C:\Local Data\XP_models\Effort\iteration.cmp"  
RENAME Iteration Iteration1
```

C.1.5 LINK <Source object> <Source node> <Destination object> <Destination node>

Links the output node of a source object to the corresponding input node in a destination object. This command links Risk Objects together to form larger nets.

Linking an output with an input node effectively identifies the two nodes as being the same node. The marginal probability distribution of the two will be identical after evidence has been propagated.

Example

```
LINK "Initial Velocity Guess" "Effective engineering factor" Iteration "proj vel  
pre"
```

C.1.6 UNLINK <Source object> <Source node> <Destination object> <Destination node>

Deletes the link between the output node of a source object and the corresponding input node in a destination object. This command removes links between Risk Objects.

Example

```
UNLINK "Initial Velocity Guess" "Effective engineering factor" Iteration "proj vel  
pre"
```

C.1.7 INCLUDE <filename>

Runs the commands in the specified file (from the “current” directory”).

C.1.8 MACRO DEFINE|USE <macro_name> <macro contents>

Create or use a simple, un-paramaterised, macro.

C.1.9 PROPAGATE ALL

Propagate all ROs.

C.2 Node Commands

Node commands act on individual nodes rather than Risk Objects.

C.2.1 SET <Risk object name> <Node name> <Value>

Sets evidence on a node. This is used to set an observed value for a node.

Example

```
SET Iteration "Accuracy Limit" 0.9
```

C.2.2 CLEAR <Risk object name> <Node name>

Clears evidence on a node.

Example

```
CLEAR Iteration "Accuracy Limit"
```

C.2.3 SET <Risk object name> <Node name> WITH <\$Var> FROM value1 value2 ...

Sets evidence on a node from within a FOR loop. This allows the for loop variable to select a value from an array. The array index starts at one. Values must be included for all possible array indexes from one to the maximum value of the loop control variable.

Example

```
FOR $i = 2 to 4  
  SET Diet y1$i WITH $i FROM 62 60 63 59  
ENDFOR
```

C.2.4 CREATE_NODE <Risk object name> <Node name> <type> <x> <y> [SIMULATION] [INPUT] [OUTPUT]

Creates a new node with the given node name in the given risk object. The location in X, Y coordinates and the node type must both be specified. Optionally, the node can be specified to be a simulation node, and an input or output node.

The node type must be one of AgenaRisk's fundamental node classes. These are:

- BooleanEN
- ContinuousEN
- ContinuousIntervalEN
- DiscreteRealEN
- IntegerIntervalEN
- LabelledEN
- NumericalEN
- RankedEN

These node types are case sensitive.

Example

```
CREATE_NODE ContinuousIntervalEN 300 100 INPUT
```

C.2.5 STATES <Risk object name> <Node name> state1 [state2 ...]

Sets the valid states on a node. Multiple states can be included. For interval nodes, interval boundaries must be separated by a tilde (“~”). The keywords –Infinity and Infinity are both recognised. Multiple ranges are allowed. A Series of ranges can be defined by adding an interval after a colon.

Example

```
STATES Iteration “proj vel pre” 0~5 5~45:1 45~Infinity
```

C.2.6 EXPR <Risk object name> <Node name> Distribution parameter1 [parameter2 ...]

Sets the expression on a node. The distribution can be any of the standard distributions allowed in AgenaRisk: arithmetic, uniform, normal etc.

Example

```
EXPR Iteration "User Story Accuracy POST" Arithmetic  
max(0,(min(Accuracy_Limit,User_Story_Accuracy_PRE+Process_Improvement)))
```

C.2.7 ADD_LINK <Risk object> <Source node> <Destination node>

Adds a link between two nodes. This identifies a causal relationship between the two nodes and causes the NPT of the destination node to change dimension.

Example

```
ADD_LINK Iteration1 "proj vel pre" "Total Effort"
```

C.2.8 COPY <Risk object> <Source node> <Destination node>

Creates a copy of a node in a risk object. It's parents, states, and NPT will all be identical.

Example

```
COPY Diet thetaA thetaB
```

C.2.9 DEL_LINK <Risk object> <Source node> <Destination node>

Deletes a link between two nodes. This is often needed when copies of nodes are being created and the links to the old parent nodes have to be deleted.

Example

```
DEL_LINK Diet thetaA y21
```

C.2.10 CONSTANT ADD|SET <Risk object>
<Source node> <Constant name > <Constant value>

Creates or modifies a constant (expression variable) in a node.

C.3 Scenario Commands

C.3.1 ADD_SCENARIO <Scenario name>

Adds a new scenario.

C.3.2 RENAME_SCENARIO <Old name> <New name>

Renames a new scenario.

C.3.3 DEFAULT_SCENARIO < Scenario name>

Sets the default scenario for use with the **CLEAR** and **SET** commands.

C.4 Output Commands

The script language provides the ability to print values to a file. This is useful when collecting values from a model after it has been run. It also allows graphs to be displayed automatically.

C.4.1 OPEN_LOG <log file>

Opens a log file for printing.

Example

```
OPEN_LOG "C:\Local Data\XP_models\Effort\log.txt"
```

C.4.2 CLOSE_LOG

Close the current log file.

C.4.3 PRINT <Risk object> <Node> MEAN | MEDIAN | SD

Prints a value to the current log file.

Example

```
PRINT Iteration1 "proj vel post" mean
```

C.4.4 GRAPH_DEFAULT <what> <value>

Sets default graph parameters. The only currently allowed values for <what> are:

Continuous X Axis

Treat min max x as percentile

min X

max X

Example

```
graph_default "Continuous X Axis" "True"  
graph_default "Treat min max x as percentile" "True"  
# NOTE - numbers below MUST include a decimal point  
graph_default "min X" 1.0  
graph_default "max X" 99.0
```

C.4.5 GRAPH <Risk Object> <Node>

Opens a graph for the specified node in the risk graph panel.

Example

```
GRAPH Iteration1 "proj vel pre"
```

C.5 Miscellaneous Commands

C.5.1 #

The # command introduces a comment. All remaining text following the # symbol, on the same line, is ignored.

Example

```
# A comment
```

C.5.2 DEFAULT NET <Risk object name>

Set the default Risk Object that further commands should act on. Once this command has been issued, the first Risk Object name in all commands that follow takes the default value. "DEFAULT NET none" removes the default network name.

Example

```
DEFAULT NET "Initial Velocity Guess"  
# The <Risk object> parameter can be omitted from commands that follow  
GRAPH "proj vel pre"
```

C.5.3 DEFAULT NODE <Node name>

Set the default node that further commands should act on. Once this command has been issued, the first node name in all commands that follow takes the default value. "DEFAULT NODE none" removes the default node name.

Example

```
DEFAULT NET "Initial Velocity Guess"  
DEFAULT NODE "Process Improvement"  
# Both the <Risk object> and <Node name> parameters are omitted below  
EXPR Uniform -2 2
```

C.5.4 LABEL <x> <y> <width> <height> <colour>

Creates a text label in the current view (either MDI view or BN view). The transparency is always set to 60% and the newly created object is always set to be behind all existing nodes and labels.

Example

```
LABEL 90 0 420 380 bfffbb "Process nodes"
```

C.5.5 FOR \$<var> = <low> TO <high> ... ENDFOR

Executes all commands between the FOR and ENDFOR commands. Each command is executed high-low+1 times. On each occasion, the loop variable \$<var> takes on one of the values from low to high inclusive.

There is no proper expression evaluator built into the scripting language. \$<var> simply defines a string substitution mechanism. Wherever \$<var> appears in any of the command between FOR and ENDFOR it is replaced with the current value of the loop variable. One exception is the special symbol \$<var>- which refers to one before the current loop variable.

Example

```
FOR $i = 2 to 6
  LOAD "C:\Local Data\XP_models\Effort\simple_effort.cmp"
  SET Iteration "Process Improvement" 0.2
  # Link Iteration1 to Iteration2 etc.
  LINK Iteration$i- "proj vel post" Iteration "proj vel pre"
  RENAME Iteration Iteration$i
ENDFOR
```

C.5.6 EXIT

Exits the script immediately. All further commands are ignored.

C.5.7 VAR <variable_name> <operator> <operand>

Creates or modifies a numeric script variable. The following operands are defined.

= Create a variable or assign a value. e.g. var x_posn = 20

+= Adds an amount to a variable. e.g. var x_posn += 10

-= Subtracts an amount to a variable. e.g. var x_posn -= 10

*= Multiplies a variable. e.g. var x_posn *= 10

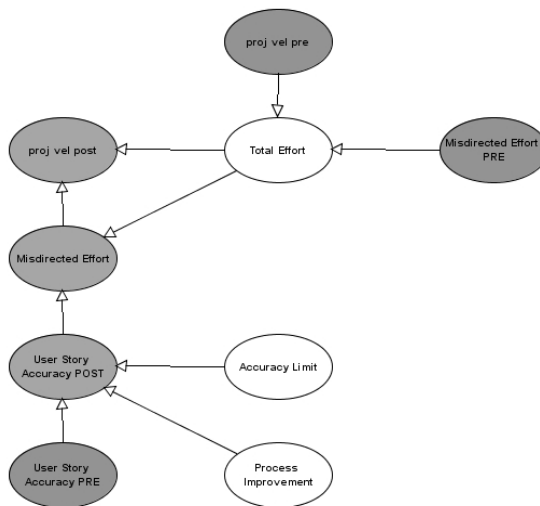
/= Divides a variable. e.g. var x_posn /= 10

Variables are held as Java double values. Most commands which accept an integer or double value will also accept a variable name.

C.6 - Example Scripts

C.6.1 Example 1 – Creating a small net.

This example creates the following net.



```
rename "New Risk Object" Iteration
default net Iteration
  default node "proj vel pre"
    create_node ContinuousIntervaleN 300 100 input
    states 0~500
    expr Arithmetic 0
    default node none

  default node "Misdirected Effort PRE"
```

```

        create_node ContinuousIntervalEN 500 200 input
        states 0~500
        expr Arithmetic 0
    default node none

    default node "Total Effort"
        create_node ContinuousIntervalEN 300 200 simulation
        states 0~500
    default node none

    add_link "proj vel pre" "Total Effort"
    add_link "Misdirected Effort PRE" "Total Effort"
    expr "Total Effort" Arithmetic proj_vel_pre+Misdirected_Effort_PRE

    default node "User Story Accuracy PRE"
        create_node ContinuousIntervalEN 100 500 input
        states 0~1
        expr Uniform 0 1
    default node none

    default node "Process Improvement"
        create_node ContinuousIntervalEN 300 500 simulation
        states -2~2
        expr Uniform -2 2
    default node none

    default node "Accuracy Limit"
        create_node ContinuousIntervalEN 300 400 simulation
        states -2~2
        expr Uniform -2 2
    default node none

    default node "User Story Accuracy POST"
        create_node ContinuousIntervalEN 100 400 simulation output
        states 0~1
    default node none

    add_link "User Story Accuracy PRE" "User Story Accuracy POST"
    add_link "Process Improvement" "User Story Accuracy POST"
    add_link "Accuracy Limit" "User Story Accuracy POST"
    expr "User Story Accuracy POST" Arithmetic max(0, (min(Accuracy_Limit,
        User_Story_Accuracy_PRE+Process_Improvement)))

    default node "Misdirected Effort"
        create_node ContinuousIntervalEN 100 300 simulation output
        states 0~500
    default node none

    add_link "Total Effort" "Misdirected Effort"
    add_link "User Story Accuracy POST" "Misdirected Effort"
    expr "Misdirected Effort" Normal Total_Effort*(1-User_Story_Accuracy_POST)
        Total_Effort*(1-User_Story_Accuracy_POST)/10

    default node "proj vel post"
        create_node ContinuousIntervalEN 100 200 simulation output
        states 0~500
    default node none

    add_link "Total Effort" "proj vel post"
    add_link "Misdirected Effort" "proj vel post"
    expr "proj vel post" Arithmetic max(0, Total_Effort-Misdirected_Effort)

    default net none

```

C.6.2 Example 2 – linking several iterations of a net together

```

# Set initial parameters
default net "Initial Velocity Guess"
    set "Number team members" 5
    set Productivity Medium
    set "Load Factor" 2
default net none

# Setup Iteration1

```



```

load "C:\Local Data\XP_models\Effort\simple_effort.cmp"
default net Iteration
  set "User Story Accuracy POST" 0.5
default net none
link "Initial Velocity Guess" "Effective engineering factor" Iteration "proj vel
pre"
link "Initial Velocity Guess" "Zero Defects" Iteration "Misdirected Effort PRE"
rename Iteration Iteration1

# Setup Iteration2
for $i = 2 to 6
  load "C:\Local Data\XP_models\Effort\simple_effort.cmp"
  set Iteration "Process Improvement" 0.2
  set Iteration "Accuracy Limit" 0.9
  link Iteration$i- "proj vel post" Iteration "proj vel pre"
  link Iteration$i- "Misdirected Effort" Iteration "Misdirected Effort PRE"
  link Iteration$i- "User Story Accuracy POST" Iteration "User Story Accuracy PRE"
  rename Iteration Iteration$i
endfor

propagate all

open_log "C:\Local Data\XP_models\Effort\log.txt"
graph Iteration1 "proj vel pre"
for $i = 1 to 6
  default net Iteration$i
  graph "Total Effort"
  graph "Misdirected Effort"
  graph "proj vel post"
  print "proj vel post" mean
  default net node
endfor
close_log

```

Appendix D - Noisy Or

Appendix H shows how simple truth tables can be used as CPDs to combine multiple Boolean causal factors into a single Boolean outcome. We now consider an extension of this where one of several multiple causes will *probably* cause an outcome, but where it is sometimes inhibited. We can model this as a set of possible boolean causes, A_i , each of which is a parent of a possible boolean inhibitor I_i . The inhibitors are then the parents of an outcome, B , where B has the logical OR truth table as its NPT.

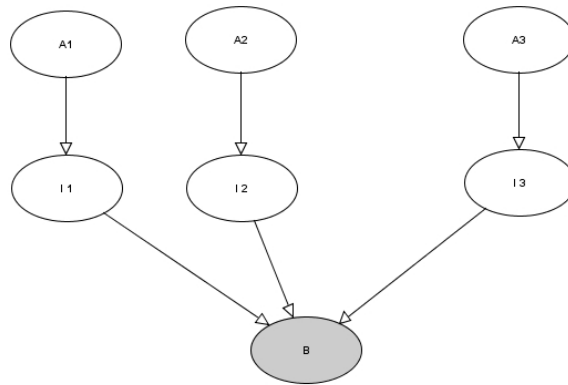


Figure C-1 The Noisy OR model

B will definitely be false if all the A_i are false. It is only when some A_i are true that B can be true. Each of the inhibitors has the NPT shown in Table D-1.

$P(I A)$	$A_i = \text{TRUE}$	$A_i = \text{false}$
$I_i = \text{TRUE}$	$1 - q_i = \bar{q}_i$	0
$I_i = \text{false}$	q_i	1

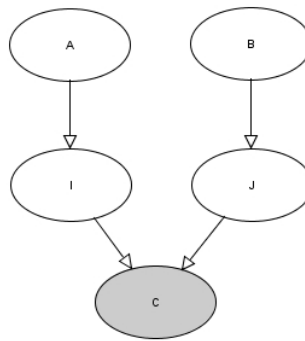
Table D-1 Node probability table for inhibitors

i.e. q_i , is simply the probability of the inhibitor working. B can only be false if all of the inhibitors of true A_i are working. As the inhibitors are all independent, the probability of them all working, is simply the product of each one individually working. If the probability of B being false is the product of the probabilities of all the inhibitors working, then the probability of B being true is simply the complement of this, as shown in Equation D-1.

$$P(B = true | A_1 \dots A_n) = 1 - \prod_{i \in \{A_i = true\}} q_i \quad \text{Equation D-1}$$

We can therefore remove the inhibitors from our BN and create an equivalent BN where the A_i are the direct parents of B and B has the NPT defined by Equation D-1.

We examine the simplest possible case, where we have two potential causes, A and B , of an outcome C . There are inhibitors between the causes and the outcome. We will label the inhibitors I and J . The BN corresponding to this situation is shown below.



For each Boolean variable, we list its states by a lower case letter and the same letter with a bar. Thus A , has states $\{a, \bar{a}\}$, B has states $\{b, \bar{b}\}$ and so forth.

P_I^A			P_C^{IJ} - Logical OR				
	a	\bar{a}		i	\bar{i}	j	\bar{j}
i	q_a	0	c	1	1	1	0
\bar{i}	$\bar{q}_a = 1 - q_a$	1	\bar{c}	0	0	0	1
P_J^B							
	b	\bar{b}					
j	q_b	0					
\bar{j}	$\bar{q}_b = 1 - q_b$	1					

It is possible to represent this as a smaller BN involving only A , B and C . To see this, we can explicitly calculate P_C^{AB} as follows:

$$P_C^{AB} = \frac{P_{ABC}}{P_{AB}} = \frac{\sum_{IJ} P_C^{IJ} P_I^A P_J^B P_A P_B}{\sum_{IJC} P_C^{IJ} P_I^A P_J^B P_A P_B} = \frac{\sum_{IJ} P_C^{IJ} P_I^A P_J^B}{\sum_{IJC} P_C^{IJ} P_I^A P_J^B}$$

We now evaluate this expression for each of the four possible combinations that leads to $C = \text{true}$.

$$\begin{aligned} P_c^{ab} &= \frac{P_c^{ij} P_i^a P_j^b + P_c^{i\bar{j}} P_i^a P_{\bar{j}}^b + P_c^{\bar{i}j} P_{\bar{i}}^a P_j^b}{P_c^{ij} P_i^a P_j^b + P_c^{i\bar{j}} P_i^a P_{\bar{j}}^b + P_c^{\bar{i}j} P_{\bar{i}}^a P_j^b + P_c^{\bar{i}\bar{j}} P_{\bar{i}}^a P_{\bar{j}}^b} = \frac{P_i^a P_j^b + P_i^a P_{\bar{j}}^b + P_{\bar{i}}^a P_j^b}{P_i^a P_j^b + P_i^a P_{\bar{j}}^b + P_{\bar{i}}^a P_j^b + P_{\bar{i}}^a P_{\bar{j}}^b} \\ &= \frac{q_a q_b + q_a \bar{q}_b + \bar{q}_a q_b}{q_a q_b + q_a \bar{q}_b + \bar{q}_a q_b + \bar{q}_a \bar{q}_b} = \frac{(1 - \bar{q}_a)(1 - \bar{q}_b) + (1 - \bar{q}_a)\bar{q}_b + \bar{q}_a(1 - \bar{q}_b)}{(1 - \bar{q}_a)(1 - \bar{q}_b) + (1 - \bar{q}_a)\bar{q}_b + \bar{q}_a(1 - \bar{q}_b) + \bar{q}_a \bar{q}_b} \\ &= \frac{1 - \bar{q}_b \bar{q}_a}{1} = 1 - \bar{q}_b \bar{q}_a \end{aligned}$$

$$\begin{aligned} P_c^{\bar{a}b} &= \frac{P_c^{ij} P_i^{\bar{a}} P_j^b + P_c^{i\bar{j}} P_i^{\bar{a}} P_{\bar{j}}^b + P_c^{\bar{i}j} P_{\bar{i}}^{\bar{a}} P_j^b}{P_c^{ij} P_i^{\bar{a}} P_j^b + P_c^{i\bar{j}} P_i^{\bar{a}} P_{\bar{j}}^b + P_c^{\bar{i}j} P_{\bar{i}}^{\bar{a}} P_j^b + P_c^{\bar{i}\bar{j}} P_{\bar{i}}^{\bar{a}} P_{\bar{j}}^b} \\ &= \frac{P_i^{\bar{a}} P_j^b + P_i^{\bar{a}} P_{\bar{j}}^b + P_{\bar{i}}^{\bar{a}} P_j^b}{P_i^{\bar{a}} P_j^b + P_i^{\bar{a}} P_{\bar{j}}^b + P_{\bar{i}}^{\bar{a}} P_j^b + P_{\bar{i}}^{\bar{a}} P_{\bar{j}}^b} = \frac{q_b}{q_b + \bar{q}_b} = 1 - \bar{q}_b \end{aligned}$$

By symmetry: $P_c^{\bar{a}\bar{b}} = 1 - \bar{q}_a$.

$$P_c^{\bar{a}\bar{b}} = \frac{P_c^{ij} P_i^{\bar{a}} P_{\bar{j}}^{\bar{b}} + P_c^{i\bar{j}} P_i^{\bar{a}} P_j^{\bar{b}} + P_c^{\bar{i}j} P_{\bar{i}}^{\bar{a}} P_j^{\bar{b}}}{P_c^{ij} P_i^{\bar{a}} P_{\bar{j}}^{\bar{b}} + P_c^{i\bar{j}} P_i^{\bar{a}} P_j^{\bar{b}} + P_c^{\bar{i}j} P_{\bar{i}}^{\bar{a}} P_j^{\bar{b}} + P_c^{\bar{i}\bar{j}} P_{\bar{i}}^{\bar{a}} P_{\bar{j}}^{\bar{b}}} = 0$$

Appendix E – Formal Description of Data Import and Mapping

In this section I present a formal description of the data query, import and mapping facilities developed in order to integrate agile BN models with appropriate quantitative data sources.

A **Data Source** consists of a set of **Connection Parameters** $C = \{c_i\}$, a set of **Query Classes** $Q = \{\kappa_j\}$ and a set of **Mapping Functions** $M = \{m_k\}$:

$$D_{CQM} = (C, Q, M).$$

A query class is a data source query string, s , expressed in the natural language of that data source (e.g. SQL) and a set of zero or more **Query Parameters**, $\{p_i\}$:

$$\kappa = (s, \{p_i\})$$

The term *query class* should not be confused with an OO class. It is intended to represent a class of queries, or a template for a query. A query class is not executable in the sense that it cannot be used to query a data source. Only instances of a query class can be used to query data sources.

Query parameters are not the same as SQL prepared statement or callable statement parameters. Prepared statement parameters are usually limited in scope. Query parameters by contrast may parameterise any aspect of a query string.

A **Query Object**, q , is an instance of a query class where the query parameters have been defined:

$$q = (s, \{p_i = v_i\})$$

A query object is a **Fully Instantiated Query** if all its parameters have been set to explicit values such that the resulting query is an **executable query**, e . Setting the query parameters is not necessarily sufficient to make a query fully instantiated since query parameters can be set to express dependencies between queries.

The results of an executable query, e , are denoted by $R(e)$. They are assumed to be tabular in form (possibly degenerate in either the rows or columns dimension) and can be expressed using a row/column notation:

$$R(e) = \{r_{ij}\}$$

Where r_{ij} is the data element in row i , column j . We denote the n 'th row of R by $ROW_n(R) = \{r_{nj}\}$ and the n 'th column of R by $COL_n(R) = \{r_{in}\}$.

A query object is a **Dependent Query** if its parameters are instantiated in such a way that it cannot be executed unless one or more **Parent Queries** are executed in advance. We define a syntax for query parameters which express query dependencies as follows:

Syntax	Meaning
? < o > <row> column_names : <start_column_name> - <end_column_name>	The selected column <i>names</i> from the results of o .
? < o > * < n >	$COL_n(R(o))$
? < o > < n > *	$ROW_n(R(o))$
? < o > ? < n >	v_n , where $o = (s, p_n = v_n)$

As shown, dependent query parameters usually take a range of values from their parents. This allows multiple **Child Queries** to be spawned: one for each result value of the parent.

For example, suppose we have three data sources: a project plan data source which defines a “phases” table, a source code data source which defines a “modules” table, and a bugs database which defines a “defects” table.

Let e be a fully instantiated query defined as:

$$e = (\text{“SELECT * FROM phases”}, \phi),$$

where ϕ indicates no parameters, and let d be a dependent query defined by:

$d = (\text{"SELECT * FROM modules WHERE phase_name = ?"}, p_1 = \text{"? e * 2"}).$

This assumes that column 2 of the “phases” table has values which make sense when compared to the phase_name column of the “modules” table. If $R(e)$ has m rows, then “? e * 2” will select:

$$\text{COL}_2(R(e)) = \{r_{i2}\}, i = 1..m.$$

This will create m fully instantiated child queries of d :

$$c_i = (\text{"SELECT * FROM modules WHERE phase_name = ?"}, p_1 = r_{i2})$$

where each query selects a different set of modules depending on a different phase name. This is more than a simple sub-select since the child queries are taking place on a different database from the parent (dependent) query.

A dependent query may also depend on a Bayesian Network node’s value after model execution.

Syntax	Meaning
? result mean median sd	The mean, median and standard deviation of the “selected” node’s marginal probability distribution.

Parameters to a parent query can be either a fully instantiated query or another dependent query. Where a parent query creates multiple child queries (as d did by generating the c_i above), any query that is dependent on their common parent inherits its dependency from the children, not the parent. This allows query trees to be instantiated.

Continuing the previous example, let d_2 be a dependent query defined by:

$$d_2 = (\text{"SELECT COUNT(*) FROM bugs WHERE module_name = ?"},$$

$$p_1 = "? d * 3").$$

d already has m children c_i . The above expression will therefore select:

$$\text{COL}_3(R(c_i))$$

If each $R(c_i)$ has k rows, then d_2 will spawn $k * m$ child queries:

$$q_{ij} = ("SELECT COUNT(*) FROM bugs WHERE module_name = ?", \\ p_1 = [\text{COL}_3[R(c_i)]]).$$

Mapping Functions define the mappings between the data representations in one data source and those in another. Let D_{CQM} be a data source, and let Σ be the set of all fully instantiated queries descended from the set of query classes, Q . Let

$$n \in \left\{ \bigcup_e \bigcup_i \text{COL}_i(R(e)), \forall e \in \Sigma \right\}$$

i.e. n is one of the columns that can be returned by a query in D_{CQM} . Let $\text{DOM}(n)$ be the set of all possible values in column n . Let E be a second data source with a query class with parameter p . Let $\text{DOM}(p)$ be the set of valid fully instantiated values of p . We define a mapping function

$$M(n, p) : \text{DOM}(n) \rightarrow \text{DOM}(p).$$

As p may represent the same column in multiple queries, the same mapping function may be used by multiple queries.

In the version of this functionality implemented in AgenaRisk, the column n was identified by its name. This mechanism relies on the uniqueness of $\text{DOM}(n)$ in D . If D contains multiple columns with the same name as n but with distinct domains then it is possible that a single Mapping Function will be impossible to construct. Similar mapping functions were created to map query results to BN node states.

Appendix F – Model Scripts

This appendix contains the scripts used to create the models described in Chapter 7. We begin with the script that creates the main timeslice.

```
new_model
rename "New Risk Object" Iteration

graph_default "Continuous X Axis" "True"
graph_default "Treat min max x as percentile" "True"
# NOTE - numbers below MUST include a decimal point
graph_default "min X" 0.1
graph_default "max X" 99.9

default net Iteration
include common_states.txt

default node "Effectiveness Limit PRE"
create_node ContinuousIntervalEN 100 100 input
macro use effectiveness_limit_states
expr Normal 0.8 0.1
default node none

default node "Process Improvement PRE"
create_node ContinuousIntervalEN 100 200 input
macro use process_improvement_states
expr Normal 0.2 0.1
default node none

default node "Process Effectiveness PRE"
create_node ContinuousIntervalEN 250 300 input
macro use process_effectiveness_states
expr Normal 0.3 0.1
default node none

default node "Effectiveness Limit POST"
create_node ContinuousIntervalEN 250 100 output
macro use effectiveness_limit_states
add_link "Effectiveness Limit PRE"
expr Arithmetic Effectiveness_Limit_PRE
default node none

default node "Process Improvement POST"
create_node ContinuousIntervalEN 250 200 output
macro use process_improvement_states
add_link "Process Improvement PRE"
expr Arithmetic Process_Improvement_PRE
default node none

default node "Process Effectiveness POST"
create_node ContinuousIntervalEN 400 200 output
macro use process_effectiveness_states
add_link "Process Effectiveness PRE"
add_link "Process Improvement POST"
add_link "Effectiveness Limit POST"
expr Arithmetic min(1,Process_Effectiveness_PRE+Process_Improvement_POST*(Effectiveness_Limit_POST-Process_Effectiveness_PRE))
default node none

default node "Iteration Effort"
create_node ContinuousIntervalEN 250 400
states 0~150:1
expr Uniform 0 150
default node none

default node "Productive Effort"
create_node ContinuousIntervalEN 400 400
states -100~300:5
add_link "Iteration Effort"
add_link "Process Effectiveness POST"
expr Arithmetic Iteration_Effort*max(-1,Process_Effectiveness_POST)
default node none

default node "Bias PRE"
create_node ContinuousIntervalEN 100 500 input
macro use bias_states
expr "Log Normal" 0 0.3
default node none

default node "User Stories Completed PRE"
create_node ContinuousIntervalEN 100 600 input
macro use user_stories_completed_states
expr TNormal 0 0.1 0 1
default node none

default node "Bias POST"
create_node ContinuousIntervalEN 250 500 output
macro use bias_states
add_link "Bias PRE"
expr Arithmetic Bias_PRE
default node none

default node "Observed Velocity"
create_node ContinuousIntervalEN 400 500
states -100~0:10 0~100:5 100~300:10 300~1000:100
add_link "Bias POST"
add_link "Productive Effort"
expr Arithmetic Bias_POST*Productive_Effort
default node none

default net none
```

The above script relies on some common state definitions shared between different nodes. These are defined in a separate script called `common_states.txt`. This script is shown below.

```
macro define effectiveness_limit_states states 0~1:0.1
macro define process_improvement_states states -1~1:0.1
macro define process_effectiveness_states states -1~1:0.05
macro define bias_states states 0~5:0.2 5~10
macro define user_stories_completed_states states 0~20:2 20~400:20 400~1000:100
```

Another script is used to read in the single timeslices and connect them together.

```

# Setup Iteration 1 through 6

new_model

graph_default "Continuous X Axis" "True"
graph_default "Treat min max x as percentile" "True"
# NOTE - numbers below MUST include a decimal point
graph_default "min X" 1.0
graph_default "max X" 99.0

# Set up the "Prior" iteration
load "C:\Local Data\XP_models\Effort\Motorola\initial_iteration.cmp"
set Iteration0 "User Stories Completed POST" 0

# Load iterations 1 and 8
var x_posn = 300
var y_posn = 100
for $i = 1 to 8
  load "C:\Local Data\XP_models\Effort\Motorola\simple_effort.cmp"
  move_object Iteration x_posn y_posn
  var x_posn += 200
  var y_posn += 100
  link Iteration$i- "Process Effectiveness POST" Iteration "Process Effectiveness PRE"
  link Iteration$i- "Effectiveness Limit POST" Iteration "Effectiveness Limit PRE"
  link Iteration$i- "Process Improvement POST" Iteration "Process Improvement PRE"
  link Iteration$i- "User Stories Completed POST" Iteration "User Stories Completed PRE"
  link Iteration$i- "Bias POST" Iteration "Bias PRE"
  set Iteration "Iteration Effort" with $i from 45 45 90 96 84 70 72 40
  rename Iteration Iteration$i
endfor

```

The initial conditions for the model are created by another script which creates “initial_iteration.cmp” model above. This script is shown below.

```

new_model
rename "New Risk Object" "Iteration0"
include common_states.txt

default net "Iteration0"

default node "Effectiveness Limit POST"
  create_node ContinuousIntervalEN 150 0 output
  macro use effectiveness_limit_states
  expr Normal 0.8 0.1
default node none

default node "Process Improvement POST"
  create_node ContinuousIntervalEN 150 100 output
  macro use process_improvement_states
  expr Normal 0.2 0.1
default node none

default node "Process Effectiveness POST"
  create_node ContinuousIntervalEN 150 200 output
  macro use process_effectiveness_states
  expr Normal 0.3 0.1
default node none

default node "Bias POST"
  create_node ContinuousIntervalEN 150 300 output
  macro use bias_states
  expr "Log Normal" 0 0.3
default node none

propagate all

default net none

```

Appendix G - Software Complexity Revisited: An Approach for Use in Causal Models

The notion of “complexity” has played an important role both in the traditional software process models described in chapter 2 and in the BN causal models described in chapter 4. In this appendix, we give a more detailed description of software complexity.

G.1 The Problem

The Philips model in section 4.4 used KLOC instead of Function Points (FPs) to measure problem size. FPs already include a measure of problem complexity, so by moving to KLOC, this aspect of the problem description was being lost. The challenge, therefore, was to produce a new technical complexity measure (which would effectively act as an 'adjustment' factor to the estimated KLOC).

This technical complexity measure had to be based on information that was readily available, and easily input, by project managers right at the start of any new code development. This rules out the many design-level and code-level complexity metrics that are catalogued in [61] as the primary observed values. Similarly, it ruled out measures of software complexity based on entropy definitions of its CVS repository [81][80], as well as data complexity metrics [143][220].

G.2 Factors excluded from Technical Complexity

Williams [216], for example, subdivides complexity into structural complexity and *uncertainty*. Xia and Lee [221] use this same split as the basis for one of the dimensions of their Information Systems Development Project (ISDP) complexity. We have no need to do this. With the causal modelling Bayesian approach uncertainty is modelled everywhere; every node in the model represents an uncertain value. Until a direct observation is made about the value of any node, its value is given by a probability distribution that quantifies our uncertainty about it.

As in Baccarini [17], we make a clear distinction between technical and management complexity. This is also the second dimension of Xia and Lee's [221] ISDP complexity model. The following management complexity factors are already

incorporated in the Philips model (as part of the various subcomponents relating to processes), and are therefore excluded from our approach to technical complexity:

- Project requirements stability
- Cost and schedule constraints
- Project infrastructure complexity (including Geographical team dispersal).
- Development team and managerial team experience.
- Process Maturity

In addition to the above 'process-type' management complexity factors, there are two key *product* factors, namely *reliability* and *novelty*, that we explicitly exclude from our definition of technical complexity because they are also modelled elsewhere in the model. However, the rationale for their exclusion needs to be explained.

Reliability is normally a requirement for the delivered product. It can be defined in terms of such measures as: number of defects discovered post-release, seriousness of defects, mean time between failures etc. [61]. Reliability is determined by software complexity and process maturity and is therefore a consequence, rather than a cause of, complexity.

Novelty: All computer programs are novel to some extent. It is this novelty that makes their development such a skilled task and clearly novelty has an impact on the potential for defect insertion. However, it is the way novelty is *managed* that determines this. Hence in our model novelty and its management are bound up in the requirements and specification process component of the model. Highly novel requirements that are poorly managed will result in low probability of avoiding defects.

A high degree of novelty, either technological or problematic, does not imply a high degree of complexity. We are just as likely to overestimate as to underestimate the complexity of the solution. This suggests that novelty and complexity should not be highly correlated. Tatikonda and Rosenthal [202] have indeed shown that novelty, at least in the technological sense, and technical complexity are largely independent.

It seems justifiable therefore to conclude that novelty is *not* a valid component of technical complexity. This would seem to be confirmed by regression based models, such as COCOMO II [47]. For example, the COCOMO II formula for effort estimation includes Product Complexity (CPLX) as an effort multiplier, whereas

Precedentedness (PREC) appears as a scaling factor in the exponent. This suggests that complexity and novelty are different in kind and should therefore be treated separately.

G.3 Technical Complexity Factors

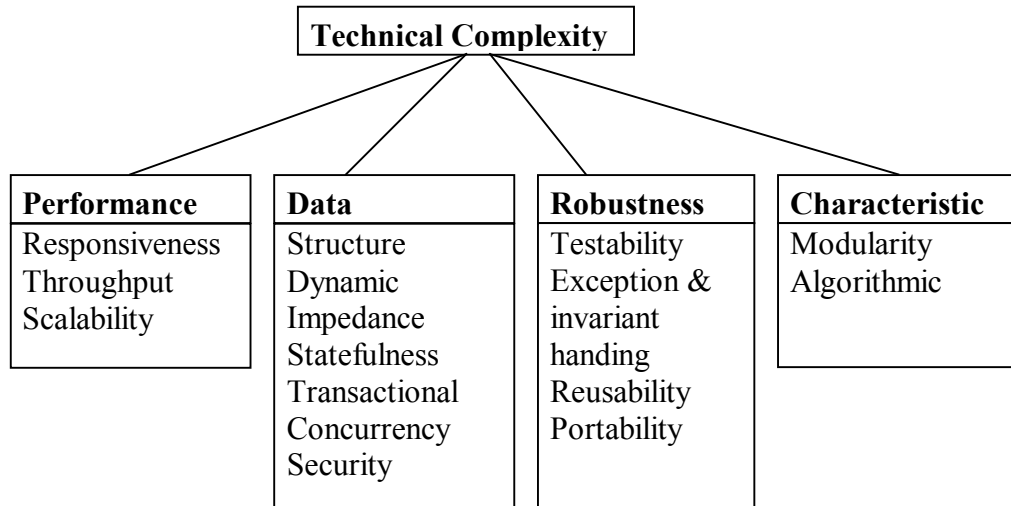


Figure G-1. Technical complexity broken down into four *groups*, each consisting of various complexity *factors*.

The main component of this approach is to identify a set of *complexity factors*, collected into a set of *complexity groups*. The aim in grouping the factors are to:

1. Create a stable set of technology factors, and
2. Create a smaller, more manageable set of factors.

The four complexity groups and their associated factors are summarised in Figure G-1. Complexity groups should possess the following qualities.

- a) As technology evolves the set of complexity groups should remain relatively static. Note, however, that the factors that form any given complexity group will almost certainly change over time.
- b) There should be sufficiently few of them that they can be practically modelled.
- c) They should be sufficiently comprehensive that any future technical complexity factors can be included in one group.

- d) The groups should be independent of one another. i.e. it should be possible for complexity in one group to vary without necessarily varying complexity in any other group. A corollary of this is that each technical complexity factor should belong to only one group. It may not be possible to meet the aim of independence between complexity groups in full; instead the aim is that any correlations between them should be small enough not to distort the assessed complexity in practice.

A description of the complexity factors in each of the complexity groups is given in the following sections.

G.3.1 Performance Complexity

This group is characterised by technical complexity factors related to speed; where operations must be performed at a given rate or within a given time interval. In each case a higher value for the factor implies higher complexity.

Responsiveness

This is the requirement for a task to respond within a given period of time.

Throughput

This is the need to perform a certain number of operations in a given time interval.

Scalability

This is the requirement that system performance must scale well, without excessive cost or resource usage.

G.3.2 Data Complexity

The Data Complexity group includes all factors that are determined by the size, integrity or persistence of data. In each case a higher value for the factor implies higher complexity. Note that database complexity is not included as a separate factor within this group. Many of the factors listed apply equally well to both databases and to a program's internal memory.

Data Structure Complexity

This is the extent to which the problem mandates the maintenance of large amounts of interdependent data.

Transactional

This is the extent to which the system must maintain transactional integrity across groups of related operations.

Dynamic Data

This is the extent to which the system is required to create and destroy data at runtime. Note that dynamic data problems do not disappear simply by using garbage collection [100].

Impedance Mismatch

An impedance mismatch occurs where the system is required to interface two technologies whose natural data representations are difficult to translate. E.g. where an OO language must save object states in a relational database. This factor is the extent to which the problem creates such mismatches.

Statefulness

This factor measures the extent to which the system must maintain a state which is dependent on the history of the system. A system that must maintain an internal state is more complex than one which is transient. If the state must be maintained across system instantiations then further complexities arise due to the need to maintain consistency whilst copying the state to non-volatile storage.

Concurrency

This is the extent to which the system must maintain multiple, concurrent states which must also share a common context. Guaranteeing the integrity of this shared context is dependent on the use of appropriate synchronisation techniques [177].

Security

This is the extent to which data must remain confidential.

G.3.3 Robustness Complexity

The complexity factors in this group all relate to requirements for design techniques or constraints to make the software robust to failure or change. Where these attributes are required this generally implies higher technical complexity.

Testability

This is the extent to which the system must be testable. The requirements may specify the thoroughness of the testing, the extent to which it must be automated, the level of instrumentation and the testing environment.

Exception Handling

The extent to which the system must recover from errors. Successful exception handling depends on the extent to which it makes sense for a system to try to recover from a failed operation and the language and/or operating system facilities available to assist in this.

Reusability/Portability

This is the extent to which the requirements specify the need for the software to be reusable once completed.

It takes more effort to create reusable components. Interfaces and algorithms must generally be more complex in order to handle a wider variety of situations than might be necessary within a single application. There is also some evidence to suggest that code reuse via class inheritance can even make maintainability more, rather than less, difficult [36][166].

As Mooney [138] points out, portability is a specialised form of reuse.

G.3.4 Characteristic Complexity

This group includes complexity factors that are characteristic of the problem being solved.

Specialist Algorithms

This is the extent to which the software is dependent on specialist algorithms. Examples include mathematical, compression or audio visual algorithms.

Modular Decomposition Uniqueness

This is the extent to which a problem admits a unique decomposition. Some problems have an "obvious" modular decomposition, others are less tractable.

Appendix H - Automated Data Import, Example XML Configuration

```

<config>
  <datasource name="Bugzilla">
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost/agna</url>
    <type>Database</type>
    <query name="versions" update="false">
      <querystring>select value,min(creation_ts) from versions,bugs where value=version AND
        value LIKE '3.12%' group by value order by 2</querystring>
      <parameterCount>0</parameterCount>
      <description></description>
      <queryinstance name="all_versions">
        </queryinstance>
    </query>
    <query name="bugs_for_version" update="false">
      <querystring>SELECT count(*) FROM bugs WHERE version LIKE '?</querystring>
      <parameterCount>1</parameterCount>
      <description></description>
      <queryinstance name="bug_count_for_version">
        <parameter>? all_versions * 0</parameter>
        </queryinstance>
    </query>
    <query name="closed_bugs_for_version" update="false">
      <querystring>SELECT count(*) FROM bugs WHERE version LIKE '?' AND
        bug_status='CLOSED'</querystring>
      <parameterCount>1</parameterCount>
      <description></description>
    </query>
    [...other queries not shown...]
  </datasource>
  <datasource name="Philips">
    <driver>sun.jdbc.odbc.JdbcOdbcDriver</driver>
    <url>jdbc:odbc:Driver={Microsoft Excel Driver (*.xls)};DBQ=C:/Local Data
      /EclipseWorkspaces/Agna 3.15/AgnaRaw.xls;DriverID=22;READONLY=false</url>
    <type>Excel</type>
    <query name="single_project_data_template" update="false">
      <querystring>SELECT * FROM [Raw Transposed$] WHERE F1=?</querystring>
      <parameterCount>1</parameterCount>
      <description>null</description>
      <queryinstance name="philips_single_project">
        <parameter>? philips_project_names * 0</parameter>
        </queryinstance>
    </query>
    [...other queries not shown...]
  </datasource>
  <evidencemapping>
    <nodename>Relevant experience of spec & doc staff</nodename>
    <query>philips_single_project</query>
    <column>3</column>
    <row>0</row>
  </evidencemapping>
  <evidencemapping>
    <nodename>Quality of any previous documentation</nodename>
    <query>philips_single_project</query>

```

```
<column>4</column>
<row>0</row>
</evidencemapping>
<evidencemapping>
  <nodename>Regularity of spec and doc reviews</nodename>
  <query>philips_single_project</query>
  <column>5</column>
  <row>0</row>
</evidencemapping>
<evidencemapping>
  <nodename>Standard procedures followed</nodename>
  <query>philips_single_project</query>
  <column>6</column>
  <row>0</row>
</evidencemapping>
[...other evidence mappings not shown...]
</config>
```

References

- [1] Abdel-Hamid T, Madnick S, Software Project Dynamics: An Integrated Approach: Prentice Hall, 1991
- [2] Abdel-Hamid T, "The Dynamics of Software Projects Staffing: A System Dynamics Based Simulation Approach," IEEE Transactions on Software Engineering, vol. 15, no. 2, pp. 109-119, 1989
- [3] Abrahamsson P, Koskela J, Extreme Programming: A Survey of Empirical Data from a Controlled Case Study, 2004 International Symposium on Empirical Software Engineering (ISESE'04), pp. 73-82
- [4] Abrahamsson P, Warsta J, Siponen MT, Ronkainen J, New directions on agile methods: a comparative analysis Software Engineering, Proceedings 25th International Conference on Software Engineering 2003, 244-254
- [5] Agile management message board, Is FDD agile?, <http://tech.groups.yahoo.com/group/agilemanagement/message/3322>, accessed 5th Mar 2008
- [6] Ahmed A, Fraz MM, Zahid FA, Some results of experimentation with extreme programming paradigm, 7th International Multi Topic Conference, INMIC 2003. Page(s): 387- 390
- [7] Agena Ltd, Software Project Risks Models Manual, Version 01.01, 17 Nov 2004
- [8] AgenaRisk, <http://www.agena.co.uk/products/desktop.shtml>, accessed 9th Aug 2007, Bayesian Network modelling toolset
- [9] Agile Manifesto, <http://www.agilemanifesto.org/>. Accessed 22 Mar 2007.
- [10] Akiyama F, An Example of Software System Debugging, Information Processing, vol. 71, pp. 353-379, 1971
- [11] Albrecht A.J., "Measuring Application Development Productivity," Proc. Joint SHARE/GUIDE/IBM Application Development Symp., pp. 83-92, 1979
- [12] Ambler S, Survey Says: Agile Works in Practice, Dr. Dobb's Journal, Issue no. 388, Sep 2006, pp. 62-64, <http://www.ddj.com/architect/191800169?cid=Ambysoft>, accessed 5 Mar 2008
- [13] <http://www.ambysoft.com/unifiedprocess/agileUP.html>, accessed 6th Mar 2008
- [14] Antoniol G, Lokan C, Caldiera G, Fiutem R, A Function Point-Like Measure for Object-Oriented Software, Empirical Software Engineering, 4 (3): 263-287, September 1999.
- [15] Arnborg S, Corneil DG, Proskurowski A, Complexity of finding embeddings in a k-tree, SIAM Journal of Algebraic and Discrete Methods, 8(2), 277{284, 1987
- [16] Aveling B, XP Lite considered harmful? Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering, Garmisch-Partenkirchen, Germany, June 2004
- [17] Baccarini D, The concept of project complexity - a review, International Journal of Project Management, Vol. 14, No. 4, pp. 201-204, 1996.
- [18] Beck K, Extreme Programming Explained, Embrace Change, Addison-Wesley Professional; 1st edition (2000)
- [19] Beck K, Andres C, Extreme Programming Explained, Embrace Change, Addison-Wesley Professional; 2nd edition (November 16, 2004)
- [20] Beck K, Fowler M, Planning Extreme Programming, Addison-Wesley, 2001
- [21] Becker F, Sims W, Offices that Work: Balancing Cost, Flexibility, and Communication, Cornell University International Workplace Studies Program,

- October 2001. Available online at
http://iwspl.human.cornell.edu/pubs/pdf/IWS_0002.PDF.
- [22] Bibi S, Stamelos I, Software Process modeling with Bayesian belief Networks, 10th International Software Metrics Symposium Chicago, September 2004
 - [23] Boas ML, Mathematical Methods in the Physical Sciences, 2nd edition, Wiley.
 - [24] Boehm B, Software engineering economics, Englewood Cliffs, NJ, Prentice-Hall, 1981
 - [25] Boehm BW, Clark B, Horowitz E, Westland JC, Madachy RJ, Selby RW, Cost Models for Future Software Life Cycle Processes: COCOMO 2.0, Ann. Software Eng, 1995, 1, pp. 57-94
 - [26] Boyen X, Koller D, Tractable inference for complex stochastic processes, Proc. of the Conf. on Uncertainty in AI, 1998
 - [27] Breiman L, Friedman J, Olshen R, Stone C, Classification and Regression Trees, Belmont, CA, Wadsworth International, 1984.
 - [28] Briand LC, El Emam K, Surmann D, Wiczorek I, Maxwell KD, An Assessment and Comparison of Common Software Cost Estimation Modeling Techniques, ICSE 1999: 313-322
 - [29] Briand LC, Wiczorek I, Resource Estimation in Software Engineering, Encyclopedia of Software Engineering, 2nd Edition, Wiley, 2001
 - [30] Broemeling L, Broemeling A, Studies in the history of probability and statistics XLVIII The Bayesian contributions of Ernest Lhoste, Biometrika, 2003, 90, 728-731
 - [31] Brooks FP, The Mythical Man-Month: essays on software engineering, 2nd edition, Addison Wesley, 1995
 - [32] Brown WJ, Malveau RC, McCormick HW, Mowbray TJ, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley Computer publishing, 1998
 - [33] Brown WJ, McCormick HW, Thomas SW, AntiPatterns in Project Management, Wiley Computer publishing, 2000
 - [34] Canfora G, Cimitile A, Visaggio CA, "Empirical Study on the Productivity of the Pair Programming", XP 2005 Conference, June, Sheffield, UK, LNCS Springer- Verlag.
 - [35] Cao L, Mohan K, Peng Xu, Balasubramaniam R, How Extreme does Extreme Programming Have to be? Adapting XP Practices to Large-scale Projects, Proceedings of the 37th Hawaii International Conference on System Sciences, 2004
 - [36] Cartwright M, 1998. An empirical view of inheritance. Inform Soft Technol 40 (4), 795-799, <http://dec.bournemouth.ac.uk/ESERG>
 - [37] Cau A, Concas G, Melis M, Turnu I, Evaluate XP Effectiveness Using Simulation Modeling, Proceedings 6th International Conference Extreme Programming and Agile Processes in Software Engineering, XP 2005, Sheffield, UK, June 18-23, 2005
 - [38] Chen R, Sivakumar K, Khargupta H, Learning Bayesian Network Structure from Distributed Data, SIAM 2003
 - [39] Cheng, J., 1998, "PowerConstructor System", <http://www.cs.ualberta.ca/~jcheng/bnpc.htm> (accessed 17 July 2007)
 - [40] Chidamber SR, Kemerer CF, A metrics suite for object-oriented design, IEEE Transactions on Software Engineering 1994, 20(6):476-493
 - [41] Chickering DM, Geiger D, Heckerman D, Learning Bayesian Networks is NP-Hard, Microsoft Technical Report, MSR-TR-94-17, Nov 1994

- [42] Christie AM, Simulation: An Enabling Technology in Software Engineering, CROSSTALK The Journal of Defense Software Engineering, April 1999, pp. 25-30.
- [43] Chulani S, Boehm B, Steece B, Bayesian analysis of empirical software engineering cost models, IEEE Transactions on Software Engineering, Special Issue on Empirical Methods in Software Engineering, Vol. 25, No. 4, July/August 1999
- [44] CMMI for Development, Version 1.2, August 2006, Carnegie Mellon Software Engineering Institute
- [45] Coad P, Lefebvre E, De Luca J, Java Modeling in Color With UML: Enterprise Components and Process, Prentice Hall International, 1999, ISBN 0-13-011510-X
- [46] Cockburn A, Williams L, The Costs and Benefits of Pair Programming, Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2000, June 2000 Cagliari, Sardinia, Italy
- [47] COCOMO II Model Definition Manual, 5 Feb 1999, <http://sunset.usc.edu/research/COCOMOII/index.html>
- [48] Cohn M, "Agile Estimating and Planning", Prentice Hall, 2005
- [49] Coolen FP, Goldstein M, Wooff DA, Using Bayesian statistics to support testing of software systems. Proceedings of the 16th Advances in Reliability Technology Symposium, ed. J. Andrews, pp 109-121.
- [50] Coolen FPA, Goldstein M, Wooff DA, Project viability assessment for support of software testing via Bayesian graphical modelling. In Safety and Reliability. Bedford & van Gelder Lisse: Swets & Zeitlinger. 417-422.
- [51] Costagliola G, Ferrucci F, Tortora G, Vitiello G, Class Point: An Approach for the Size Estimation of Object-Oriented Systems, IEEE Transactions On Software Engineering, Vol. 31, No. 1, pp. 52-74, January 2005
- [52] Cozman F, Krotkov E, Truncated Gaussians as Tolerance Sets, Robotics Institute, Carnegie Mellon University, Technical Report CMU-RI-TR-94-35, Sep. 1994
- [53] Cusumano M, MacCormack A, Kemerer CF, Crandall B, Software Development Worldwide: The State of the Practice, IEEE Software, November/December 2003 (Vol. 20, No. 6) pp. 28-34
- [54] Dale C, Data requirements for software reliability prediction, Software Reliability: Assessment and Achievement, B. Littlewood editor, Blackwell, London, pp. 144-153, 198
- [55] Danielson DA, Vectors and tensors in engineering and physics, Addison-Wesley, 1997
- [56] DeMarco T, Lister T, Peopleware: productive projects and teams, Dorset House Publishing Co. Inc., 1987
- [57] Denton AD, Accurate Software Reliability Estimation, Master of Science Thesis, Colorado State University, Fort Collins, Colorado, Fall 1999
- [58] Deursen, A, (Ed). Proceedings of the XP Workshop on Customer Involvement in Extreme Programming, Sardinia, Italy, 2001.
- [59] Elssamadisy A, "XP On A Large Project – A Developer’s View," in Proceedings of XP/Agile Universe, Raleigh, NC, 2001
- [60] Erdogmus H, Morisio M, Torchiano M, On the Effectiveness of the Test-First Approach to Programming, IEEE Transactions On Software Engineering, Vol. 31, no. 3, march 2005

- [61] Fenton & Pfleeger, Software Metrics, A Rigorous and Practical Approach, PWS publishing 1997
- [62] Fenton NE, Krause P, Neil M, "Software Measurement: Uncertainty and Causal Modelling", IEEE Software 10(4), 116-122, 2002
- [63] Fenton NE, Marsh W, Neil M, Cates P, Forey S, Tailor T, Making Resource Decisions for Software Projects. In Proceedings of 26th International Conference on Software Engineering (ICSE 2004), (Edinburgh, United Kingdom, May 2004) IEEE Computer Society 2004, ISBN 0-7695-2163-0, 397-406
- [64] Fenton NE, Neil M, "A Critique of Software Defect Prediction Models," IEEE Transactions on Software Engineering, 25(4):675-689, September 1999
- [65] Fenton NE, Neil M, and Caballero JG, Using Ranked nodes to model qualitative judgements in Bayesian Networks, to appear IEEE TKDE, 2007
- [66] Fenton N, Neil M, Marsh W, Hearty P, Marquez D, Krause P, Mishra R, Predicting software defects in varying development lifecycles using Bayesian nets. Inf Softw Technol 2007;49(1):32-43
- [67] Fenton N, Neil M, Marsh W, Hearty P, Radlinski L, Krause P, Project Data Incorporating Qualitative Facts for Improved Software Defect Prediction, PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering
- [68] Finnie GR, Wittig GE, Desharnais JM, A Comparison of Software Effort Estimation Techniques: Using Function Points with Neural Networks, Case-Based Reasoning and Regression Models. Journal of Systems and Software, vol. 39, no. 3, 281-289, (December, 1997)
- [69] Gelman A, Carlin J, Stern H, Rubin D, (2003), Bayesian Data Analysis, Second Edition. Chapman & Hall/CRC
- [70] George B, Williams L, A structured experiment of test-driven development. Information and Software Technology, 46(5):337-342, 2004.
- [71] Geras A, Smith M, Miller J, A Prototype Empirical Evaluation of Test Driven Development, Proceedings of the 10th International Symposium on Software Metrics (METRICS'04)
- [72] Gittens M, Lutfiyya H, Bauer M, An Extended Operational Profile Model, Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
- [73] Glass RL, The Standish report: does it really describe a software crisis? Commun. ACM, ACM Press, 2006, 49, 15-16
- [74] Glass RL, IEEE Software, May/June 2005, Vol. 22 number 3, IT Failure Rates – 70% or 10-15%.
- [75] Goel AL, Okumoto K, (1979), Time dependent error detection rate model for software reliability and other performance measures. IEEE Transactions in Reliability R-28 , 206-211
- [76] Gras JJ, End-to-End Defect Modeling, IEEE SOFTWARE, Vol 21, no 5, pp 98-100, Sep/Oct 2004
- [77] Gulezian R, Utilizing COCOMO Inputs as a Basis for Developing Generalized Software Development Cost Estimation Model, May 1986, COCOMO/WICOMO Users' Group Meeting, Wang Institute Tyngsboro, MA.
- [78] Halstead M, Elements of Software Science, Elsevier, North Holland, 1977
- [79] Hamer PG, Frewin GD, M.H. Halstead's Software Science - a critical examination, Proceedings of the 6th international conference on Software engineering, Tokyo, Japan, 1982, Pages: 197 – 206

- [80] Harrison W, An Entropy-Based Measure of Software Complexity. IEEE Transactions on Software Engineering, 18(11):1025–1029, Nov. 1992.
- [81] Hassan AE, Holt RC. The chaos of software development. (Conference Paper) Proceedings. Sixth International Workshop on Principles of Software Evolution. IEEE Comput. Soc. 2003, pp.84-94. Los Alamitos, CA, USA.
- [82] Hay D, Healy K, Defining business rules - what are they really? Guide business rule project report, 1996
- [83] Hearty P, Fenton N, Neil M, Cates P, Automated population of causal models for improved software risk assessment. ASE 2005: 433-434
- [84] Heckerman D, A Tutorial on Learning With Bayesian Networks, Technical Report, Microsoft Research, November 1996
- [85] Heiberg S, Puus U, Salumaa P, Seeba A, Pair-Programming Effect on Developers Productivity, Proceedings of XP2003 (Springer LNCS 2675), pages 215 - 224, 2003
- [86] Hotle M, Understanding and Improving the AD Estimating Process, Applications Development and Management System Strategies, The Gartner Group, Stamford, Conn., November, 1996, pp. 25
- [87] Huang C, Darwiche A, Inference in belief networks: A procedural guide, International Journal of Approximate Reasoning, vol. 15, num. 3, 1996
- [88] Hulkko H, Abrahamsson P, A Multiple Case Study on the Impact of Pair Programming on Product Quality, 27th International Conference on Software Engineering, ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA
- [89] IFPUG, Function Point Counting Practices Manual , Release 4.1, International Function Points Users Group (IFPUG), Mequon, Wisconsin, USA, 1999
- [90] ISBSG, Worldwide Software Development - the Benchmark . International Software Benchmarking Standards Group - ISBSG, from www.isbsg.org (accessed 7 Feb 2007).
- [91] Jaeger RM, 1997, Complementary methods for research in education. 2nd Edition, pp. 589-608, Washington DC: American Educational Research Association
- [92] 589-608, Washington DC: American Educational Research Association
- [93] Jensen F, Bayesian Networks and Decision Graphs, Springer-Verlag, New York, 2001
- [94] Jensen FV, Olesen KG, Andersen SK, An algebra of Bayesian belief universes for knowledge based systems, Networks, 20(5), 637–659.
- [95] Jensen FV, Lauritzen SL, Olesen KG, Bayesian updating in causal probabilistic networks by local computation, Computational Statistics Quarterly, 4, 269–282.
- [96] Jones, C. Programmer Productivity, McGraw Hill, 1986.
- [97] Jones, C. Software sizing, IEE Review 45(4), 165-167, 1999.
- [98] Jones, C. Variations in Software Development Practices, IEEE Software, Vol. 20, No. 6, pp. 22-27
- [99] Jones CG, Test-Driven Development Goes To School, Consortium for Computing Sciences in Colleges, 2004
- [100] Jones R, Garbage Collection, John Wiley & Sons, Ltd, ISBN 0-471-94148-4
- [101] Jørgensen M, A review of studies on expert estimation of software development effort, Journal of Systems and Software, Volume 70, Issues 1-2, February 2004, Pages 37-60

- [102] Jørgensen M, Shepperd MJ, A Systematic Review of Software Development Cost Estimation Studies, *IEEE Transactions on Software Engineering*, 33(1), pp. 33-53, 2007
- [103] Jørgensen M, Sjøberg DIK, An effort prediction interval approach based on the empirical distribution of previous estimation accuracy, *Information & Software Technology*, 45(3), pp. 123-136, 2003
- [104] Kalman RE, A New Approach to Linear Filtering and Prediction Problems, *Transaction of the ASME—Journal of Basic Engineering*, pp. 35-45 (March 1960)
- [105] Karlsson J, Software requirements prioritizing, *Proc Int Conf Req Eng Colorado Springs, Colorado, USA*, (1996), pp. 110–116
- [106] Karlsson L, Thelin T, Regnell B, Berander P, Wohlin C, Pair-wise comparisons versus planning game partitioning—experiments on requirements prioritisation techniques, *Empirical Software Engineering*, Volume 12, Number 1, pp. 3-33, February 2007
- [107] Keirse D, Bates M. Please Understand Me: Character and Temperament Types: Prometheus Nemesis Book Company, 1998
- [108] Kellner MI, Madachy RJ, Raffo DM, "Software process simulation modeling: why? what? how?," *Journal of System and Software*, vol. 46, pp. 91-105, 1999 (Vol 46, issues 2-3 is a special issue on software process simulation)
- [109] Kemerer CF, An empirical validation of software cost estimation models, *Communications of the ACM* vol. 30, no. 5 (May 1987) 416-429
- [110] Kemerer CF, Reliability of Function Points Measurement. A Field Experiment, *Communications of the ACM*, Vol.36, No.2, pp.85-97, February 1993
- [111] Kitchenham BA, Pickard LM, and Linkman SJ, An Evaluation of Some Design Metrics, *Software Eng J.*, vol. 5, no. 1, pp. 50-58, 1990
- [112] Korkala M, Abrahamsson P, Kyllönen P, A Case Study on the Impact of Customer Communication on Defects in Agile Software Development, *Proceedings of AGILE 2006 Conference (AGILE'06)*
- [113] Kozlov AV, Koller D, Nonuniform dynamic discretization in hybrid networks, in Geiger D, Shenoy PP (eds.), *Uncertainty in Artificial Intelligence*, 13: 314–325, 1997
- [114] Kim JO, Mueller CW, Factor analysis: Statistical methods and practical issues, Sage University Paper Series on Quantitative Applications in the Social Sciences, series no. 07-014, 1978, Newbury Park, CA: Sage
- [115] Kjaerulff U, Triangulation of graphs - algorithms giving small total state space, Technical Report R-90-09, Dept. of Math. and Comp. Sci., Aalborg University, Denmark, 1990
- [116] Krause PJ, Learning probabilistic networks, *The Knowledge Engineering Review*, Volume 13, Issue 04, pp. 321-351
- [117] Kuppuswami, S., Vivekanandan K., and Paul Rodrigues (2003): A System Dynamics Simulation Model to Find the Effects of XP on Cost of Change Curve. In proceedings of Fourth International Conference on Extreme Programming and Agile process in Software Engineering, (XP2003), May 25-29, 2003, Genova, Italy
- [118] Lay D, *Linear Algebra and It's Applications*, Addison-Wesley, New York, 2000.
- [119] Lauritzen SL, Jensen F, Stable local computation with conditional Gaussian Distributions, *Statistics and Computing*, 11, 2001, 191–203

- [120] Lauritzen SL, Spiegelhalter DJ, Local computations with probabilities on graphical structures and their application to expert systems (with discussion). *J.R. Statistical Soc. Series B*, 50, no. 2, pp. 157-224, 1988
- [121] Levendel Y, 1989, Defects and Reliability Analysis of Large Software Systems: Field Experience, Proc. 19th IEEE International Symposium on Fault-Tolerant Computing, Chicago, June 1989, pp. 238-244
- [122] Lepar V, Shenoy PP, A Comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer Architectures for Computing Marginals of Probability Distributions, in Cooper GF, Moral S. (eds.), *Uncertainty in Artificial Intelligence*, Vol. 14, 1999, pp. 328--337, Morgan Kaufmann, San Francisco, CA
- [123] Lethbridge TC, Laganriere R, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2Rev Ed edition (31 Dec 2004)
- [124] Li W, Another metric suite for object-oriented programming, *The Journal of Systems and Software* 1998; 44(2):155–162
- [125] Little T, Value creation and capture: a model of the software development process, *IEEE Software*, Vol 21, issue 3, pp 48-53
- [126] Little T, Schedule Estimation and Uncertainty Surrounding the Cone of Uncertainty, *IEEE SOFTWARE* May/June 2006
- [127] Littlewood B, The problems of assessing software reliability...when you really need to depend on it. In *Proceedings of the 8th Safety Critical Systems Symposium (SCSS'00)*, Southampton, UK, 2000.
- [128] Lui KM, Chan KCC, “When Does a Pair Outperform Two Individuals?” *XP2003*, Italy, 2003
- [129] Lyu, M. (ed.), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press 1996
- [130] McCabe T, “A software complexity measure”, *IEEE Transactions on Software Engineering*, SE-2(4), pp. 308-20, 1976
- [131] MacCormack A, Kemerer CF, Cusumano M, Crandall B, Trade-offs between Productivity and Quality in Selecting Software Development Practices, *IEEE Software*, September/October 2003 (Vol. 20, No. 5), pp. 78-85
- [132] Martin A, Biddle R, Noble J, The XP Customer Role in Practice: Three Studies, *Proceedings of the Agile Development Conference (ADC'04)*
- [133] Maximilien EM, Williams L, “Assessing Test-Driven Development at IBM,” *Proc. Int'l Conf. Software Eng. (ICSE)*, 2003
- [134] Melnik G, Maurer F, Introducing agile methods: three years of experience, *Proceedings 30th Euromicro Conference*, 2004, pp. 334-341
- [135] Minana EP, Gras JJ, Improving fault prediction using Bayesian networks for the development of embedded software applications, *Software Testing, Verification And Reliability* 2006, 16:157–174
- [136] Mistic, V., Gevaert, H., Rennie M. (2002) “Extreme dynamics: modelling the extreme programming software development process ”. *Workshop on empirical evaluation of agile processes, XP/Agile Universe* 2002
- [137] Molokken K, Jorgensen M, A review of software surveys on software effort estimation, 2003 *International Symposium on, Empirical Software Engineering*, ISESE 2003 223-230
- [138] Mooney JD, Portability and reusability: common issues and differences, February 1995, *Proceedings of the 1995 ACM 23rd annual conference on Computer science*

- [139] Moore R, Reff K, Graham J, Hackerson B, Scrum at a Fortune 500 Manufacturing Company, AGILE 2007, pp. 175-180
- [140] Muller M. Are Reviews an Alternative to Pair Programming? Empirical Software Engineering 2004;9(4):335-51
- [141] Mulzer W, Rote G, Minimum weight triangulation is NP-hard. In Proceedings of the Twenty-Second Annual Symposium on Computational Geometry (Sedona, Arizona, USA, June 05 - 07, 2006), SCG '06.
- [142] Munson JC, Khoshgoftaar TM, Regression Modelling of Software Quality: An Empirical Investigation, Information and Software Technology, vol. 32, no. 2, pp. 106-114, 1990
- [143] Munson JC, Kahshgoftaar TM, Measuring data structure complexity, J. Systems & Software, vol. 20, 217-225, 1993
- [144] Murphy KP, Dynamic Bayesian Networks: Representation, Inference and Learning, PhD thesis, UC Berkeley, 2002
- [145] Musa JD, Software Reliability Data, Data & Analysis Center for Software, January 1980. <http://www.dacs.dtic.mil/databases/sled/swrel.shtml>
- [146] Musa JD, Iannino A, Okumoto K, Software Reliability: Measurement, Prediction, Application, Mcgraw-Hill College, ISBN 007044093X
- [147] Myers I, Manual: The Myers-Briggs Type Indicator, Palo Alto, California: Consulting Psychologists Press, 1975.
- [148] Nawrocki J, Wojciechowski A, Experimental Evaluation of Pair Programming, Proceedings of the 12th European Software Control and Metrics Conference, pp. 269-276, 2001
- [149] Neapolitan RE, Learning Bayesian networks, Pearson Prentice Hall, 2004
- [150] Neil M, Statistical Modelling of Software Metrics, Ph.D. dissertation, South Bank University, December 1992
- [151] Neil M, Fenton N, Improved Software Defect Prediction. 10th European SEPG, London, 2005
- [152] Neil M, Krause P, Fenton NE, Software Quality Prediction Using Bayesian Networks in Software Engineering with Computational Intelligence, (Ed Khoshgoftaar TM), Kluwer, ISBN 1-4020-7427-1, Chapter 6, 2003
- [153] Neil M, Tailor M, Marquez D, Inference in hybrid Bayesian Networks using dynamic discretisation, accepted for publication in Statistics and Computing.
- [154] Nosek JT, The case for collaborative programming, Communications of the ACM, Volume 41, Issue 3 (March 1998) Pages: 105 – 108
- [155] Object Management Group, Business Semantics of Business Rules Request For Proposal, OMG Document: br/2003-06-03
- [156] Padberg F, Muller M, Analyzing the Cost and Benefit of Pair Programming, Proceedings of the Ninth International Software Metrics Symposium (METRICS'03)
- [157] Pancur M, Ciglaric M, Trampus M, Vidmar T, Towards empirical evaluation of test-driven development in a university environment, presented at EUROCON 2003. Computer as a Tool. The IEEE Region 8, 2003
- [158] Parrish A, Smith R, Hale D, Hale J, A Field Study of Developer Pairs: Productivity Impacts and Implications, Sep/Oct 2004 IEEE SOFTWARE
- [159] Pearl J, Fusion, Propagation, and Structuring in Belief Networks, Artificial Intelligence, Vol. 29, No. (3), pages 241-288
- [160] Pearl J, Causality: Models, Reasoning, and Inference, Cambridge University Press, 2000

- [161] Pearl J, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference Morgan Kaufman, 1988. (Revised in 1997)
- [162] Pelrine J, Modelling infection scenarios – a fixed-price eXtreme Programming success story, OOPSLA 2000 Minneapolis, Addendum, pp. 23-24
- [163] phpMyAdmin Project, http://www.phpmyadmin.net/home_page/index.php, accessed 9th Aug 2007
- [164] Piwoworski P, A nesting level complexity measure, Sigplan Notices, 17(9), Sept. 1981, pp. 44-50.
- [165] Prechelt L, The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical Report 1999-18, 25 pages, Universität Karlsruhe, Fakultät für Informatik, Germany, December 1999
- [166] Prechelt L, Unger B, Philippsen M, Tichy WF, A controlled experiment on inheritance depth as a cost factor for code maintenance, Journal of Systems and Software 65(2) p. 115-126, 2003
- [167] Rabiner L, Juang B, An introduction to hidden Markov models, ASSP Magazine, IEEE [see also IEEE Signal Processing Magazine] 3, no. 1: 16, 4
- [168] Ratcliff B, Rollo AL, Adapting function point analysis to Jackson system development, Software Engineering Journal, v.5 n.1, p.79-84, Jan. 1990
- [169] Rees K, Coolen FPA, Goldstein M, Wooff DA, Managing the uncertainties of software testing: a Bayesian approach. Quality and Reliability Engineering International 17: 191-203.
- [170] Riggelsen C, Learning Bayesian Networks from Incomplete Data: An Efficient Method for Generating Approximate Predictive Distributions, SIAM 2006
- [171] Rising L, Janoff NS, The Scrum Software Development Process for Small Teams, IEEE Software Vol 17/4, pp. 26-32, 2002
- [172] Ross SA, Fundamentals of Corporate Finance, Irwin/McGraw-Hill, 1996
- [173] Royce W, Managing the Development of Large Software Systems, Proc. IEEE Wescon, 1970, pp. 1-9
- [174] Ruiz M, Ramos I, Toro M, A simplified model of software project dynamics, Journal of Systems and Software, vol. 59, no. pp. 299-309, 2001
- [175] Sackman H, Erikson WJ, Grant EE, Exploratory experimental studies comparing online and offline programming performance, Communications of the ACM, 11(1):3–11, January 1968
- [176] Schalliol G, "Challenges for Analysts on a Large XP Project," in Proceedings of XP/Agile Universe, Raleigh, NC, 2001
- [177] Schneider S, Concurrent and Real Time Systems: The CSP Approach (Worldwide Series in Computer Science), Wiley 1999
- [178] Schwaber K, Advanced Development Methods. SCRUM Development Process, <http://jeffsutherland.com/oopsla/schwapub.pdf>, accessed 28 Feb 2008
- [179] Schwaber K, Beedle M, Agile Software Development with SCRUM, Prentice Hall, 18 Feb 2002
- [180] Schwaber K, Agile software development with scrum, scrum faq, <http://www.scrum-master.com/resources/conchango%20scrum%20faq%20by%20ken%20schwaber.pdf>, accessed 29 Feb 2008
- [181] Settas D, Bibi S, Sfetsos P, Stamelos I, Gerogiannis VC, Using Bayesian Belief Networks to Model Software Project Management Antipatterns. SERA 2006: 117-124
- [182] Sfetsos P, Stamelos I, Angelis L, Deligiannis I, Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair

- Programming - An Empirical Study, 7th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2006). June 2006, Oulu, Finland.
- [183] Sfetos P, Angelis L, Stamelos I, Investigating the extreme programming system—An empirical study, *Empirical Software Engineering* Vol 11(2):269-301, 2006
 - [184] Sfetos P, Stamelos I, Angelis L, Deligiannis I. Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair Programming - An Empirical Study. *Extreme Programming and Agile Processes in Software Engineering*, 2006;43-52
 - [185] Shen VY, Conte SD, Dunsmore H, “Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support”, *IEEE Transactions on Software Engineering*, Vol. SE-9.,No. 2, pp. 155-165, March 1983
 - [186] Shenoy PP, Shafer G, Axioms for probability and belief-function propagation,” in R. D. Shachter, T. S. Levitt, Lemmer JF, Kanal LN (eds), *Uncertainty in Artificial Intelligence*, 4, 169–198, North-Holland, Amsterdam.
 - [187] Shepperd MJ, A critique of cyclomatic complexity as a software metric, *Software Engineering Journal*, March 1988, pp. 30-36
 - [188] Shepperd M, Schofield C, Estimating Software Project Effort Using Analogies. *IEEE Transactions on Software Engineering*, vol. 23, no 12, 736-743, (1997)
 - [189] Software Process Change, Proceedings of the International Software Process Workshop and International Workshop on Software Process Simulation and Modeling, SPW/ProSim 2006, Shanghai, China, May 20-21, 2006. Springer Lecture Notes in Computer Science, Volume 3966/2006, DOI 10.1007/11754305
 - [190] Sommerville, *Software Engineering*, Addison-Wesley 1992
 - [191] Srinivasan B, Is nested control more complex? *Sigplan Notices*, 18(12), Dec. 1983, pp., 120-121
 - [192] Srinivasan K, Fisher D. Machine Learning Approaches to Estimating Software Development Effort. *IEEE Transactions on Software Engineering*, vol. 21, no. 2, (1995)
 - [193] Stamelos, L. Angelis, P. Dimou, and E. Sakellaris. On the use of bayesian belief networks for the prediction of software productivity. *Information & Software Technology*, 45(1):51–60, 2003
 - [194] Standish Group, *The Chaos Report*. 1994, The Standish Group
 - [195] Stanford Encyclopedia of Philosophy, <http://plato.stanford.edu/entries/bayes-theorem/supplement.html>, accessed 14 Nov 2007
 - [196] Stephens M, Rosenberg D, *Extreme Programming Refactored: The Case Against XP*, Apress; 1 edition (August 5, 2003)
 - [197] Stroud JM, "The Fine Structure of Psychological Time", *Annals of New York Academy of Science*, Vol. 138, No. 2, pp 623 -631, 1967
 - [198] Sulaiman T, Barton B, Blackburn T, *AgileEVM - Earned Value Management in Scrum Projects*, agile, pp. 7-16, AGILE 2006 (AGILE'06), 2006
 - [199] Sutherland J, *Agile Development: Lessons learned from the first scrum*, October 2004, <http://jeffsutherland.com/Scrum/FirstScrum2004.pdf>, accessed 28 Feb 2008
 - [200] Symons CR, *Function Point Analysis: Difficulties and Improvements*, *IEEE Transactions on Software Engineering*, 1985

- [201] Takeuchi H, Nonaka I, The New New Product Development Game, Harvard Business Review, Jan-Feb 1986
- [202] Tatikonda MV, Rosenthal SR, Technology Novelty, Project Complexity, and Product Development Project Execution Success: A Deeper Look at Task Uncertainty in Product Innovation, IEEE Transactions on Engineering Management, vol. 47, no. 1, february 2000
- [203] Vanhanen J, Lassenius C, Effects of Pair Programming at the Development Team Level: An Experiment, Proceedings of International Symposium on Empirical Software Engineering (ISESE 2005)
- [204] Verner J, Tate G, Estimating Size and Effort in Fourth-Generation Development, IEEE Software, v.5 n.4, p.15-22, July 1988
- [205] Walston CE, Felix CP, A Method of Programming Measurement and Estimation. IBM Systems Journal, Vol. 16, No. 1, pp. 54-73, 1977. Also in: Tutorial on Programming Productivity: Issues for the Eighties, IEEE Computer Society, Second Edition, 1986
- [206] Wang H, Peng F, Zhang C, Pietschker A, Software Project Level Estimation Model Framework based on Bayesian Belief Networks, Sixth International Conference on Quality Software (QSIC'06)
- [207] Wang X, Sun J, Yang X, He Z, Maddineni SR, Human factors in extracting business rules from legacy systems, 2004 IEEE International Conference on Systems, Man and Cybernetics, Volume 1, 10-13 Oct. 2004 Page(s):200 - 205 vol.1
- [208] Watson H, McCabe T, NIST Special Publication 500-235*, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, <http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm>
- [209] Welch G, Bishop G, An Introduction to the Kalman Filter, SIGGRAPH 2001 Course
- [210] Whittaker JA, Toward a More Reliable Theory of Software Reliability, IEEE Computer, December 2000
- [211] Williams, L. and Erdogmus, H., On the Economic Feasibility of Pair Programming, International Workshop on Economics-Driven Software Engineering in conjunction with the International Conference on Software Engineering, May 2002.
- [212] Williams L, Kessler R, Pair Programming Illuminated, Addison-Wesley, 2003
- [213] Williams L, Kessler RR, Cunningham W, Jeffries R, Strengthening the Case for Pair Programming, IEEE Software, July/August 2000 (Vol. 17, No. 4) pp. 19-25
- [214] Williams L, McDowell C, Nagappan N, Fernald J, Werner L, Building Pair Programming Knowledge through a Family of Experiments, Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)
- [215] Williams L, Shukla A, Antón AI, An Initial Exploration of the Relationship Between Pair Programming and Brooks' Law, Proceedings of the Agile Development Conference (ADC'04)
- [216] Williams TM, The need for new paradigms for complex projects, International Journal of Project Management, Vol. 17, No. 5, pp. 269-273, 1999.
- [217] Wolverton RW, The Cost of Developing Large-Scale Software. IEEE Transactions on Computer, Volume C-23, No. 6, pp. 615-636, June 1974. Also in: Tutorial on Programming Productivity: Issues for the Eighties, IEEE Computer Society, Second Edition, 1986

- [218] Wood WA, Kleb WL, "Exploring XP for Scientific Research," IEEE Software, vol. 20, pp. 30 - 36, 2003.
- [219] Wooff DA, Goldstein M, Coolen FPA, Bayesian Graphical Models for Software Testing, IEEE Transactions on Software Engineering, Vol 28, Issue 5, pp. 510-525
- [220] Xia F, An Information Coding Based Data Complexity Model, 3rd International Software Metrics Symposium (METRICS '96) From Measurement to Empirical Results, 1996
- [221] Xia W, Lee G, Complexity of Information Systems Development Projects: Conceptualization and Measurement Development J. Manage. Inf. Syst., M. E. Sharpe, Inc., 2005, 22, 45-83
- [222] Yang D, Wan Y, Tang Z, Wu S, He M, Li M, COCOMO-U: An Extension of COCOMO II for Cost Estimation with Uncertainty. Software Process Change, 2006;132-41
- [223] Zhang H, Kitchenham B, Semi-quantitative Simulation Modeling of Software Engineering Process, pp. 242-253, in Software Process Change, Proceedings of the International Software Process Workshop and International Workshop on Software Process Simulation and Modeling, SPW/ProSim 2006, Shanghai, China, May 20-21, 2006. Springer Lecture Notes in Computer Science, Volume 3966/2006, DOI 10.1007/11754305
- [224] Zweig G, A forward-backward algorithm for inference in Bayesian networks and an empirical comparison with HMMs, Master's thesis, Dept. Comp. Sci., U.C. Berkeley, 1996.